

Mastering Python: A Comprehensive Guide Through the Official Tutorial

This book provides an in-depth guide to Python, walking readers through the official documentation. It's structured to help new Python programmers understand language essentials, from basic syntax to advanced concepts like modules, exceptions, and advanced classes. A perfect companion for those eager to learn Python through its well-documented features and examples.

Table of Contents

1. Whetting Your Appetite

- [1. Whetting Your Appetite](#)

2. Using the Python Interpreter

- [2. Using the Python Interpreter](#)
- [2.1. Invoking the Interpreter](#)
- [2.1.1. Argument Passing](#)
- [2.1.2. Interactive Mode](#)
- [2.2. The Interpreter and Its Environment](#)
- [2.2.1. Source Code Encoding](#)

3. An Informal Introduction to Python

- [3. An Informal Introduction to Python](#)
- [3.1. Using Python as a Calculator](#)
- [3.1.1. Numbers](#)
- [3.1.2. Text](#)
- [3.1.3. Lists](#)
- [3.2. First Steps Towards Programming](#)

4. More Control Flow Tools

- [4. More Control Flow Tools](#)
- [4.1. `if` Statements](#)
- [4.2. `for` Statements](#)
- [4.3. The `range\(\)` Function](#)
- [4.4. `break` and `continue` Statements](#)
- [4.5. `else` Clauses on Loops](#)
- [4.6. `pass` Statements](#)
- [4.7. `match` Statements](#)
- [4.8. Defining Functions](#)
- [4.9. More on Defining Functions](#)
- [4.9.1. Default Argument Values](#)
- [4.9.2. Keyword Arguments](#)
- [4.9.3. Special parameters](#)
- [4.9.3.1. Positional-or-Keyword Arguments](#)
- [4.9.3.2. Positional-Only Parameters](#)
- [4.9.3.3. Keyword-Only Arguments](#)
- [4.9.3.4. Function Examples](#)
- [4.9.3.5. Recap](#)
- [4.9.4. Arbitrary Argument Lists](#)
- [4.9.5. Unpacking Argument Lists](#)
- [4.9.6. Lambda Expressions](#)
- [4.9.7. Documentation Strings](#)
- [4.9.8. Function Annotations](#)
- [4.10. Intermezzo: Coding Style](#)

5. Data Structures

- 5. Data Structures
- 5.1. More on Lists
- 5.1.1. Using Lists as Stacks
- 5.1.2. Using Lists as Queues
- 5.1.3. List Comprehensions
- 5.1.4. Nested List Comprehensions
- 5.2. The `del` statement
- 5.3. Tuples and Sequences
- 5.4. Sets
- 5.5. Dictionaries
- 5.6. Looping Techniques
- 5.7. More on Conditions
- 5.8. Comparing Sequences and Other Types

6. Modules

- 6. Modules
- 6.1. More on Modules
- 6.1.1. Executing modules as scripts
- 6.1.2. The Module Search Path
- 6.1.3. “Compiled” Python files
- 6.2. Standard Modules
- 6.3. The `dir()` Function
- 6.4. Packages
- 6.4.1. Importing `*` From a Package
- 6.4.2. Intra-package References
- 6.4.3. Packages in Multiple Directories

7. Input and Output

- 7. Input and Output
- 7.1. Fancier Output Formatting
- 7.1.1. Formatted String Literals
- 7.1.2. The String `format()` Method
- 7.1.3. Manual String Formatting
- 7.1.4. Old string formatting
- 7.2. Reading and Writing Files
- 7.2.1. Methods of File Objects
- 7.2.2. Saving structured data with `json`

8. Errors and Exceptions

- 8. Errors and Exceptions
- 8.1. Syntax Errors
- 8.2. Exceptions
- 8.3. Handling Exceptions
- 8.4. Raising Exceptions
- 8.5. Exception Chaining
- 8.6. User-defined Exceptions
- 8.7. Defining Clean-up Actions
- 8.8. Predefined Clean-up Actions
- 8.9. Raising and Handling Multiple Unrelated Exceptions
- 8.10. Enriching Exceptions with Notes

9. Classes

- 9. Classes
- 9.1. A Word About Names and Objects
- 9.2. Python Scopes and Namespaces
- 9.2.1. Scopes and Namespaces Example
- 9.3. A First Look at Classes
- 9.3.1. Class Definition Syntax
- 9.3.2. Class Objects
- 9.3.3. Instance Objects
- 9.3.4. Method Objects
- 9.3.5. Class and Instance Variables
- 9.4. Random Remarks
- 9.5. Inheritance
- 9.5.1. Multiple Inheritance
- 9.6. Private Variables
- 9.7. Odds and Ends
- 9.8. Iterators
- 9.9. Generators
- 9.10. Generator Expressions

10. Brief Tour of the Standard Library

- 10. Brief Tour of the Standard Library
- 10.1. Operating System Interface
- 10.2. File Wildcards
- 10.3. Command Line Arguments
- 10.4. Error Output Redirection and Program Termination
- 10.5. String Pattern Matching
- 10.6. Mathematics
- 10.7. Internet Access
- 10.8. Dates and Times
- 10.9. Data Compression
- 10.10. Performance Measurement
- 10.11. Quality Control
- 10.12. Batteries Included

11. Brief Tour of the Standard Library — Part II

- 11. Brief Tour of the Standard Library — Part II
- 11.1. Output Formatting
- 11.2. Templating
- 11.3. Working with Binary Data Record Layouts
- 11.4. Multi-threading
- 11.5. Logging
- 11.6. Weak References
- 11.7. Tools for Working with Lists
- 11.8. Decimal Floating-Point Arithmetic

12. Virtual Environments and Packages

- 12. Virtual Environments and Packages
- 12.1. Introduction
- 12.2. Creating Virtual Environments
- 12.3. Managing Packages with pip

13. What Now?

- [13. What Now?](#)

14. Interactive Input Editing and History Substitution

- [14. Interactive Input Editing and History Substitution](#)
- [14.1. Tab Completion and History Editing](#)
- [14.2. Alternatives to the Interactive Interpreter](#)

15. Floating-Point Arithmetic: Issues and Limitations

- [15. Floating-Point Arithmetic: Issues and Limitations](#)
- [15.1. Representation Error](#)

16. Appendix

- [16. Appendix](#)
- [16.1. Interactive Mode](#)
- [16.1.1. Error Handling](#)
- [16.1.2. Executable Python Scripts](#)
- [16.1.3. The Interactive Startup File](#)
- [16.1.4. The Customization Modules](#)

Whetting Your Appetite

If you do much work on computers, eventually you find that there's some task you'd like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you'd like to write a small custom database, or a specialized GUI application, or a simple game.

If you're a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you're writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you've written a program that could use an extension language, and you don't want to design and implement a whole new language for your application.

Python is just the language for you.

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, macOS, and Unix operating systems, and will help you get the job done more quickly.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a *very-high-level language*, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is *extensible*: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

By the way, the language is named after the BBC show “Monty Python's Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

Now that you are all excited about Python, you'll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.

Using the Python Interpreter — Python 3.13.5 documentation

Table of Contents

- [2. Using the Python Interpreter](#)
 - [2.1. Invoking the Interpreter](#)
 - [2.1.1. Argument Passing](#)
 - [2.1.2. Interactive Mode](#)
 - [2.2. The Interpreter and Its Environment](#)
 - [2.2.1. Source Code Encoding](#)

Previous topic

[1. Whetting Your Appetite](#)

Next topic

[3. An Informal Introduction to Python](#)

This page

- [Report a bug](#)
- [Show source](#)

Navigation

- [index](#)
- [modules](#)
- [next](#)
- [previous](#)
- [Python »](#)
- [3.13.5 Documentation](#)
- [The Python Tutorial](#)
- [2. Using the Python Interpreter](#)

2. Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the [Microsoft Store](#), the `python3.13` command will be available. If you have the [py.exe launcher](#) installed, you can use the `py` command. See [Setting environment variables](#) for other ways to launch Python.

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features include interactive editing, history substitution, and code completion on systems that support the [GNU Readline](#) library. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see [Interactive Input Editing and History Substitution](#) for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

All command line options are described in [Command line and environment](#).

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c` command is used, `sys.argv[0]` is set to `'-c'`. When `-m` module is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c` or `-m` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually `>>>`; for continuation lines it prompts with the *secondary prompt*, by default `...`. The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
```

Type "help", "copyright", "credits" or "license" for more information.
>>>

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

```
the_world_is_flat = True
if the_world_is_flat:
    print("Be careful not to fall off!")
```

For more on interactive mode, see [Interactive Mode](#).

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers, and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid [codecs](#) supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a [UNIX “shebang” line](#). In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Footnotes

[1]: On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

Footer

© [Python Software Foundation](#)

Copyright 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.
[Please donate.](#)

Last updated on Jul 05, 2025 (07:10 UTC).
[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

Using the Python Interpreter — Python 3.13.5 documentation

2. Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the [Microsoft Store](#), the `python3.13` command will be available. If you have the [py.exe launcher](#), you can use the `py` command. See [setting environment variables](#) for other ways to launch Python.

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support the [GNU Readline](#) library. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see [Interactive Input Editing and History Substitution](#) for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

All command line options are described in [Command line and environment](#).

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `-` (meaning standard input),

`sys.argv[0]` is set to `-`. When `-c` command is used, `sys.argv[0]` is set to `-c`. When `-m module` is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c command` or `-m module` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually `>>>`; for continuation lines it prompts with the *secondary prompt*, by default `...`. The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this [if statement](#):

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

For more on interactive mode, see [Interactive Mode](#).

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid [codecs](#) supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a [UNIX “shebang” line](#). In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Footnotes

[1] On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

Using the Python Interpreter — Python 3.13.5 documentation

Table of Contents

- [2. Using the Python Interpreter](#)
 - [2.1. Invoking the Interpreter](#)
 - [2.1.1. Argument Passing](#)
 - [2.1.2. Interactive Mode](#)
 - [2.2. The Interpreter and Its Environment](#)
 - [2.2.1. Source Code Encoding](#)
-

Previous topic

[Whetting Your Appetite](#)

Next topic

[An Informal Introduction to Python](#)

This page

- [Report a bug](#)
 - [Show source](#)
-

Navigation

- [index](#)
 - [modules](#)
 - [next](#)
 - [previous](#)
 - [Python »](#)
 - [3.13.5 Documentation »](#)
 - [The Python Tutorial »](#)
 - [2. Using the Python Interpreter](#)
-

2. Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

`python3.13`

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the [Microsoft Store](#), the `python3.13` command will be available. If you have the [py.exe launcher](#) installed, you can use the `py` command. See [Setting environment variables](#) for other ways to launch Python.

Typing an end-of-file character (`Control-D` on Unix, `Control-Z` on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command:

```
quit()
```

The interpreter's line-editing features include interactive editing, history substitution, and code completion on systems that support the [GNU Readline](#) library. Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing; see [Interactive Input Editing and History Substitution](#) for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a filename argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is

```
python -c command [arg] ...
```

which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using

```
python -m module [arg] ...
```

which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterward. This can be done by passing the `-i` option before the script.

All command line options are described in [Command line and environment](#).

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing:

```
import sys
```

The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c` command is used, `sys.argv[0]` is set to `'-c'`. When `-m` module is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c` or `-m` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually `>>>`; for continuation lines it prompts with the *secondary prompt*, by default `...`. The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. For example, take a look at this [if](#) statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

For more on interactive mode, see [Interactive Mode](#).

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers, and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid [codecs](#) supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a [UNIX “shebang” line](#). In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Footnotes

[1] On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

Created using [Sphinx](#) 8.2.3.

Using the Python Interpreter — Python 3.13.5 documentation

Table of Contents

- [2. Using the Python Interpreter](#)
 - [2.1. Invoking the Interpreter](#)
 - [2.1.1. Argument Passing](#)
 - [2.1.2. Interactive Mode](#)
 - [2.2. The Interpreter and Its Environment](#)
 - [2.2.1. Source Code Encoding](#)
-

Previous topic

[^1]: [1. Whetting Your Appetite](#)

Next topic

[^2]: [3. An Informal Introduction to Python](#)

This page

- [Report a bug](#)
 - [Show source](#)
-

Navigation

- [index](#)
 - [modules](#) |
 - [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#)
 - [The Python Tutorial](#)
 - [2. Using the Python Interpreter](#)
-

2. Using the Python Interpreter¶

2.1. Invoking the Interpreter¶

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the [Microsoft Store](#), the `python3.13` command will be available. If you have the [py.exe launcher](#) installed, you can use the `py` command. See [Setting environment variables](#) for other ways to launch Python.

Typing an end-of-file character (`Control - D` on Unix, `Control - Z` on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command:

```
quit()
```

The interpreter's line-editing features include interactive editing, history substitution, and code completion on systems that support the [GNU Readline](#) library. Perhaps the quickest check to see whether command line editing is supported is typing `Control - P` at the first Python prompt you get. If it beeps, you have command line editing; see Appendix [Interactive Input Editing and History Substitution](#) for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a filename argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is:

```
python -c command [arg] ...
```

which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using:

```
python -m module [arg] ...
```

which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterward. This can be done by passing `-i` before the script.

All command line options are described in [Command line and environment](#).

2.1.1. Argument Passing¶

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c command` is used, `sys.argv[0]` is set to `'-c'`. When `-m module` is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c command` or `-m`

module are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode ¶

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (`>>>`); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

For more on interactive mode, see [Interactive Mode](#).

2.2. The Interpreter and Its Environment ¶

2.2.1. Source Code Encoding ¶

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers, and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid [codecs supported by Python](#).

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a [UNIX “shebang” line](#). In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

[^1]: On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

[^2]: [3. An Informal Introduction to Python](#)

Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the [Microsoft Store](#), the `python3.13` command will be available. If you have the [py.exe launcher](#), you can use the `py` command. See [Setting environment variables](#) for other ways to launch Python.

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features include interactive editing, history substitution, and code completion on systems that support the [GNU Readline](#) library. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see [Interactive Input Editing and History Substitution](#) for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

All command line options are described in [Command line and environment](#).

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c` command is used, `sys.argv[0]` is set to `'-c'`. When `-m` module is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c` or `-m` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (`>>>`); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

For more on interactive mode, see [Interactive Mode](#).

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers, and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid [codecs](#) supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a [UNIX “shebang” line](#). In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Footnotes

[1] On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

Using the Python Interpreter — Python 3.13.5 documentation

Table of Contents

- [2. Using the Python Interpreter](#)
 - [2.1. Invoking the Interpreter](#)
 - [2.1.1. Argument Passing](#)
 - [2.1.2. Interactive Mode](#)
 - [2.2. The Interpreter and Its Environment](#)
 - [2.2.1. Source Code Encoding](#)
-

Previous topic

[1. Whetting Your Appetite](#)

Next topic

[3. An Informal Introduction to Python](#)

This page

- [Report a bug](#)
 - [Show source](#)
-

Navigation

- [index](#)
 - [modules](#) |
 - [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#)
 - [The Python Tutorial](#) »
 - [2. Using the Python Interpreter](#)
-

2. Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.13` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.13
```

to the shell. [1] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the [Microsoft Store](#), the `python3.13` command will be available. If you have the [launcher](#) installed, you can use the `py` command. See [Setting environment variables](#) for other ways to launch Python.

Typing an end-of-file character (`Control - D` on Unix, `Control - Z` on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support the [GNU Readline](#) library. Perhaps the quickest check to see whether command line editing is supported is typing `Control - P` to the first Python prompt you get. If it beeps, you have command line editing; see Appendix [Interactive Input Editing and History Substitution](#) for an introduction to the keys. If nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is

```
python -c command [arg] ... ,
```

which executes the statement(s) in *command*, analogous to the shell's `-c` option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using

```
python -m module [arg] ... ,
```

which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing the `-i` [command line option](#) before the script.

All command line options are described in [Command line and environment](#).

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c` command line option is used, `sys.argv[0]` is set to `'-c'`. When `-m` option is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c` or `-m` are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually `>>>`; for continuation lines it prompts with the *secondary prompt*, by default `...`. The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this `if` statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

For more on interactive mode, see [Interactive Mode](#).

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid [codecs](#) supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a [UNIX “shebang” line](#). In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

© Python Software Foundation

Copyright 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
{'a', 'c', 'd', 'e', 'f', 'g', ...}'
```

The `pprint` module offers more sophisticated control over printing both built-in and user-defined objects in a way that is readable by the interpreter. When the result is longer than one line, the "pretty printer" adds line breaks and indentation to more clearly reveal data structure:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow']]]
>>> pprint.pprint(t, width=30)
[[['black', 'cyan'],
   'white',
   ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The `locale` module accesses a database of culture-specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv() # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x),
'$1,234,567.80')
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or keyword arguments. For mail-merge style applications, user-supplied data may be incomplete and `safe_substitute()` may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
# Raises KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=dict(date=date), n=i, f=ext)
...     print('{0} --&gt; {1}'.format(filename, newname))
img_1074.jpg --&gt; Ashley_0.jpg
img_1076.jpg --&gt; Ashley_1.jpg
img_1077.jpg --&gt; Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable-length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `"<"` indicates that they are standard size and in little-endian byte order:

```
import struct
```

```

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # skip to the next header

```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can improve the responsiveness of applications that accept user input while other tasks run in the background, or enable I/O to happen in parallel with computations.

The following code shows how the high level [threading](#) module can run tasks in the background while the main program continues:

```

import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join() # Wait for the background task to finish
print('Main program waited until background was done.')

```

The principal challenge of multi-threaded applications is coordinating threads that share data or resources. The [threading](#) module provides synchronization primitives such as locks, events, condition variables, and semaphores.

While these tools are powerful, minor errors can cause difficult-to-reproduce problems. The recommended approach is to route all access to shared resources through a single thread and communicate requests via the [queue](#) module:

```
import queue

# Example of thread-safe queue usage
q = queue.Queue()
# Producer thread
q.put(item)
# Consumer thread
item = q.get()
```

11.5. Logging

The [logging](#) module provides a full-featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces output similar to:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and output goes to standard error. Other options include routing messages through email, datagrams, sockets, or an HTTP server. Filters can route messages based on their severity:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

The logging configuration can be set programmatically or via an external configuration file.

11.6. Weak References

Python manages memory through reference counting and garbage collection. However, sometimes there is a need to keep track of objects only as long as they are needed by other parts of the program, without increasing reference counts.

The [weakref](#) module provides tools for referencing objects without creating strong references. When the last strong reference is gone, the object is automatically destroyed, and any associated weak references are invalidated.

Example:

```
import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
```

```

        return str(self.value)

a = A(10)                # create a reference
d = weakref.WeakValueDictionary()
d['primary'] = a          # does not create a strong reference
del a                    # remove the one reference
gc.collect()             # run garbage collection right away
print(d.get('primary'))   # entry was automatically removed
# Output: None

```

11.7. Tools for Working with Lists

Many data structure needs can be satisfied with the built-in list type. However, alternative implementations may offer performance advantages.

The [array](#) module provides an `array` object that stores homogeneous data more compactly:

```

from array import array

a = array('H', [4000, 10, 700, 22222])
sum(a) -> 26932
a[1:3] -> array('H', [10, 700])

```

The [collections](#) module offers a [deque](#) object, which is like a list with faster appends and pops from the left, suitable for queues and breadth-first searches:

```

from collections import deque

d = deque(["task1", "task2", "task3"])
d.append("task4")
print("Handling", d.popleft()) # Handling task1

```

Example of breadth-first search:

```

unsearched = deque([starting_node])

def breadth_first_search():
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)

```

In addition, the [bisect](#) module offers functions for maintaining sorted lists:

```

import bisect

scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
bisect.insort(scores, (300, 'ruby'))
# scores now is [(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'pyth

```

The [heapq](#) module provides heap operations based on lists:

```

from heapq import heapify, heappop, heappush

data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapify(data) # rearrange into heap order

```

```
heappush(data, -5) # add new entry
[min for _ in range(3)] = [heappop(data) for _ in range(3)] # fetch three smallest en
```

11.8. Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to binary floating point, it is especially useful for:

- Financial applications
- Control over precision
- Control over rounding
- Tracking significant decimal places
- Ensuring results match manual calculations

Example:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(0.70 * 1.05, 2)
0.73
```

The `Decimal` class maintains exact decimal representation, enabling precise modulo operations and comparisons:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995
>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == Decimal('1.0')
# False due to binary floating point issues
```

Arithmetic with specified precision:

```
>>> getcontext().prec = 36
>>> Decimal(1)/Decimal(7)
Decimal('0.142857142857142857142857142857')
```

— *End of Content*

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The [replib](#) module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import replib
>>> replib.repr(set('supercalifragilisticexpialidocious'))
{'a', 'c', 'd', 'e', 'f', 'g', ...}'
```

The [pprint](#) module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
>>> pprint.pprint(t, width=30)
[[['black', 'cyan'],
  'white',
  ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]
```

The [textwrap](#) module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
a list of strings instead of one big string with newlines to separate"""
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The [locale](#) module accesses a database of culture specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv() # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x),
'$1,234,567.80')
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `{ $ }` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('$village folk send $10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without

using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `"<"` indicates that they are standard size and in little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields
    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)
    start += extra_size + comp_size # skip to the next header
```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join() # Wait for the background task to finish
print('Main program waited until background was done.')
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the `threading` module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and

then use the [queue](#) module to feed that thread with requests from other threads. Applications using [Queue](#) objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5. Logging

The [logging](#) module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6. Weak References

Python does automatic memory management (reference counting for most objects and [garbage collection](#) to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent.

The [weakref](#) module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return str(self.value)

a = A(10) # create a reference
d = weakref.WeakValueDictionary()
d['primary'] = a # does not create a reference
del a # remove the one reference
gc.collect() # run garbage collection right away
print(d['primary']) # entry was automatically removed
```

Output: KeyError

11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the `bisect` module with functions for manipulating sorted lists:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The `heapq` module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.8. Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for:

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857')
```

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
import reprlib
reprlib.repr(set('supercalifragilisticexpialidocious'))
# Output: "{ 'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```
import pprint
t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
pprint.pprint(t, width=30)
# Output:
# [[['black', 'cyan'],
#     'white',
#     ['green', 'red']],
#  [['magenta', 'yellow'],
#     'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
import textwrap
doc = """The wrap() method is just like fill() except that it returns
a list of strings instead of one big string with newlines to separate"""
print(textwrap.fill(doc, width=40))
# Output:
# The wrap() method is just like fill()
# except that it returns a list of strings
# instead of one big string with newlines
# to separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

```
import locale
locale.setlocale(locale.LC_ALL, 'English_United States.1252')
conv = locale.localeconv() # get a mapping of conventions
x = 1234567.8
locale.format_string("%d", x, grouping=True) # '1,234,567'
locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x), grou
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
from string import Template
t = Template('${village}folk send $10 to $cause.')
t.substitute(village='Nottingham', cause='the ditch fund')
# Output: 'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
t = Template('Return the $item to $owner.')
d = dict(item='unladen swallow')
t.substitute(d) # KeyError: 'owner'
t.safe_substitute(d) # 'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
import time, os.path

photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']

class BatchRename(Template):
    delimiter = '%'

fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
t = BatchRename(fmt)

date = time.strftime('%d%b%y')
for i, filename in enumerate(photofiles):
    base, ext = os.path.splitext(filename)
    newname = t.substitute(d=dict(date=date), n=i, f=ext)
    print(f'{filename} --> {newname}')

# Output:
# img_1074.jpg --> Ashley_0.jpg
# img_1076.jpg --> Ashley_1.jpg
# img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `"<"` indicates that they are standard size and in little-endian byte order:

```
import struct
```

```

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # skip to the next header

```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level [threading](#) module can run tasks in background while the main program continues to run:

```

import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
background.join() # Wait for the background task to finish
print('Main program waited until background was done.')

```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the [queue](#) module to feed that thread with requests from other threads. Applications using [Queue](#) objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5. Logging

The [logging](#) module offers a full featured and flexible logging system.

At its simplest, log messages are sent to a file or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: [DEBUG](#), [INFO](#), [WARNING](#), [ERROR](#), and [CRITICAL](#).

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6. Weak References

Python does automatic memory management (reference counting for most objects and [garbage collection](#) to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The [weakref](#) module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return str(self.value)

a = A(10) # create a reference
d = weakref.WeakValueDictionary()
d['primary'] = a # does not create a reference
del a # remove the one reference
gc.collect() # run garbage collection right away
print(d['primary']) # entry was automatically removed
# Output will raise KeyError if object is gone
```

11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The [array](#) module provides an [array](#) object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
from array import array
a = array('H', [4000, 10, 700, 22222])
sum(a)
# Output: 26932
a[1:3]
# Output: array('H', [10, 700])
```

The [collections](#) module provides a [deque](#) object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
from collections import deque
d = deque(['task1', 'task2', 'task3'])
d.append('task4')
print("Handling", d.popleft()) # Handling task1
# Output: Handling task1
```

Example usage in a breadth-first search:

```
unsearched = deque([starting_node])

def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the [bisect](#) module with functions for manipulating sorted lists:

```
import bisect
scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
bisect.insort(scores, (300, 'ruby'))
# scores: [(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The [heapq](#) module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```
from heapq import heapify, heappop, heappush
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapify(data) # rearrange the list into heap order
heappush(data, -5) # add a new entry
smallest_three = [heappop(data) for _ in range(3)] # fetch the three smallest entries
# Output: [-5, 0, 1]
```

11.8. Decimal Floating-Point Arithmetic

The [decimal](#) module offers a [Decimal](#) datatype for decimal floating-point arithmetic. Compared to the built-in [float](#) implementation of binary floating point, the class is especially helpful for:

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```
from decimal import *
round(Decimal('0.70') * Decimal('1.05'), 2) # Decimal('0.74')
round(.70 * 1.05, 2) # 0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. `Decimal` reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
Decimal('1.00') % Decimal('.10') # Decimal('0.00')
1.00 % 0.10 # 0.09999999999999995
sum([Decimal('0.1')]*10) == Decimal('1.0') # True
0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0 # False
```

The `decimal` module provides arithmetic with as much precision as needed:

```
getcontext().prec = 36
Decimal(1) / Decimal(7)
# Output: Decimal('0.142857142857142857142857142857')
```

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The [reprlib](#) module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

The [pprint](#) module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structures:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', ... 'yell
>>> pprint.pprint(t, width=30)
[[['black', 'cyan'],
   'white',
   ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]]
```

The [textwrap](#) module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate"""
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The [locale](#) module accesses a database of culture specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv() # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x),
' $1,234,567.80'
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('$ {village}folk send $10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print(f'{filename} --&gt; {newname}')
img_1074.jpg --&gt; Ashley_0.jpg
img_1076.jpg --&gt; Ashley_1.jpg
img_1077.jpg --&gt; Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `<` indicates that they are standard size and in little-endian byte order:

```

import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # skip to the next header

```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level [threading](#) module can run tasks in background while the main program continues to run:

```

import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run.')

background.join() # Wait for the background task to finish
print('Main program waited until background was done.')

```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the [queue](#) module to feed that thread with requests from other threads. Applications using [Queue](#)

objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5. Logging

The [logging](#) module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6. Weak References

Python does automatic memory management (reference counting for most objects and [garbage collection](#) to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The [weakref](#) module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10) # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a # does not create a new reference
>>> del a # remove the one reference
>>> gc.collect() # run garbage collection right away
```

```

0
>>> d['primary'] # entry was automatically removed
Traceback (most recent call last):
...
KeyError: 'primary'

```

11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The [array](#) module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```

>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])

```

The [collections](#) module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```

>>> from collections import deque
>>> d = deque(['task1', 'task2', 'task3'])
>>> d.append('task4')
>>> print('Handling', d.popleft())
Handling task1

```

```

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)

```

In addition to alternative list implementations, the library also offers other tools such as the [bisect](#) module with functions for manipulating sorted lists:

```

>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]

```

The [heapq](#) module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```

>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry

```



```
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.8. Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for:

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```



Copyright 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation. [Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
import reprlib
reprlib.repr(set('supercalifragilisticexpialidocious'))
# Output: {'a', 'c', 'd', 'e', 'f', 'g', ...}
```

The `pprint` module offers more sophisticated control over printing both built-in and user-defined objects in a way that is readable by the interpreter. When the result is longer than one line, the "pretty printer" adds line breaks and indentation to more clearly reveal data structures:

```
import pprint
t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
pprint.pprint(t, width=30)
# Output:
# [[['black', 'cyan'],
#     'white',
#     ['green', 'red']],
#  [['magenta', 'yellow'],
#     'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
import textwrap
doc = """The wrap() method is just like fill() except that it returns
a list of strings instead of one big string with newlines to separate"""
print(textwrap.fill(doc, width=40))
# Output:
# The wrap() method is just like fill()
# except that it returns a list of strings
# instead of one big string with newlines
# to separate the wrapped lines.
```

The `locale` module accesses a database of culture-specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

```
import locale
locale.setlocale(locale.LC_ALL, 'English_United States.1252')
conv = locale.localeconv() # get a mapping of conventions
x = 1234567.8
locale.format_string("%d", x, grouping=True) # Output: '1,234,567'
locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x), grou
# Output: '$1,234,567.80'
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
from string import Template
t = Template('${village} folk send $10 to $cause.')
t.substitute(village='Nottingham', cause='the ditch fund')
# Output: Nottinghamfolk send $10 to the ditch fund.
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user-supplied data may be incomplete, and `safe_substitute()` may be more appropriate — it will leave placeholders unchanged if data is missing:

```
t = Template('Return the $item to $owner.')
d = dict(item='unladen swallow')
try:
    t.substitute(d)
except KeyError as e:
    print(e)
# Output: KeyError: 'owner'

print(t.safe_substitute(d))
# Output: Return the unladen swallow to $owner.
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
import time, os.path

class BatchRename(Template):
    delimiter = '%'

fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
t = BatchRename(fmt)
date = time.strftime('%d%b%y')
photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
for i, filename in enumerate(photofiles):
    base, ext = os.path.splitext(filename)
    newname = t.substitute(d=dict(date=date), n=i, f=ext)
    print(f'{filename} --> '{newname}''')
# Example output:
# img_1074.jpg --> Ashley_0.jpg
# img_1076.jpg --> Ashley_1.jpg
# img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `<` indicates standard size and little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # skip to the next header
```

11.4. Multi-threading

Threading is a technique for decoupling tasks that are not sequentially dependent. Threads can improve responsiveness when applications accept user input while other tasks run in the background. They are also used to run I/O in parallel with computations in another thread.

The high-level `threading` module can run tasks in the background while the main program continues:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
background.join() # Wait for the background task to finish
print('Main program waited until background was done.')
```

The principal challenge of multi-threaded applications is coordinating threads that share data or resources. The `threading` module provides synchronization primitives like locks, events, condition variables, and semaphores. The `queue` module enables thread-safe communication:

```
import queue
q = queue.Queue()
```

While powerful, minor design errors can cause tricky problems. A common approach is to centralize resource access in one thread and use the Queue module for requests:

```
from queue import Queue
```

11.5. Logging

The [logging](#) module offers a full-featured, flexible logging system. Log messages can be sent to files or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning: config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces output like:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed, and output goes to standard error. Messages can be routed through email, sockets, HTTP, etc. Filters can direct messages based on priority levels

```
[DEBUG](../library/logging.html#logging.DEBUG), [INFO]
(../library/logging.html#logging.INFO), [WARNING]
(../library/logging.html#logging.WARNING), [ERROR]
(../library/logging.html#logging.ERROR), [CRITICAL]
(../library/logging.html#logging.CRITICAL).
```

The logging system can be configured programmatically or via configuration files for customized logging.

11.6. Weak References

Python's automatic memory management (via reference counting and garbage collection) frees objects shortly after the last reference is gone. Sometimes, it's useful to track objects only as long as they are used elsewhere. The [weakref](#) module provides tools for tracking objects without creating strong references. When an object is no longer needed, it can be automatically removed from a weak reference table:

```
import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return str(self.value)

a = A(10) # create a reference
d = weakref.WeakValueDictionary()
d['primary'] = a # does not create a strong reference
del a # remove the last strong reference
gc.collect() # run garbage collection right away
print(d.get('primary')) # Entry was automatically removed: None
```

11.7. Tools for Working with Lists

Many data needs are met with Python's built-in list type, but alternative implementations can provide performance benefits.

The `array` module provides an array object storing homogeneous data compactly:

```
from array import array
a = array('H', [4000, 10, 700, 2222])
sum(a)
# Output: 26932
a[1:3]
# Output: array('H', [10, 700])
```

The `collections` module offers a `deque` object optimized for append/pop operations from both ends, useful for queues:

```
from collections import deque
d = deque(['task1', 'task2', 'task3'])
d.append('task4')
print('Handling', d.popleft())
# Output: Handling task1
```

```
# Example: Breadth-first search
unsearched = deque([starting_node])
def breadth_first_search():
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)
```

Other list tools include the `bisect` module for inserting into sorted lists:

```
import bisect
scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
bisect.insort(scores, (300, 'ruby'))
# scores now includes (300, 'ruby') in sorted position
```

The `heapq` module provides heap queue algorithms:

```
from heapq import heapify, heappop, heappush
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapify(data) # Rearrange into heap order
heappush(data, -5) # Add a new element
smallest_three = [heappop(data) for i in range(3)] # Fetch three smallest elements
# Output: [-5, 0, 1]
```

11.8. Decimal Floating-Point Arithmetic

The `decimal` module provides a `Decimal` data type for precise decimal floating-point arithmetic, useful for financial applications requiring exact decimal representation, control over precision and rounding, and matching hand calculations:

```
from decimal import *
round(Decimal('0.70') * Decimal('1.05'), 2)
# Output: Decimal('0.74')
```

```
round(.70 * 1.05, 2)
# Output: 0.73
```

`Decimal` results can keep trailing zeros, and the module supports modulo, equality, and more accurate arithmetic than binary floating point:

```
Decimal('1.00') % Decimal('.10')
# Output: Decimal('0.00')
Decimal('1.00') % 0.10
# Output: 0.09999999999999995
```

```
sum([Decimal('0.1')]*10) == Decimal('1.0')
# Output: True
```

```
0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
# Output: False
```

The `decimal` module allows setting arbitrary precision:

```
getcontext().prec = 36
Decimal(1)/Decimal(7)
# Output: Decimal('0.142857142857142857142857142857')
```


11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
import reprlib
reprlib.repr(set('supercalifragilisticexpialidocious'))
# Output: "{ 'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

The `pprint` module offers more sophisticated control over printing both built-in and user-defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```
import pprint
t = [[['black', 'cyan' ], 'white', [ 'green', 'red' ]], [[ 'magenta', 'yellow' ], 'b
pprint.pprint(t, width=30)
# Output:
# [[['black', 'cyan' ],
#    'white',
#    [ 'green', 'red' ]],
#   [[ 'magenta', 'yellow' ],
#    'blue']]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
import textwrap
doc = """The wrap() method is just like fill() except that it returns
a list of strings instead of one big string with newlines to separate
the wrapped lines."""
print(textwrap.fill(doc, width=40))
# Output:
# The wrap() method is just like fill()
# except that it returns a list of strings
# instead of one big string with newlines
# to separate the wrapped lines.
```

The `locale` module accesses a database of culture-specific data formats. The grouping attribute of locale’s `format` function provides a direct way of formatting numbers with group separators:

```
import locale
locale.setlocale(locale.LC_ALL, 'English_United States.1252')
conv = locale.localeconv() # get a mapping of conventions
x = 1234567.8
locale.format_string("%d", x, grouping=True) # Output: '1,234,567'
locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x), grou
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
from string import Template
t = Template('$ {village} folk send $10 to $cause.')
t.substitute(village='Nottingham', cause='the ditch fund')
# Output: 'Nottingham folk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied. For mail-merge applications where data may be incomplete, `safe_substitute()` leaves placeholders unchanged if data is missing:

```
t = Template('Return the $item to $owner.')
d = dict(item='unladen swallow')
t.substitute(d)
# Raises KeyError: 'owner'
t.safe_substitute(d)
# Output: 'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility might use `%` signs:

```
import time, os.path

photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']

class BatchRename(Template):
    delimiter = '%'

fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
t = BatchRename(fmt)

date = time.strftime('%d%b%y')
for i, filename in enumerate(photofiles):
    base, ext = os.path.splitext(filename)
    newname = t.substitute(d=date, n=i, f=ext)
    print(f'{filename} --> {newname}')

# Example output:
# img_1074.jpg --> Ashley_0.jpg
# img_1076.jpg --> Ashley_1.jpg
# img_1077.jpg --> Ashley_2.jpg
```

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable-length binary record formats. For example, to loop through header information in a ZIP file without using the `zipfile`

module, use pack codes `"H"` and `"I"` for two and four-byte unsigned numbers, respectively, with `<` indicating little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # skip to the next header
```

11.4. Multi-threading ¶

Threading decouples tasks which are not sequentially dependent. Threads can improve responsiveness in applications accepting user input while other tasks run in the background. One use case is running I/O in parallel with computations.

The `threading` module can run tasks in background while the main program continues:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
background.join() # Wait for the background task to finish
print('Main program waited until background was done.')
```

Coordination of shared data involves synchronization primitives like locks, events, condition variables, and semaphores. Preferably, use `queue` for inter-thread communication:

```
from queue import Queue
```

```
# Example of a simple consumer/producer pattern
q = Queue()
```

```
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        process(item)
```

```
# Or use in algorithms like breadth-first search:
unsearched = deque([starting_node])
def breadth_first_search():
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)
```

And for sorted lists, the `bisect` module provides functions for efficient insertion:

```
import bisect
scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
bisect.insort(scores, (300, 'ruby'))
# scores = [(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The `heapq` module provides functions for implementing heaps:

```
from heapq import heapify, heappop, heappush
```

```
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapify(data)           # rearrange into heap order
heappush(data, -5)      # add a new entry
smallest_three = [heappop(data) for _ in range(3)] # fetch the three smallest entries
# Result: [-5, 0, 1]
```

11.5. Logging

The `logging` module offers a full-featured and flexible logging system. Basic usage:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Sample output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Logs are sent to standard error by default; other handlers include email, datagrams, sockets, or HTTP. Filters enable routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

Configuration can be set programmatically or via a user-defined configuration file.

11.6. Weak References¶

Python manages memory automatically with reference counting and garbage collection. Sometimes, it's useful to track objects *only* as long as they are being used elsewhere. The `weakref` module provides tools to create references that do not increase reference counts. When the object is no longer needed, it is automatically removed from the weak reference table, and a callback may be invoked:

```
import weakref, gc

class A:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return str(self.value)

a = A(10)                # create a reference
d = weakref.WeakValueDictionary()
d['primary'] = a          # does not create a strong reference
del a                    # remove the last strong reference
gc.collect()              # run garbage collection
print(d['primary'])       # Entry was automatically removed: KeyError
```

11.7. Tools for Working with Lists¶

Built-in list types are often sufficient, but alternative implementations can optimize performance.

The `array` module provides an `array` object storing homogeneous data more compactly:

```
from array import array
a = array('H', [4000, 10, 700, 22222])
sum(a) # 26932
a[1:3] # array('H', [10, 700])
```

The `collections` module offers `deque`, which functions like a list with faster append/pop operations from ends:

```
from collections import deque
d = deque(['task1', 'task2', 'task3'])
d.append('task4')
print('Handling', d.popleft()) # Handling task1
```

Useful in algorithms such as breadth-first search:

```
unsearched = deque([starting_node])
def breadth_first_search():
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Other list tools include the `bisect` module for maintaining sorted lists and `heapq` for heap-based structures.

11.8. Decimal Floating-Point Arithmetic¶

The `decimal` module offers `Decimal`, a datatype for decimal floating-point arithmetic. It is especially helpful for:

- Financial applications requiring exact decimal representation
- Controlling precision and rounding
- Ensuring compliance with legal or regulatory standards
- Tracking significant decimal places
- Mathematical calculations aligned with manual computation

Example of precise tax calculation:

```
from decimal import *
round(Decimal('0.70') * Decimal('1.05'), 2) # Decimal('0.74')
round(.70 * 1.05, 2)                        # 0.73 (binary floating point)
```

The `Decimal` result maintains trailing zeros and supports exact modulo and equality operations:

```
Decimal('1.00') % Decimal('.10') # Decimal('0.00')
Decimal('1.00') % 0.10           # 0.09999999999999995 (binary floating)
sum([Decimal('0.1')]*10) == Decimal('1.0') # True
0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0 # False
```

Arithmetic with arbitrary precision:

```
getcontext().prec = 36
Decimal(1)/Decimal(7) # Decimal('0.142857142857142857142857142857')
```

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
{'a', 'c', 'd', 'e', 'f', 'g', ...}
```

The `pprint` module offers more sophisticated control over printing both built-in and user-defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structures:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'bl
>>> pprint.pprint(t, width=30)
[[['black', 'cyan'],
   'white',
   ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The `locale` module accesses a database of culture-specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv() # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x),
'$1,234,567.80')
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('$ {village}folk send $10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user-supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{} --&gt; {}'.format(filename, newname))
img_1074.jpg --&gt; Ashley_0.jpg
img_1076.jpg --&gt; Ashley_1.jpg
img_1077.jpg --&gt; Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable-length binary record formats. The following example shows how to loop through header information in a ZIP file without

using the `zipfile` module. Pack codes `"H"` and `"I"` represent two- and four-byte unsigned numbers respectively. The `<` indicates that they are standard size and in little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields
    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)
    start += extra_size + comp_size # skip to the next header
```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high-level `threading` module can run tasks in the background while the main program continues to run:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join() # Wait for the background task to finish
print('Main program waited until background was done.')
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the `threading` module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and

then use the [queue](#) module to feed that thread with requests from other threads. Applications using [Queue](#) objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5. Logging

The [logging](#) module offers a full-featured and flexible logging system.

At its simplest, log messages are sent to a file or to `sys.stderr` :

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG` , `INFO` , `WARNING` , `ERROR` , and `CRITICAL` .

The logging system can be configured directly from Python or loaded from a user-editable configuration file for customized logging without altering the application.

11.6. Weak References

Python does automatic memory management (reference counting for most objects and [garbage collection](#) to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The [weakref](#) module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10) # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a # does not create a reference
>>> del a # remove the one reference
>>> gc.collect() # run garbage collection right away
0
>>> d['primary'] # entry was automatically removed
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
KeyError: 'primary'
```

11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The [array](#) module provides an `array` object that is like a list but stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two-byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The [collections](#) module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth-first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the [bisect](#) module with functions for manipulating sorted lists:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The [heapq](#) module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.8. Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for:

- Financial applications and other uses which require exact decimal representation,
- Control over precision,
- Control over rounding to meet legal or regulatory requirements,
- Tracking of significant decimal places, or
- Applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four-place significance from multiplicands with two-place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == Decimal('1.0')
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```
>>> getcontext().prec = 36
>>> Decimal(1)/Decimal(7)
Decimal('0.142857142857142857142857142857')
```

— End of Content

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
{'a', 'c', 'd', 'e', 'f', 'g', ...}
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the "pretty printer" adds line breaks and indentation to more clearly reveal data structure:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
>>> pprint.pprint(t, width=30)
[[['black', 'cyan'],
  'white',
  ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate"""
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv() # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'], conv['frac_digits'], x),
'$1,234,567.80')
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('$ {village} folk send $10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications with possibly incomplete data, `safe_substitute()` may be more appropriate — it leaves placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=dict(n=i, date=date), f=ext)
...     print('{} --> {}'.format(filename, newname))
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

This templating approach can also be used to separate program logic from multiple output formats, such as XML, plain text reports, or HTML web pages.

11.3. Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. Example:

```
import struct
```

```

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3): # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size # skip to the next header

```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve responsiveness or run I/O in parallel with computations.

Example of running tasks in the background:

```

import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')
background.join() # Wait for the background task to finish
print('Main program waited until background was done.')

```

Coordination of threads sharing data is achieved through synchronization primitives like locks, events, condition variables, and semaphores. Many applications use the [queue](#) module for thread-safe request queues:

```

>>> from collections import deque
>>> unsearched = deque([starting_node])

def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)

```

Other list tools include the [bisect](#) module for manipulating sorted lists:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The [heapq](#) module provides functions for implementing heaps based on regular lists:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.5. Decimal Floating-Point Arithmetic

The [decimal](#) module offers a [Decimal](#) datatype for precise decimal floating-point arithmetic, useful in financial applications, controlling precision and rounding, and other uses requiring exact decimal representation:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

The Decimal result preserves significance and avoids issues with binary floating point:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995
>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

You can set arbitrary precision:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571428571')
```



[Copyright](#) 2001-2025, Python Software Foundation. This page is licensed under the Python Software Foundation License Version 2. Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License. See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation. [Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

12. Virtual Environments and Packages ¶

12.1. Introduction ¶

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a [virtual environment](#), a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

12.2. Creating Virtual Environments ¶

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python -m venv tutorial
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

A common directory location for a virtual environment is `.venv`. This name keeps the directory typically hidden in your shell and thus out of the way while giving it a name that explains why the directory exists. It also prevents clashing with `.env` environment variable definition files that some tooling supports.

Once you've created a virtual environment, you may activate it.

On Windows

```
tutorial-env\Scripts\activate
```

On Unix or MacOS

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the `csh` or `fish` shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36) ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python3.5.zip', ..., '~/envs/tutorial-env/lib/python3.5/site-packa
>>>
```

To deactivate a virtual environment, type:

```
deactivate
```

Into the terminal.

12.3. Managing Packages with pip ¶

You can install, upgrade, and remove packages using a program called `pip`. By default `pip` will install packages from the [Python Package Index](#). You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: “install,” “uninstall,” “freeze,” etc. (Consult the [Installing Python Modules](#) guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

You can also install a specific version of a package by giving the package name followed by `==` and the version number:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, `pip` will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run `python -m pip install --upgrade` to upgrade the package to the latest version:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
  Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
  Successfully installed requests-2.7.0
```

`python -m pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`python -m pip show` will display information about a particular package:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` will produce a similar list of the installed packages, but the output uses the format that `python -m pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The `requirements.txt` can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
Installing collected packages: novas, numpy, requests
Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the [Installing Python Modules](#) guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the [Python packaging user guide](#).

12. Virtual Environments and Packages

12.1. Introduction

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a [virtual environment](#), a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

12.2. Creating Virtual Environments

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python -m venv tutorial
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

A common directory location for a virtual environment is `.venv`. This name keeps the directory typically hidden in your shell and thus out of the way while giving it a name that explains why the directory exists. It also prevents clashing with `.env` environment variable definition files that some tooling supports.

Once you've created a virtual environment, you may activate it.

On Windows:

```
tutorial-env\Scripts\activate
```

On Unix or MacOS:

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the `csh` or `fish` shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36) ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python3.5.zip', ..., '~/envs/tutorial-env/lib/python3.5/site-packa
>>>
```

To deactivate a virtual environment, type:

```
deactivate
```

into the terminal.

12.3. Managing Packages with pip

You can install, upgrade, and remove packages using a program called `pip`. By default, `pip` will install packages from the [Python Package Index](#). You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: “install”, “uninstall”, “freeze”, etc. (Consult the [Installing Python Modules](#) guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

You can also install a specific version of a package by giving the package name followed by `==` and the version number:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, `pip` will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run `python -m pip install --upgrade` to upgrade the package to the latest version:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
Found existing installation: requests 2.6.0
Uninstalling requests-2.6.0:
  Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`python -m pip show` will display information about a particular package:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` will produce a similar list of the installed packages, but the output uses the format that `python -m pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The `requirements.txt` can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
Installing collected packages: novas, numpy, requests
Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the [Installing Python Modules](#) guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the [Python packaging user guide](#).

12. Virtual Environments and Packages

12.1. Introduction

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a **virtual environment**, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

12.2. Creating Virtual Environments

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

A common directory location for a virtual environment is `.venv`. This name keeps the directory typically hidden in your shell and thus out of the way while giving it a name that explains why the directory exists. It also prevents clashing with `.env` environment variable definition files that some tooling supports.

Once you've created a virtual environment, you may activate it.

- On Windows, run:

```
tutorial-env\Scripts\activate
```

- On Unix or MacOS, run:

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the `csh` or `fish` shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36) ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python3.5.zip', ..., '~/envs/tutorial-env/lib/python3.5/site-packa
>>>
```

To deactivate a virtual environment, type:

```
deactivate
```

into the terminal.

12.3. Managing Packages with `pip`

You can install, upgrade, and remove packages using a program called `pip`. By default `pip` will install packages from the [Python Package Index](#). You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: “install”, “uninstall”, “freeze”, etc. (Consult the [Installing Python Modules](#) guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

You can also install a specific version of a package by giving the package name followed by `==` and the version number:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, `pip` will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run `python -m pip install --upgrade` to upgrade the package to the latest version:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
  Installing collected packages: requests
    Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`python -m pip show` will display information about a particular package:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` will produce a similar list of the installed packages, but the output uses the format that `python -m pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The `requirements.txt` can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
Installing collected packages: novas, numpy, requests
Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the [Installing Python Modules](#) guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the [Python packaging user guide](#).

12. Virtual Environments and Packages

12.1. Introduction

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a [virtual environment](#), a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

12.2. Creating Virtual Environments

The module used to create and manage virtual environments is called [venv](#). `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

A common directory location for a virtual environment is `.venv`. This name keeps the directory typically hidden in your shell and thus out of the way while giving it a name that explains why the directory exists. It also prevents clashing with `.env` environment variable definition files that some tooling supports.

Once you've created a virtual environment, you may activate it.

On Windows:

```
tutorial-env\Scripts\activate
```

On Unix or MacOS:

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the `csh` or `fish` shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36) ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python3.5.zip', ..., '~/envs/tutorial-env/lib/python3.5/site-packa
>>>
```

To deactivate a virtual environment, type:

```
deactivate
```

into the terminal.

12.3. Managing Packages with pip

You can install, upgrade, and remove packages using a program called **pip**. By default, `pip` will install packages from the [Python Package Index](#). You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: "install", "uninstall", "freeze", etc. (Consult the [Installing Python Modules](#) guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

You can also install a specific version of a package by giving the package name followed by `==` and the version number:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, `pip` will notice the requested version is already installed and do nothing. You can specify a different version number to get that version, or run:

```
python -m pip install --upgrade requests
```

to upgrade the package to the latest version.

The command:

```
python -m pip uninstall <package_name>
```

followed by one or more package names will remove the packages from the virtual environment.

The command:

```
python -m pip show <package_name>
```

will display information about a particular package:

```
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

The command:

```
python -m pip list
```

will display all of the packages installed in the virtual environment:

```
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

The command:

```
python -m pip freeze
```

will produce a similar list of the installed packages, but the output uses the format that `pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
```

and then display it:

```
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The `requirements.txt` can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with:

```
(tutorial-env) $ python -m pip install -r requirements.txt
```

This will install all packages listed in the file.

`pip` has many more options. Consult the [Installing Python Modules](#) guide for complete documentation. When you've written a package and want to make it available on the Python Package Index, see the [Python packaging user guide](#).

13. What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python's documentation set. Some other documents in the set are:

- [The Python Standard Library](#):
You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a *lot* of additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what's available.
- [Installing Python Modules](#):
Explains how to install additional modules written by other Python users.
- [The Python Language Reference](#):
A detailed explanation of Python's syntax and semantics. It's heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- <https://www.python.org>: The major Python web site. It contains code, documentation, and pointers to Python-related pages around the web.
- <https://docs.python.org>: Fast access to Python's documentation.
- <https://pypi.org>: The Python Package Index, previously also nicknamed the Cheese Shop [1], is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- <https://code.activestate.com/recipes/langs/python/>: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled *Python Cookbook* (O'Reilly & Associates, ISBN 0-596-00797-3).
- <https://pyvideo.org>: Collects links to Python-related videos from conferences and user-group meetings.
- <https://scipy.org>: The Scientific Python project includes modules for fast array computations and manipulations plus a host of packages for such things as linear algebra, Fourier transforms, non-linear solvers, random number distributions, statistical analysis, and the like.

For Python-related questions and problem reports, you can post to the newsgroup *comp.lang.python*, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are hundreds of postings a day, asking (and answering) questions, suggesting new features, and announcing new modules. Mailing list archives are available at <https://mail.python.org/pipermail/>.

Before posting, be sure to check the list of [Frequently Asked Questions](#) (also called the FAQ). The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

Footnotes

[1] “Cheese Shop” is a Monty Python's sketch: a customer enters a cheese shop, but whatever cheese he asks for, the clerk says it's missing.

14. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the [GNU Readline](#) library, which supports various styles of editing. This library has its own documentation which we won't duplicate here.

14.1. Tab Completion and History Editing

Completion of variable and module names is [automatically enabled](#) at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `.` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `getattr()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2. Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an `INDENT` token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is [IPython](#), which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is [bpython](#).

Navigation

- [index](#)
 - [modules](#) |
 - [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#) »
 - [The Python Tutorial](#) »
 - [14. . Interactive Input Editing and History Substitution](#)
-

Contents

Previous topic

[13. What Now?](#)

Next topic

[15. Floating-Point Arithmetic: Issues and Limitations](#)

This page

- [Report a bug](#)
 - [Show source](#)
-



[Copyright](#) 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

14. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the [GNU Readline](#) library, which supports various styles of editing. This library has its own documentation which we won't duplicate here.

14.1. Tab Completion and History Editing

Completion of variable and module names is [automatically enabled](#) at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `.` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `getattr()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2. Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an `INDENT` token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is [IPython](#), which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is [bpython](#).

Navigation

- [index](#)
 - [modules](#) |
 - [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#) »
 - [The Python Tutorial](#) »
 - **14. Interactive Input Editing and History Substitution**
-

Table of Contents

- [14. Interactive Input Editing and History Substitution](#)
 - [14.1. Tab Completion and History Editing](#)
 - [14.2. Alternatives to the Interactive Interpreter](#)
-

Previous topic

[13. What's Now?](#)

Next topic

[15. Floating-Point Arithmetic: Issues and Limitations](#)

This page

- [Report a bug](#)
 - [Show source](#)
-

Navigation

- [index](#)
 - [modules](#) |
 - [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#)
 - [The Python Tutorial](#)
 - **[14. Interactive Input Editing and History Substitution](#)**
-

Copyright

© [Python Software Foundation](#) 2001-2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate.](#)

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

14. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the [GNU Readline](#) library, which supports various styles of editing. This library has its own documentation which we won't duplicate here.

14.1. Tab Completion and History Editing

Completion of variable and module names is [automatically enabled](#) at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `.` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `getattr()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2. Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an `INDENT` token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is [IPython](#), which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is [bpython](#).

Navigation

- [index](#)
 - [modules](#) |
 - [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#) »
 - [The Python Tutorial](#) »
 - [14. Interactive Input Editing and History Substitution](#)
-

Table of Contents

14. Interactive Input Editing and History Substitution

14.1. Tab Completion and History Editing

14.2. Alternatives to the Interactive Interpreter

Copyright and License

© [Copyright](#) 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

15. Floating-Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the **decimal** fraction `0.625` has value $6/10 + 2/100 + 5/1000$, and in the same way the **binary** fraction `0.101` has value $1/2 + 0/4 + 1/8$. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction $1/3$. You can approximate that as a base 10 fraction:

`0.3`

or, better,

`0.33`

or, better,

`0.333`

and so on. No matter how many digits you're willing to write down, the result will never be exactly $1/3$, but will be an increasingly better approximation of $1/3$.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, $1/10$ is the infinitely repeating fraction:

`0.0001100110011001100110011001100110011001100110011...`

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of $1/10$, the binary fraction is `3602879701896397 / 2**55`, which is close to but not exactly equal to the true value of $1/10$.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for `0.1`, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead:

```
>>> 1/10
0.1
```

Just remember, even though the printed result looks like the exact value of $1/10$, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers `0.1` and `0.10000000000000001` and

`0.1000000000000000055511151231257827021181583404541015625` are all approximated by `3602879701896397 / 2**55`. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would choose the one with 17 significant digits, `0.10000000000000001`. Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display `0.1`.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> import math
>>> format(math.pi, '.12g') # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f') # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

It's important to realize that this is, in a real sense, an illusion: you're simply rounding the *display* of the true machine value.

One illusion may beget another. For example, since 0.1 is not exactly 1/10, summing three values of 0.1 may not yield exactly 0.3, either:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Also, since the 0.1 cannot get any closer to the exact value of 1/10 and 0.3 cannot get any closer to the exact value of 3/10, then pre-rounding with `round()` cannot help:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Though the numbers cannot be made closer to their intended exact values, the `math.isclose()` function can be useful for comparing inexact values:

```
>>> import math
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the `round()` function can be used to compare rough approximations:

```
>>> round(math.pi, 2) == round(22/7, 2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with “0.1” is explained in precise detail below, in the “Representation Error” section. See [Examples of Floating Point Problems](#) for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see [The Perils of Floating Point](#) for a more complete account of other common surprises.

As that says near the end, “there are no easy answers.” Still, don't be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the

order of no more than 1 part in 2^{53} per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method's format specifiers in [Format String Syntax](#).

For use cases which require exact decimal representation, try using the `decimal` module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the `fractions` module which implements arithmetic based on rational numbers (so the numbers like $1/3$ can be represented exactly).

If you are a heavy user of floating-point operations you should take a look at the NumPy package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```
>>> float.fromhex('0x1.921f9f01b866ep+1') == x
True
```

Since the representation is exact, it is useful for reliably porting values across different versions of Python (platform independence) and exchanging data with other languages that support the same format (such as Java and C99).

Another helpful tool is the `sum()` function which helps mitigate loss-of-precision during summation. It uses extended precision for intermediate rounding steps as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:

```
>>> sum([0.1] * 10) == 1.0
False
>>> sum([Fraction(x) for x in arr]) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                  # Single rounding
8.042173697819788e-13
>>> sum(arr)                        # Multiple roundings in extended precision
8.042178034628478e-13
```

The `math.fsum()` goes further and tracks all of the “lost digits” as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in

uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...        -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr))) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                # Single rounding
8.042173697819788e-13
>>> sum(arr)                      # Multiple roundings in standard precision
-0.0051575902860057365
```

15.1. Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won’t display the exact decimal number you expect.

Why is that? $1/10$ is not exactly representable as a binary fraction. Since at least 2000, almost all machines use IEEE 754 binary floating-point arithmetic, and almost all platforms map Python floats to IEEE 754 binary64 “double precision” values. IEEE 754 binary64 values contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J / 2^{**N}$ where J is an integer containing exactly 53 bits. Rewriting:

$$1 / 10 \sim J / (2^{**N})$$

as

$$J \sim 2^{**N} / 10$$

and recalling that J has exactly 53 bits (is $\geq 2^{**52}$ but $< 2^{**53}$), the best value for N is 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

That is, 56 is the only value for N that leaves J with exactly 53 bits. The best possible value for J is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
>>> q + 1
7205759403792794
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

$$7205759403792794 / 2^{**56}$$

Therefore the best possible approximation to $1/10$ in IEEE 754 double precision is:

$$7205759403792794 / 2^{**56}$$

Dividing both the numerator and denominator by two reduces the fraction to:

$$3602879701896397 / 2^{**55}$$

Note that since we rounded up, this is actually a little bit larger than $1/10$; if we had not rounded up, the quotient would have been a little bit smaller than $1/10$. But in no case can it be *exactly* $1/10$!

So the computer never “sees” $1/10$: what it sees is the exact fraction given above, the best IEEE 754 double approximation it can get:

```
>>> 0.1 * 2**55
3602879701896397.0
```

If we multiply that fraction by 10^{55} , we can see the value out to 55 decimal digits:

```
>>> 3602879701896397 * 10**55 // 2**55  
100000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value `0.1000000000000000055511151231257827021181583404541015625`. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.100000000000000001'
```

The `fractions` and `decimal` modules make these calculations easy:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)
```

```
>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)
```

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

```
>>> format(Decimal.from_float(0.1), '.17')
'0.100000000000000001'
```

Navigation

- [index](#)
- [modules](#) | [next](#)
- [previous](#)
- <https://www.python.org/> »
- [3.13.5 Documentation](#) »
- [The Python Tutorial](#) »
- [15. Floating-Point Arithmetic: Issues and Limitations](#)



Copyright 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD

License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

15. Floating-Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the **decimal** fraction `0.625` has value $6/10 + 2/100 + 5/1000$, and in the same way the **binary** fraction `0.101` has value $1/2 + 0/4 + 1/8$. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction $1/3$. You can approximate that as a base 10 fraction:

`0.3`

or, better,

`0.33`

or, better,

`0.333`

and so on. No matter how many digits you're willing to write down, the result will never be exactly $1/3$, but will be an increasingly better approximation of $1/3$.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value `0.1` cannot be represented exactly as a base 2 fraction. In base 2, $1/10$ is the infinitely repeating fraction:

`0.0001100110011001100110011001100110011001100110011...`

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of $1/10$, the binary fraction is `3602879701896397/2**55` which is close to but not exactly equal to the true value of $1/10$.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for `0.1`, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead:

```
>>> 1/10
0.1
```

Just remember, even though the printed result looks like the exact value of $1/10$, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers `0.1` and `0.10000000000000001` and

`0.1000000000000000055511151231257827021181583404541015625` are all approximated by `3602879701896397/2**55`. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would choose the one with 17 significant digits, `0.10000000000000001`. Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display `0.1`.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> import math
>>> format(math.pi, '.12g') # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f') # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

It's important to realize that this is, in a real sense, an illusion: you're simply rounding the *display* of the true machine value.

One illusion may beget another. For example, since `0.1` is not exactly $1/10$, summing three values of 0.1 may not yield exactly 0.3, either:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Also, since the 0.1 cannot get any closer to the exact value of $1/10$ and 0.3 cannot get any closer to the exact value of $3/10$, then pre-rounding with `round()` function cannot help:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Though the numbers cannot be made closer to their intended exact values, the `math.isclose()` function can be useful for comparing inexact values:

```
>>> import math
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the `round()` function can be used to compare rough approximations:

```
>>> round(math.pi, ndigits=2) == round(22/7, ndigits=2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with “0.1” is explained in precise detail below, in the “Representation Error” section. See [Examples of Floating Point Problems](#) for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see [The Perils of Floating Point](#) for a more complete account of other common surprises.

As that says near the end, “there are no easy answers.” Still, don't be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the

order of no more than 1 part in 2^{53} per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method's format specifiers in [Format String Syntax](#).

For use cases which require exact decimal representation, try using the `decimal` module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the `fractions` module which implements arithmetic based on rational numbers (so the numbers like $1/3$ can be represented exactly).

If you are a heavy user of floating-point operations you should take a look at the NumPy package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```
>>> x == 3537115888337719/1125899906842624
True
```

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```
>>> float.fromhex('0x1.921f9f01b866ep+1')
1.5
```

Since the representation is exact, it is useful for reliably porting values across different versions of Python (platform independence) and exchanging data with other languages that support the same format (such as Java and C99).

Another helpful tool is the `sum()` function which helps mitigate loss-of-precision during summation. It uses extended precision for intermediate rounding steps as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:

```
>>> sum([0.1]*10) == 1.0
False
>>> sum([Fraction(x) for x in [0.1]*10]) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr) # Single rounding
8.042173697819788e-13
>>> sum(arr) # Multiple roundings in extended precision
8.042178034628478e-13
```

The `math.fsum()` goes further and tracks all of the “lost digits” as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in

uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16, ... -1434011
>>> float(sum(map(Fraction, arr))) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr) # Single rounding
8.042173697819788e-13
>>> sum(arr) # Multiple roundings in standard precision
-0.0051575902860057365
```

15.1. Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won’t display the exact decimal number you expect.

Why is that? $1/10$ is not exactly representable as a binary fraction. Since at least 2000, almost all machines use IEEE 754 binary floating-point arithmetic, and almost all platforms map Python floats to IEEE 754 binary64 “double precision” values. IEEE 754 binary64 values contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^{*N}$ where J is an integer containing exactly 53 bits. Rewriting:

$$1/10 \sim J/(2^{*N})$$

as

$$J \sim 2^{*N} / 10$$

and recalling that J has exactly 53 bits (is ≥ 252 but < 253), the best value for N is 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

That is, 56 is the only value for N that leaves J with exactly 53 bits. The best possible value for J is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
>>> q + 1
7205759403792794
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

$$7205759403792794/2^{*56}$$

Therefore, the best possible approximation to $1/10$ in IEEE 754 double precision is:

$$7205759403792794/2^{*56}$$

Dividing both the numerator and denominator by two reduces the fraction to:

$$3602879701896397/2^{*55}$$

Note that since we rounded up, this is actually a little bit larger than $1/10$; if we had not rounded up, the quotient would have been a little bit smaller than $1/10$. But in no case can it be *exactly* $1/10$!

So the computer never “sees” $1/10$: what it sees is the exact fraction given above, the best IEEE 754 double approximation it can get:

```
>>> 0.1 * 2**55
3602879701896397.0
```

If we multiply that fraction by 10^{55} , we can see the value out to 55 decimal digits:

```
>>> 3602879701896397 * 10**55 // 2**55
1000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value

`0.1000000000000000055511151231257827021181583404541015625`. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The [fractions](#) and [decimal](#) modules make these calculations easy:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17f')
'0.10000000000000001'
```

Note: All the other content provided would be part of the same document, but given the length, the above is a condensed, clean, formatted Markdown translation focusing on the main content and structure.

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive [REPL](#). The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with [curses](#) support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see [Controlling color](#) for details. Function keys provide some additional functionality.

F1	enters the interactive help browser pydoc .
F2	allows for browsing command-line history with neither output nor the <code>>>></code> and <code>...</code> prompts.
F3	enters “paste mode”, which makes pasting larger blocks of code easier. Press <code>F3</code> to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the [PYTHON_BASIC_REPL](#) environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually `Control-C` or `Delete`) to the primary or secondary prompt cancels the input and returns to the primary prompt.^[1] Typing an interrupt while a command is executing raises the [KeyboardInterrupt](#) exception, which may be handled by a `try` statement.

16.1.2. Executable Python Scripts

On BSD’ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#!/usr/bin/env python3
```

(assuming that the interpreter is on the user’s `PATH`) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows (`'\r\n'`) line ending. Note that the hash, or pound, character, `'#'`, is used to start a comment in Python.

The script can be given an executable mode, or permission, using the `chmod` command:

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates `.py` files with `python.exe` so that a double-click on a Python file will run it as a script. The extension can also be `.pyw`, in which case, the console window that normally appears is suppressed.

16.1.3. The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4. The Customization Modules

Python provides two hooks to let you customize it: `sitecustomize` and `usercustomize`. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the `-S` command-line option to disable the automatic import.

`sitecustomize` works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before `usercustomize`. See the documentation of the `site` module for more details.

Additional Resources

- [Table of Contents](#)
- [Previous topic: Floating-Point Arithmetic: Issues and Limitations](#)
- [Next topic: Python Setup and Usage](#)
- [Report a bug](#)
- [Show source](#)
- [Python](#)
- [3.13.5 Documentation](#)
- [The Python Tutorial](#)

© [Python Software Foundation](#), 2001-2025.

This page is licensed under the [Python Software Foundation License Version 2](#).

Examples, recipes, and other code in the documentation are additionally licensed under the [Zero Clause BSD License](#).

[See the History and License](#) for more information.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive [REPL](#). The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with [curses](#) support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see [Controlling color](#) for details. Function keys provide some additional functionality:

- **F1** enters the interactive help browser [pydoc](#)
- **F2** allows for browsing command-line history with neither output nor the `>>>` and `...` prompts
- **F3** enters “paste mode”, making pasting larger blocks of code easier

Press **F3** again to return to the regular prompt.

When using the new interactive shell, exit the shell by typing **exit** or **quit** (call parentheses are not required).

This shell can be disabled via the `PYTHON_BASIC_REPL` environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; if input came from a file, it exits with a nonzero status after printing the stack trace. Exceptions handled by an `except` clause are not errors. Fatal errors, such as internal inconsistencies or resource exhaustion, cause a nonzero exit status. Errors are written to standard error; output of commands goes to standard output.

Typing `Control-C` or `Delete` cancels input and returns to the primary prompt. Pressing `Control-C` during execution raises the `KeyboardInterrupt` exception, which can be handled in a `try` statement.

16.1.2. Executable Python Scripts

Unix systems can make Python scripts directly executable:

```
#!/usr/bin/env python3
```

at the top of the script, and give the file executable permission (e.g., `chmod +x myscript.py`). On Windows, `.py` files are associated with `python.exe`, so double-clicking runs the script. Use `.pyw` to suppress the console window.

16.1.3. The Interactive Startup File

Set the `PYTHONSTARTUP` environment variable to a file containing startup commands executed when Python starts interactively. It runs in the same namespace as interactive commands, enabling custom prompts or imports.

You can explicitly read the startup file in a script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as f:
```

```
startup_code = f.read()
exec(startup_code)
```

16.1.4. The Customization Modules

Python supports `sitecustomize` and `usercustomize` modules for configuration. To locate the user site-packages directory:

```
import site
site.getusersitepackages()
```

Create `usercustomize.py` in that directory for user-specific settings. The `sitecustomize` module, usually installed globally, is imported before `usercustomize`.

Footnotes:

- [1] A problem with the GNU Readline package may prevent this.

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive *REPL*. The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with [curses support](#), a new interactive shell is used by default. This shell supports color, multiline editing, history browsing, and paste mode. To disable color, see [Controlling color](#) for details. Function keys provide additional functionality:

- F1 enters the interactive help browser [pydoc](#).
- F2 allows browsing command-line history without output or the `>>>` and `...` prompts.
- F3 enters “paste mode” for easier pasting of larger code blocks; press F3 again to return to the regular prompt.

When using the new interactive shell, exit by typing `exit` or `quit` without parentheses.

To disable the new shell, set the `PYTHON_BASIC_REPL` environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints a message and stack trace. In interactive mode, it returns to the primary prompt; when input was from a file, it exits with a nonzero status after printing the stack trace. Errors caught by an `except` clause are not errors here. Fatal errors include internal inconsistencies and memory exhaustion, causing an exit with a nonzero status. All error messages go to standard error; command output goes to standard output.

Interrupting with Ctrl-C or Delete cancels input and returns to the primary prompt. If the command is executing, it raises a `KeyboardInterrupt` exception that can be handled with a try statement.

16.1.2. Executable Python Scripts

On BSD Unix systems, scripts can be made executable by adding:

```
#!/usr/bin/env python3
```

as the first line, assuming Python is on the system PATH, and making the file executable with `chmod +x`.

On Windows, Python files are associated with `python.exe`, so double-clicking runs the script. `.pyw` scripts suppress the console window.

16.1.3. The Interactive Startup File

Setting the `PYTHONSTARTUP` environment variable to a filename enables executing commands on startup (similar to `.profile` in shells). This file is read only in interactive sessions, not when running scripts or with `/dev/tty`. It affects the interactive namespace directly and can modify prompts via `sys.ps1` and `sys.ps2`.

To load an additional startup file from the current directory:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as f:
```

```
startup_file = f.read()
exec(startup_file)
```

16.1.4. The Customization Modules

Python allows customizing with `sitecustomize` and `usercustomize` modules. To find your user `site-packages` directory, run:

```
import site
site.getusersitepackages()
```

Create `usercustomize.py` in that directory to affect all Python invocations (unless disabled with `-s`). `sitecustomize` is typically placed by administrators in the global site-packages directory, affecting all users.

Footnote:

[1] A problem with GNU Readline may prevent this.

Navigation

- [Index](#)
- [Modules](#)
- [Next](#)
- [Previous](#)
- [Python »](#)

This page

- [Report a bug](#)
- [Show source](#)

3.13.5 Documentation

This page

- [The Python Tutorial](#)
- [16. Appendix](#)

Licensing and acknowledgments

© [Python Software Foundation](#), 2001–2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples and other code are also licensed under the Zero Clause BSD License.

Further details: [History and License](#)

The Python Software Foundation is a non-profit. [Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Report a bug](#)

Created using [Sphinx](#).

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive [REPL](#). The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with [curses](#) support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see [Controlling color](#) for details. Function keys provide some additional functionality:

- **F1**: enters the interactive help browser [pydoc](#)
- **F2**: allows for browsing command-line history with neither output nor the `>>>` and `...` prompts
- **F3**: enters “paste mode”, which makes pasting larger blocks of code easier; press F3 to return to the regular prompt

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the [PYTHON_BASIC_REPL](#) environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero status after printing the stack trace. Exceptions handled by an `except` clause in a `try` statement are not errors in this context. Some errors are unconditionally fatal and cause an exit with a nonzero status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to standard error; normal output from executed commands is written to standard output.

Typing the interrupt character (usually **Control-C** or **Delete**) to the primary or secondary prompt cancels the input and returns to the primary prompt [1]. Typing an interrupt while a command is executing raises the [KeyboardInterrupt](#), which may be handled by a `try` statement.

16.1.2. Executable Python Scripts

On BSD-style Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line:

```
#!/usr/bin/env python3
```

(assuming the interpreter is on the user's `PATH`) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows (`'\r\n'`) line ending. Note that the hash character `#` is used to start a comment in Python.

The script can then be given an executable mode using the `chmod` command:

```
$ chmod +x myscript.py
```

On Windows, there is no concept of an “executable mode”. The Python installer automatically associates `.py` files with `python.exe`, so double-clicking a Python file runs it as a script. The extension can also be `.pyw`, in which case the console window is suppressed.

16.1.3. The Interactive Startup File

When using Python interactively, it can be handy to have some standard commands executed every time the interpreter starts. This can be done by setting an environment variable `PYTHONSTARTUP` to the filename of a start-up commands file. This is similar to the `.profile` feature of Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands. It is executed in the same namespace as interactive commands, so objects defined or imported there can be used directly. You can also change prompts `sys.ps1` and `sys.ps2` in this file.

To read an additional start-up file from the current directory, include code like:

```
if os.path.isfile('.pythonrc.py'):
    exec(open('.pythonrc.py').read())
```

in the global start-up file. To do this explicitly in a script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4. The Customization Modules

Python provides hooks `sitecustomize` and `usercustomize` for customization. To use them, find your user site-packages directory with the following code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Create a file named `usercustomize.py` in that directory with whatever you'd like; it affects every Python run unless started with the `-s` option to disable automatic import.

`sitecustomize` works similarly but is usually created by an administrator in the global site-packages directory and imported before `usercustomize`. See the [site](#) module documentation for details.

[1] A problem with the GNU Readline package may prevent this.

Navigation

- [index](#)
- [modules](#)
- [next](#)
- [previous](#)
- [Python »](#)
- [3.13.5 Documentation](#)
- [The Python Tutorial »](#)
- [16. Appendix](#)

Copyright and Licensing

© [Python Software Foundation](#) 2001-2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

For more information, see [History and License](#).

The Python Software Foundation is a non-profit corporation. [Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive [REPL](#). The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with [curses](#) support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see [Controlling color](#) for details. Function keys provide some additional functionality:

- **F1** enters the interactive help browser [pydoc](#).
- **F2** allows for browsing command-line history with neither output nor the `>>>` and `...` prompts.
- **F3** enters “paste mode”, which makes pasting larger blocks of code easier. Press **F3** to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the [PYTHON_BASIC_REPL](#) environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually **Control-C** or **Delete**) to the primary or secondary prompt cancels the input and returns to the primary prompt. [1] Typing an interrupt while a command is executing raises the [KeyboardInterrupt](#) exception, which may be handled by a `try` statement.

16.1.2. Executable Python Scripts

On BSD-ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line:

```
#!/usr/bin/env python3
```

(assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows (`'\r\n'`) line ending. Note that the hash or pound character, `'#'`, is used to start a comment in Python.

The script can be given an executable mode, or permission, using the `chmod` command:

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates `.py` files with `python.exe`, so double-clicking a Python file will run it as a script. The extension can also be `.pyw`, in which case, the console window that normally appears is suppressed.

16.1.3. The Interactive Startup File

When you use Python interactively, it is often handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your start-up commands. This is similar to the `.profile` feature of Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4. The Customization Modules

Python provides two hooks to let you customize it: `sitecustomize` and `usercustomize`. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python unless it is started with the `-s` command-line option to disable automatic import.

`sitecustomize` works similarly but is typically created by an administrator of the computer in the global site-packages directory. It is imported before `usercustomize`. See the [site](#) module documentation for more details.

Additional links

- [Previous topic](#): 15. Floating-Point Arithmetic: Issues and Limitations
- [Next topic](#): Python Setup and Usage

This page

- [Report a bug](#)
- [Show source](#)

Navigation

- [index](#)
- [modules](#) |

- [next](#) |
 - [previous](#) |
 - [Python](#) »
 - [3.13.5 Documentation](#)
 - [The Python Tutorial](#) »
 - [16. Appendix](#)
-

Footer

© [Python Software Foundation](#), 2001-2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation. [Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive [REPL](#). The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with [curses](#) support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see [Controlling color](#) for details. Function keys provide some additional functionality:

- **F1** enters the interactive help browser [pydoc](#)
- **F2** allows for browsing command-line history with neither output nor the `>>>` and `...` prompts
- **F3** enters “paste mode”, which makes pasting larger blocks of code easier. Press **F3** to return to the regular prompt

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the [PYTHON_BASIC_REPL](#) environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an `except` clause in a `try` statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero status, including internal inconsistencies and cases of running out of memory. All error messages are written to standard error; normal output from commands is written to standard output.

Typing the interrupt character (usually **Control-C** or **Delete**) to the primary or secondary prompt cancels the input and returns to the primary prompt. [1] Typing an interrupt while a command is executing raises the [KeyboardInterrupt](#) exception, which may be handled by a `try` statement.

16.1.2. Executable Python Scripts

On BSD-like Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line:

```
#!/usr/bin/env python3
```

(assuming that the interpreter is on the user’s `PATH`) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`\n`), not a Windows (`\r\n`) line ending. Note that the hash or pound character (`#`) is used to start a comment in Python.

The script can be given an executable mode, or permission, using the `chmod` command:

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates `.py` files with `python.exe` so that a double-click on a Python file will run it as a script. The extension can also be `.pyw`, in which case, the console window that normally appears is suppressed.

16.1.3. The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named `PYTHONSTARTUP` to the name of a file containing your startup commands. This is similar to the `.profile` feature of Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional startup file from the current directory, you can program this in the global startup file:

```
if os.path.isfile('.pythonrc.py'):
    exec(open('.pythonrc.py').read())
```

If you want to use the startup file in a script, you must do this explicitly:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4. The Customization Modules

Python provides two hooks to let you customize it: `sitecustomize` and `usercustomize`. To see how it works, you need to find the location of your user site-packages directory. Start Python and run:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the `-s` command line option to disable automatic import.

`sitecustomize` works similarly, but is typically created by an administrator of the computer in the global site-packages directory and is imported before `usercustomize`. See the [site](#) module documentation for more details.

Footnotes

[1]: A problem with the GNU Readline package may prevent this.

Other Navigation and Legal Information

- [Table of Contents](#)
- [Report a bug](#)
- [Show source](#)
- [Python](#)
- [3.13.5 Documentation](#)
- [The Python Tutorial](#)

- [The Python Software Foundation Website](#)

Copyright

[Python Software Foundation](#) 2001-2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

3. An Informal Introduction to Python¶

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the “Copy” button (it appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

3.1. Using Python as a Calculator¶

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn’t take long.)

3.1.1. Numbers¶

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` can be used to perform arithmetic; parentheses `( )` can be used for grouping.

For example:

```
>>> 2 + 2
4
```

```
>>> 50 - 5 * 6
20
```

```
>>> (50 - 5 * 6) / 4
5.0
```

```
>>> 8 / 5 # division always returns a floating-point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division `( / )` always returns a float. To do [floor division](#) and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
```

```
>>> 17 // 3 # floor division discards the fractional part
5
```

```
>>> 17 % 3 # the % operator returns the remainder of the division
2
```

```
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17
```

With Python, it is possible to use the `**` operator to calculate powers [1]:

```
>>> 5 ** 2 # 5 squared
25
```

```
>>> 2 ** 7 # 2 to the power of 7
128
```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
```

```
>>> price + _
113.0625
```

```
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for `complex` numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters `!`, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2].

For example:

```
>>> 'spam eggs' # single quotes
'spam eggs'

>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'

>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use ' to escape the single quote...
'doesn\'t'

>>> "doesn't" # ...or use double quotes instead
'doesn't'

>>> '"Yes," they said.'
'"Yes," they said.'

>>> "\"Yes,\" they said."
"\"Yes,\" they said."

>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new line
First line.
Second line.
```

If you don’t want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print('C:\\some\\name') # here \n means newline!
C:\some\name

>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see [the FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it’s possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
... -h Display this usage message
... -H hostname Hostname to connect to
... """)
Usage: thingy [OPTIONS] -h Display this usage message -H hostname Hostname to
connect to`
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> 'un' * 3 + 'ium'
'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
```

SyntaxError

```
>>> ('un' * 3) 'ium' # same here
```

SyntaxError

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Note that since `-0` is the same as `0`, negative indices start from `-1`.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+
0 1 2 3 4 5 6
-6 -5 -4 -3 -2 -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42] # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'

>>> word[42:]
''
```

Python strings cannot be changed — they are [immutable](#). Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
>>> word[2:] = 'py' # same here
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
```

```
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

- [Strings are examples of sequence types, and support the common operations supported by such types.](#)
- [Strings support a large number of methods for basic transformations and searching.](#)
- [String literals that have embedded expressions \(“f-strings”\).](#)
- [Information about string formatting with `str.format\(\)`.](#)
- [The old formatting operations invoked when strings are the left operand of the `%` operator.] ([../library/stdtypes.html#old-string-formatting](http://library/stdtypes.html#old-string-formatting))

3.1.3. Lists¶

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

For example:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in [sequence](#) types), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are [immutable](#), lists are a [mutable](#) type, i.e., it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` *method* (we will see more about methods later):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
['Red', 'Green', 'Blue', 'Alpha']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a [shallow copy](#) of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
['Red', 'Green', 'Blue', 'Alpha']
>>> rgba
['Red', 'Green', 'Blue', 'Alpha']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the “Copy” button (it appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1 # and this is the second comment
# ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` can be used to perform arithmetic; parentheses `( )` can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # division always returns a floating-point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do [floor division](#) and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>> 17 // 3 # floor division discards the fractional part
```



```

5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17

```

With Python, it is possible to use the `**` operator to calculate powers [1]:

```

>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128

```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```

>>> width = 20
>>> height = 5 * 9
>>> width * height
900

```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```

>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined

```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```

>>> 4 * 3.75 - 1
14.0

```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```

>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06

```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for `complex numbers`, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2].

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
'doesn\'t'
>>> "doesn't" # ...or use double quotes instead
'doesn\'t'
>>> '"Yes," they said.' # quote inside double quotes
'"Yes," they said.'
>>> '"Yes," they said.' # quote inside single quotes
'"Yes," they said.'
>>> '"Isn\'t," they said.' # quote inside string with escaped quote
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new l
First line.
Second line.
```

If you don’t want characters prefaced by `\` to be interpreted as special characters, you can use [raw strings](#) by adding an `r` before the first quote:

```
>>> print('C:\\some\\name') # here \n means newline!
C:\some\name
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see [the FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it’s possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                Display this usage message
...     -H hostname       Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix + 'thon' # can concatenate a variable and a string literal
'Python'
>>> ('un' * 3) + 'ium' # cannot concatenate variable and literal directly
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Note that since `-0` is the same as `0`, negative indices start from `-1`.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are `immutable`. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

- [Text Sequence Type — str](#)
- [String Methods](#)
- [f-strings](#)
- [Format String Syntax](#)
- [printf-style String Formatting](#)

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* types), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]
# slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are `immutable`, lists are a `mutable` type, i.e., it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` method (we will see more about methods later):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
["Red", "Green", "Blue", "Alpha"]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a *shallow copy* of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
>>> rgba
["Red", "Green", "Blue", "Alpha"]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
```

```
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...:`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the “Copy” button (it appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1 # and this is the second comment
# ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` can be used to perform arithmetic; parentheses `( )` can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # division always returns a floating-point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do floor division and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>> 17 // 3 # floor division discards the fractional part
```



```

5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17

```

With Python, it is possible to use the `**` operator to calculate powers [1]:

```

>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128

```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```

>>> width = 20
>>> height = 5 * 9
>>> width * height
900

```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```

>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined

```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```

>>> 4 * 3.75 - 1
14.0

```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```

>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06

```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for complex numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2].

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
'doesn\'t'
>>> "doesn't" # ...or use double quotes instead
'doesn\'t'
>>> '"Yes," they said.' # double quotes inside single quotes
'"Yes," they said.'
>>> "\"Yes,\" they said." # escape double quotes inside double quotes
'"Yes," they said.'
>>> '"Isn\'t," they said.' # escape single quote inside double quotes
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new l
First line.
Second line.
```

If you don’t want characters prefaced by `\` to be interpreted as special characters, you can use raw strings by adding an `r` before the first quote:

```
>>> print('C:\\some\\name') # here \n means newline!
C:\some\name
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see the [FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it’s possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                Display this usage message
...     -H hostname       Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more string literals (i.e. the ones enclosed between quotes) next to each other are automatically concatenated:

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
File "<stdin>", line 1
    prefix 'thon'
    ^^^^^^^
SyntaxError: invalid syntax
```

```
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
    ^^^^^^^
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be **indexed** (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Note that since `-0` is the same as `0`, negative indices start from `-1`.

In addition to indexing, **slicing** is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced:

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are **immutable**. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

- [Text Sequence Type — str](#): Strings are examples of sequence types, and support the common operations supported by such types.
- [String Methods](#): Strings support a large number of methods for basic transformations and searching.
- [f-strings](#): String literals that have embedded expressions.
- [Format String Syntax](#): Information about string formatting with `str.format()`.
- [Printf-style String Formatting](#): The old formatting operations invoked when strings are the left operand of the `%` operator are described in more detail here.

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]
# slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are **immutable**, lists are a **mutable** type, i.e., it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` method:

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
['Red', 'Green', 'Blue', 'Alpha']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a *shallow copy* of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
['Red', 'Green', 'Blue', 'Alpha']
>>> rgba
['Red', 'Green', 'Blue', 'Alpha']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts [`>>>`] and [`...`]: to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the “Copy” button (it appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1 # and this is the second comment
# ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` can be used to perform arithmetic; parentheses `( )` can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # division always returns a floating-point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do [floor division](#) and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>> 17 // 3 # floor division discards the fractional part
```



```

5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17

```

With Python, it is possible to use the `**` operator to calculate powers [1]:

```

>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128

```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```

>>> width = 20
>>> height = 5 * 9
>>> width * height
900

```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```

>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined

```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```

>>> 4 * 3.75 - 1
14.0

```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```

>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06

```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for [complex numbers](#), and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2].

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
'doesn\'t'
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.' # double quotes within single quotes
'"Yes," they said.'
>>> '"Yes," they said.' # escape the double quotes
'"Yes," they said.'
>>> '"Isn\'t," they said.' # escape the single quote inside double quotes
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new l
First line.
Second line.
```

If you don’t want characters prefaced by `\` to be interpreted as special characters, you can use [raw strings](#) by adding an `r` before the first quote:

```
>>> print('C:\\some\\name') # here \n means newline!
C:\some\name
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see [the FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it’s possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                    Display this usage message
...     -H hostname          Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                    Display this usage message
    -H hostname          Hostname to connect to
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more string literals (i.e., the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix + 'thon' # concatenating a variable and a string literal
'Python'
>>> prefix='Py'
>>> ( 'un' * 3) + 'ium' # concatenating expressions is valid
'unununium'
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be `indexed` (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, `slicing` is also supported. While indexing is used to obtain individual characters, `slicing` allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are [immutable](#). Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

- [Text Sequence Type — str](#)
- [String Methods](#)
- [f-strings](#)
- [Format String Syntax](#)
- [Printf-style String Formatting](#)

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in [sequence types](#)), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]
# slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are [immutable](#), lists are a [mutable](#) type, i.e., it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` method (we will see more about methods later):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
['Red', 'Green', 'Blue', 'Alpha']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a [shallow copy](#) of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
['Red', 'Green', 'Blue', 'Alpha']
>>> rgba
['Red', 'Green', 'Blue', 'Alpha']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
```

```
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.1. Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` can be used to perform arithmetic; parentheses `( )` can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # division always returns a floating-point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do *floor division* and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17
```

With Python, it is possible to use the `**` operator to calculate powers [1]:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
```



```
File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for *complex numbers*, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2].

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
'doesn't'
>>> "doesn't" # ...or use double quotes instead
'doesn't'
>>> '"Yes," they said.' # quote inside the string
'"Yes," they said.'
>>> "\\Yes,\" they said." # or with escape characters
'"Yes," they said.'
>>> '"Isn\'t," they said.' # mixing quotes
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new l
First line.
Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print('C:\some\name') # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see [the FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                Display this usage message
...     -H hostname       Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated:

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...        'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix + 'thon' # concatenating variable and string literal
'Python'
>>> ('un' * 3) + 'ium' # concatenating expression and string literal
'unununium'
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'p'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6] # first character
'p'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are *immutable*. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

- [Text Sequence Type — str](#): Strings are examples of *sequence types*, and support the common operations supported by such types.
- [String Methods](#): Strings support a large number of methods for basic transformations and searching.
- [f-strings](#): String literals that have embedded expressions.
- [Format String Syntax](#): Information about string formatting with `str.format()`.
- [printf-style String Formatting](#): The old formatting operations invoked when strings are the left operand of the `%` operator are described in more detail here.

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* types), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]
`[9, 16, 25]`
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are *immutable*, lists are a *mutable* type, i.e., it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` method:

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append("Alpha")
>>> rgb
['Red', 'Green', 'Blue', 'Alpha']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a *shallow copy* of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
['Red', 'Green', 'Blue', 'Alpha']
```

```
>>> rgba
['Red', 'Green', 'Blue', 'Alpha']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the “Copy” button (it appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

3.1. Using Python as a Calculator

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn’t take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*`, and `/` can be used to perform arithmetic; parentheses `( )` can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0
>>> 8 / 5 # division always returns a floating-point number
1.6
```

The integer numbers (e.g., `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g., `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do [floor division](#) and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # floored quotient * divisor + remainder
17
```

With Python, it is possible to use the `**` operator to calculate powers ¹:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for [complex numbers](#), and uses the `j` or `J` suffix to indicate the imaginary part (e.g., `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type `str`, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :). They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result ².

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!" # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975' # digits and numerals enclosed in quotes are also strings
'1975'
```


To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t' # use \' to escape the single quote...
'doesn\'t'
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.' # strings can contain quotes if they are properly escaped
'"Yes," they said.'
>>> '"Yes," they said.' # string with embedded double quote
'"Yes," they said.'
>>> '"Isn\'t,' they said.'" # embedded single quote with escape
'"Isn\'t,' they said.'"

```

In the Python shell, the string definition and output string can look different. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s) # with print(), special characters are interpreted, so \n produces new l
First line.
Second line.

```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use `raw strings` by adding an `r` before the first quote:

```
>>> print('C:\\some\\name') # here \n means newline!
C:\some\name
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name

```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see [the FAQ entry](#) for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...     -h                Display this usage message
...     -H hostname       Hostname to connect to
... """)
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to

```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununium'

```

Two or more string literals (i.e., the ones enclosed between quotes) next to each other are automatically concatenated:

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix + 'thon' # concatenate variable and string literal
'Python'
>>> ('un' * 3) + 'ium' # concatenation of expression and literal
'unununium'
>>> prefix 'thon' # syntax error
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be [indexed](#) (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6] # first character
'P'
```

Note that since `-0` is the same as `0`, negative indices start from `-1`.

In addition to indexing, [slicing](#) is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2] # character from the beginning to position 2 (excluded)
'Py'
```

```
>>> word[4:] # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
|+-----+-----+-----+||P|y|t|h|o|n||+-----+-----+-----+||0|1|2|3|4|5|6||-6|-5|-4|-3|-2|-1||
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from `i` to `j` consists of all characters between the edges labeled `i` and `j`, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42] # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indices are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are `immutable`. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

- [Text Sequence Type — str](#): Strings are examples of sequence types and support the common operations supported by such types.
 - [String Methods](#): Strings support a large number of methods for basic transformations and searching.
 - [f-strings](#): String literals that have embedded expressions.
 - [Format String Syntax](#): Information about string formatting with `str.format()`.
 - [Printf-style String Formatting](#): The old formatting operations invoked when strings are the left operand of the `%` operator are described in more detail here.
-

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in [sequence types](#)), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1] # last item
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are [immutable](#), lists are a [mutable](#) type, i.e., it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `list.append()` method (we will see more about methods later):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that

refer to it:

```
>>> rgb = ['Red', 'Green', 'Blue']
>>> rgba = rgb
>>> id(rgb) == id(rgba) # they reference the same object
True
>>> rgba.append('Alpha')
>>> rgb
['Red', 'Green', 'Blue', 'Alpha']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a `shallow copy` of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
['Red', 'Green', 'Blue', 'Alpha']
>>> rgba
['Red', 'Green', 'Blue', 'Alpha']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for 'else if', and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [match Statements](#).

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers, Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
```

```
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. `break` and `continue` Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. `else` Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes. In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred. In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
```



```

...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3

```

4.6. `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```

>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...

```

This is commonly used for creating minimal classes:

```

>>> class MyEmptyClass:
...     pass
...

```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```

>>> def initlog(*args):
...     pass # Remember to implement this!
...

```

4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript, but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```

def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"

```

```

    case _:
        return "Something's wrong with the internet"

```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```

case 401 | 403 | 404:
    return "Not allowed"

```

Patterns can look like unpacking assignments, and can be used to bind variables. For example:

```

# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")

```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it’s set to (“x”, “y”), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (`foo.bar`), attribute names (`x=` and `y=` above) or class names (recognized by the “...” next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 0), Point(1, 1)]
```

```
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don’t match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
    ...
```

This will capture the second element of the input as `p2` (as long as the input is a sequence of two points)

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variables:

```
from enum import Enum

class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named

in a `global` statement, or, for variables of enclosing functions, named in a `nonlocal` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by object reference* — the value of the object, not the variable name itself, is passed. When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 0x7f7e2b9a4d60>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:

```
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see [Classes](#)). The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument:

```
ask_ok('Do you really want to quit?')
```

- giving one of the optional arguments:

```
ask_ok('OK to overwrite the file?', 2)
```

- or even giving all arguments:

```
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that:

```
>>> i = 5
>>> def f(arg=i): print(arg)
>>> i = 6
>>> f()
5
```

will print `5`.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L
```

This will print:

```
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments

Functions can also be called using *keyword arguments* of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()
# required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g., `actor` is not a valid argument for `parrot`), and their order is not important. This also includes non-optional arguments (e.g., `parrot(voltage=1000)` is valid too). No argument may receive a value more than once.

Here's an example that fails due to this restriction:

```
def function(a):
    pass

function(0, a=0)
# TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name`, which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
```

```

print("-- I'm sorry, we're all out of", kind)
for arg in arguments:
    print(arg)
print("-" * 40)
for kw in keywords:
    print(kw, ":", keywords[kw])

```

It could be called like this:

```

cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client='John Cleese',
           sketch='Cheese Shop Sketch')

```

and of course it would print:

```

-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

```

(Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.)

4.9.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    # Positional-only: before /
    # Positional or keyword: after /, before *
    # Keyword-only: after *

```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also called named parameters.

4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.9.3.2. Positional-Only Parameters

Looking at this in more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are

placed before a `/`. The `/` is used to separate these from the rest of the parameters. If there is no `/` in the function definition, there are no positional-only parameters.

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

4.9.3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, indicating they must be passed by keyword argument, place a `*` in the argument list just before the first *keyword-only* parameter.

4.9.3.4. Function Examples

Consider the following function definitions emphasizing the `/` and `*` markers:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

The first function, `standard_arg`, has no restrictions, arguments can be passed by position or keyword. For example:

```
>>> standard_arg(2)
2
>>> standard_arg(arg=2)
2
```

The second function, `pos_only_arg`, only allows positional arguments because of `/`:

```
>>> pos_only_arg(1)
1
>>> pos_only_arg(arg=1)
# Error: TypeError: pos_only_arg() got some positional-only arguments passed as keyword
```

The third function, `kwd_only_arg`, requires arguments to be passed by keyword:

```
>>> kwd_only_arg(3)
# Error: TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given
>>> kwd_only_arg(arg=3)
3
```

The last function combines all three conventions; it accepts positional-only, positional-or-keyword, and keyword-only arguments:

```
>>> combined_example(1, 2, 3)
# Error: TypeError: combined_example() takes 2 positional arguments but 3 were given
>>> combined_example(1, 2, kwd_only=3)
1 2 3
>>> combined_example(pos_only=1, standard=2, kwd_only=3)
# Error: TypeError: combined_example() got some positional-only arguments passed as keyword
```

Finally, consider this function with a potential collision between a positional argument `name` and `**kwds`, which has `name` as a key:

```
def foo(name, **kwds):  
    return 'name' in kwds
```

There is no possible call that will make it return `True` as the keyword `name` will always bind to the first parameter:

```
>>> foo(1, **{'name': 2})  
# False, because positional argument `name` is assigned before keyword `name`
```

But using `/` (positional-only arguments), it is possible since it allows `name` as a positional argument and `'name'` as a key in the keyword arguments:

```
>>> def foo(name, /, **kwds):  
...     return 'name' in kwds  
>>> foo(1, **{'name': 2})  
True
```

In other words, the names of positional-only parameters can be used in `**kwds` without ambiguity.

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An [if ... elif ... elif ...](#) sequence is a substitute for the [switch](#) or [case](#) statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [Match Statements](#).

4.2. for Statements

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s [for](#) statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
...
# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```

```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
...
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` ([./library/stdtypes.html#range](http://library/stdtypes.html#range)) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways, the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4))
6 # 0 + 1 + 2 + 3
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. break and continue Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. else Clauses on Loops

In a `for` or `while` loop, the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred. In a `while` loop, it's executed after the loop's condition becomes false. In either case, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following [for](#) loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.6. `pass` Statements

The [pass](#) statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place [pass](#) can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The [pass](#) is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

4.7. `match` Statements

A [match](#) statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java, or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed, and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
```

```

case 404:
    return "Not found"
case 418:
    return "I'm a teapot"
case _:
    return "Something's wrong with the internet"

```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```

case 401 | 403 | 404:
    return "Not allowed"

```

Patterns can look like unpacking assignments, and can be used to bind variables:

```

# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")

```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g., dataclasses). You can also define a specific position for attributes in patterns by setting the

`__match_args__` special attribute in your classes. If it's set to ("x", "y"), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```


Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

[Link to the Python documentation on control flow tools](#)

4. More Control Flow Tools¶

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements¶

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword `elif` is short for `else if`, and is useful to avoid excessive indentation. An `if ... elif ... <else>` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [match Statements](#).

4.2. for Statements¶

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```

```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The `range()` Function ¶

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
...
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the

list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](#) statement is such a construct, while an example of a function that takes an iterable is [sum\(\)](#):

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about [list\(\)](#).

4.4. break and continue Statements¶

The [break](#) statement breaks out of the innermost enclosing [for](#) or [while](#) loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n // x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The [continue](#) statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. else Clauses on Loops¶

In a [for](#) or [while](#) loop the [break](#) statement may be paired with an [else](#) clause. If the loop finishes without executing the [break](#), the [else](#) clause executes. For example:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n // x)
...             break
...     else:
...         # loop fell through without finding a factor
```

```

...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3

```

This can be used in algorithms such as searching for prime numbers.

4.6. pass Statements¶

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action:

```

>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...

```

This is commonly used for creating minimal classes:

```

>>> class MyEmptyClass:
...     pass
...

```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```

>>> def initlog(*args):
...     pass # Remember to implement this!
...

```

4.7. match Statements¶

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```

def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"

```

Note the last block: the “variable name” `_` acts as a [wildcard](#) and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

You can use positional parameters with some built-in classes that provide an ordering for their attributes (e.g., dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `("x", "y")`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what.

Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the “(…)” next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 2), Point(3, 4)]
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don’t match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
    ...
```

will capture the second element of the input as `p2` (as long as the input is a sequence of two points)

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.

- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variables:

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#), which is written in a tutorial format.

4.8. Defining Functions¶

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a `global` statement, or, for variables of enclosing functions, named in a `nonlocal` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by object reference* (where the *value* is always an object *reference*, not the value of the object). When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `[None]` ([../glossary.html#term-None](#)). Falling off the end of a function also returns `[None]` ([../glossary.html#term-None](#)).
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using [classes](#), see [Classes](#)). The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9. More on Defining Functions¶

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values¶

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f() # prints 5
```

The above will print `5`.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L
```

```
print(f(1))
print(f(2))
print(f(3))
```

This will print:

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments ¶

Functions can also be called using [keyword arguments](#) of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments, different order
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot() # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a [tuple](#) containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
```

```

    print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])

```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.", "It's really very, VERY runny, sir.",
```

and of course it would print:

```

-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.9.3. Special parameters¶

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    # pos1 and pos2 are positional-only
    # pos_or_kwd can be positional or keyword
    # kwd1 and kwd2 are keyword-only

```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

4.9.3.1. Positional-or-Keyword Arguments¶

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.9.3.2. Positional-Only Parameters¶

Looking at this in a bit more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a `/` in the function definition. The `/` is used to logically separate the positional-only parameters from the rest.

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

4.9.3.3. Keyword-Only Arguments¶

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an `*` in the arguments list just before the first *keyword-only* parameter.

4.9.3.4. Function Examples¶

Consider the following example function definitions paying close attention to the markers `/` and `*`:

```
>>> def standard_arg(arg):
...     print(arg)
>>> def pos_only_arg(arg, /):
...     print(arg)
>>> def kwd_only_arg(*, arg):
...     print(arg)
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

The first function `standard_arg` places no restrictions on the calling convention.

The second function `pos_only_arg` is restricted to only use positional parameters as there is a `/` in the function definition.

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition.

The last example `combined_example` uses all three calling conventions.

Call examples:

```
standard_arg(2)
pos_only_arg(1)
kwd_only_arg(arg=3)
combined_example(1, 2, kwd_only=3)
```

And invalid calls:

```
pos_only_arg(arg=1) # error, arg is positional-only
kwd_only_arg(3)     # error, arg must be keyword
combined_example(1, 2, 3) # error, missing keyword for third argument
```

4.9.4. Arbitrary Argument Lists¶

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these *variadic* arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

Call examples:

```
def concat(*args, sep="/"):
    return sep.join(args)
```

```
concat("earth", "mars", "venus") # 'earth/mars/venus'
concat("earth", "mars", "venus", sep=".")
```

4.9.5. Unpacking Argument Lists¶

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...
>>> d = {'voltage': 'four million', 'state': 'bleedin\ demised', 'action': 'VOOM'}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' de
```

4.9.6. Lambda Expressions¶

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntax sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument. For instance, `list.sort()` takes a sorting key function *key* which can be a lambda function:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Now pairs is sorted by the second element of each tuple
```

4.9.7. Documentation Strings¶

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens

to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything."""
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.
No, really, it doesn't do anything.
```

4.9.8. Function Annotations¶

[Function annotations](#) are completely optional metadata information about the types used by user-defined functions (see [PEP 3107](#) and [PEP 484](#) for more information).

[Annotations](#) are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the [def](#) statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10. Intermezzo: Coding Style¶

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about [coding style](#). Most languages can be written (or more concise, [formatted](#)) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, [PEP 8](#) has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use [UpperCamelCase](#) for classes and [lowercase_with_underscores](#) for functions and methods. Always use `self` as the name for the first method argument.
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only a slight chance people speaking a different language will read or maintain the code.

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements ¶

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An [if ... elif ... elif ...](#) sequence is a substitute for the [switch](#) or [case](#) statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [match Statements](#).

4.2. for Statements ¶

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s [for](#) statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
```

```

del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The range() Function ¶

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
...
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](#) statement is such a construct, while an example of a function that takes an iterable is [sum\(\)](#):

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about [list\(\)](#).

4.4. break and continue Statements ¶

The [break](#) statement breaks out of the innermost enclosing [for](#) or [while](#) loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The [continue](#) statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. else Clauses on Loops ¶

In a [for](#) or [while](#) loop the [break](#) statement may be paired with an [else](#) clause. If the loop finishes without executing the [break](#), the [else](#) clause executes.

In a [for](#) loop, the [else](#) clause is executed after the loop finishes its final iteration, that is, if no break occurred.

In a [while](#) loop, it's executed after the loop's condition becomes false.

In either kind of loop, the [else](#) clause is **not** executed if the loop was terminated by a [break](#). Of course, other ways of ending the loop early, such as a [return](#) or a raised exception, will also skip execution of the [else](#) clause.

This is exemplified in the following [for](#) loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.6. pass Statements ¶

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

4.7. match Statements ¶

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
```

```

case 418:
    return "I'm a teapot"
case _:
    return "Something's wrong with the internet"

```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```

case 401 | 403 | 404:
    return "Not allowed"

```

Patterns can look like unpacking assignments and can be used to bind variables:

```

# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")

```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the *variable binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to (`"x"`, `"y"`), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what.

Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the “(...)” next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 2), Point(3, 4)]
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don’t match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the “bandwidth” and “latency” values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
    ...
```

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

4.8. Defining Functions ¶

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n): # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function [definition](#). It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or [docstring](#). (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The [execution](#) of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a [global](#) statement, or, for variables of enclosing functions, named in a [nonlocal](#) statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using [call by object reference](#) (where the *value* is always an object *reference*, not the value of the object). When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a [return](#) statement do return a value, albeit a rather boring one. This value is called [None](#) (it's a built-in name). Writing the value [None](#) is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using [print\(\)](#):

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The [return](#) statement returns with a value from a function. [return](#) without an expression argument returns [None](#). Falling off the end of a function also returns [None](#).
- The statement `result.append(a)` calls a [method](#) of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using [classes](#), see [Classes](#).) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9. More on Defining Functions ¶

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values ¶

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument:

```
parrot('Do you really want to quit?')
```

- giving one of the optional arguments:

```
parrot('OK to overwrite the file?', 2)
```

- or even giving all arguments:

```
parrot('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

but all the following calls would be invalid:

```
parrot()
parrot('dead', 'no', 'nope', 'extra')
parrot(110, voltage=220)
parrot(actor='John Cleese')
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a [tuple](#) containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`). For example, if we define a function like this:

```
def cheeseshop(kind, *, arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
```

```

print("-" * 40)
for kw in keywords:
    print(kw, ":", keywords[kw])

```

It could be called like this:

```

cheeseshop("Limburger", "It's very runny, sir.", "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin", client="John Cleese", sketch="Cheese Shop Sketc

```

and of course it would print:

```

-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.9.3.. Special parameters ¶

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    # Positional-only parameters before '/'
    # Positional-or-keyword parameters between '/' and '*'
    # Keyword-only parameters after '*'

```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

4.9.3.1. Positional-or-Keyword Arguments ¶

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.9.3.2. Positional-Only Parameters ¶

Certain parameters can be marked as *positional-only*. If *positional-only*, the parameters' order matters, and they cannot be passed by keyword. Positional-only parameters are placed before a `/`. The `/` is used to logically separate the positional-only parameters from the rest. If there is no `/`, there are no positional-only parameters.

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

4.9.3.3. Keyword-Only Arguments ¶

To mark parameters as *keyword-only*, place an `*` in the argument list before the first *keyword-only* parameter.

4.9.3.4. Function Examples ¶

Consider these function definitions:

```
>>> def standard_arg(arg):
...     print(arg)
>>> def pos_only_arg(arg, /):
...     print(arg)
>>> def kwd_only_arg(*, arg):
...     print(arg)
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

The first function, `standard_arg`, places no restrictions; arguments may be passed by position or keyword.

```
>>> standard_arg(2)
2
```

```
>>> standard_arg(arg=2)
2
```

The second function, `pos_only_arg`, only accepts positional arguments:

```
>>> pos_only_arg(1)
1
```

```
>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword argumen
```

The third function, `kwd_only_arg`, only accepts keyword arguments:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given
```

```
>>> kwd_only_arg(arg=3)
3
```

The last function, `combined_example`, uses all conventions:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given
```

```
>>> combined_example(1, 2, kwd_only=3)
1 2 3
```

```
>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword arg
```

Finally, consider this function with potential collision between positional argument and `**kwargs`:

```
def foo(name, **kws):
    return 'name' in kws
```

There is no call that makes it return `True`, as `'name'` will always bind to the first parameter:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
```

But using `/` (positional-only arguments), it is possible since it allows `name` as a positional argument and `'name'` as a key:

```
>>> def foo(name, /, **kws):
...     return 'name' in kws
>>> foo(1, **{'name': 2})
True
```

In other words, the names of positional-only parameters can be used in `**kws` without ambiguity.

4.9.3.5. Recap ¶

The use case will determine which parameters to use in the function definition:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

As guidance:

- Use **positional-only** if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
- Use **keyword-only** when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
- For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

4.9.4. Arbitrary Argument Lists ¶

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these *variadic* arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep='/'):
...     return sep.join(args)
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep='.')
'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists ¶

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate `start` and `stop` arguments. If they are not available separately, write the function call with the `*` operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**` operator:

```
>>> def parrot(voltage, *, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...     print("-- E's", state, "!")
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' de
```

4.9.6. Lambda Expressions ¶

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument. For instance, `list.sort()` takes a sorting key function `key` which can be a lambda function:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.9.7. Documentation Strings ¶

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything."""
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.
No, really, it doesn't do anything.
```

4.9.8. Function Annotations ¶

[Function annotations](#) are completely optional metadata information about the types used by user-defined functions (see [PEP 3107](#) and [PEP 484](#) for more information).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str='eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10. Intermezzo: Coding Style ¶

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about [coding style](#). Most languages can be written (or more concise, [formatted](#)) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, [PEP 8](#) has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters.
This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use [UpperCamelCase](#) for classes and [lowercase_with_underscores](#) for functions and methods. Always use [self](#) as the name for the first method argument.
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

Footnotes

[1]: Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

More Control Flow Tools — Python

3.13.5 documentation

4.1. `if` Statements¶

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for `else if`, useful to avoid excessive indentation. An `if ... elif ...` sequence is a substitute for `switch` or `case` statements in other languages.

If you are comparing the same value to several constants, the `match` statement may be useful. See [match Statements](#) for details.

4.2. `for` Statements¶

The `for` statement in Python iterates over the items of any sequence, in the order that they appear, unlike in C or Pascal where you might iterate over an arithmetic progression.

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

# Output:
# cat 3
# window 6
# defenestrate 12
```

Modifying a collection while iterating over it can be tricky. Common strategies include iterating over a copy:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]
```


Or creating a new collection:

```
# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function ¶

Used to iterate over a sequence of numbers. It generates arithmetic progressions:

```
for i in range(5):
    print(i)
```

Output:

```
# 0
# 1
# 2
# 3
# 4
```

The end point is not included; `range(10)` yields 10 values (indices 0-9). You can specify start, stop, and step:

```
list(range(5, 10))          # [5, 6, 7, 8, 9]
list(range(0, 10, 3))       # [0, 3, 6, 9]
list(range(-10, -100, -30)) # [-10, -40, -70]
```

To iterate over the indices of a sequence:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
# Output:
# 0 Mary
# 1 had
# 2 a
# 3 little
# 4 lamb
```

Generally, it's more convenient to use `enumerate()`. `range()` objects are iterable, saving space compared to lists.

```
sum(range(4)) # 6
```

4.4. `break` and `continue` Statements ¶

`break` exits the innermost loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
    else:
```

```
# Loop fell through without finding a factor
print(f"{n} is a prime number")
```

`continue` skips to the next iteration:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
```

4.5. `else` Clauses on Loops¶

In loops, the `break` statement can be paired with an `else` clause, which executes if no `break` occurs during the loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
    else:
        # loop fell through without finding a factor
        print(f"{n} is a prime number")
```

`else` executes after the loop's final iteration if `break` was not encountered. It is similar to the `else` of a `try` statement.

Example: search for primes with `break` :

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
    else:
        print(f"{n} is a prime number")
```

4.6. `pass` Statements¶

`pass` does nothing; it's used where syntactically necessary but no action is needed.

```
while True:
    pass # Busy-wait for keyboard interrupt
```

For minimal classes:

```
class MyEmptyClass:
    pass
```

As a placeholder body during development:

```
def initlog(*args):
    pass # Remember to implement later
```

4.7. `match` Statements¶

`match` compares an expression's value against patterns. It resembles switch-case in other languages but supports pattern matching and extraction:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

The `_` wildcard matches anything. Patterns can combine literals with `|`:

```
case 401 | 403 | 404:
    return "Not allowed"
```

Unpacking and binding patterns:

```
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Class pattern matching:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Class pattern matching can also use `__match_args__` to specify attributes:

```
Point.__match_args__ = ('x', 'y')
```

Matching sequences of points:

```

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 1)]
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

Guards:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")

```

Key features:

- Sequence patterns support extended unpacking: `[x, y, *rest]` or `(x, y, *rest)`.
- Mapping patterns like `{"bandwidth": b, "latency": l}` extract dictionary values.
- Subpatterns can be captured via `as`.
- Literals are compared by equality, except `True`, `False`, and `None`, which use identity.
- Named constants: e.g., `Color.RED`.

Refer to [PEP 636](#) for more detailed pattern matching.

4.8. Defining Functions¶

Create a Fibonacci function:

```

def fib(n):
    """Print a Fibonacci series less than n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()

```

Call:

```

fib(2000)
# Output:
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

```

A function:

- Begins with `def` followed by name and parentheses.

- Can include a docstring.
- Executes in a local symbol table.
- Arguments are passed via call by object reference.
- Functions are objects that can be assigned or passed.

4.9. More on Defining Functions ¶

4.9.1. Default Argument Values

Specify defaults:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries -= 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Usage:

```
ask_ok('Do you really want to quit?') # with default retries and reminder
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

Warning: default values are evaluated once at function definition time. Mutable defaults can cause issues:

```
def f(a, L=[]):
    L.append(a)
    return L
```

```
print(f(1))
print(f(2))
print(f(3))
# Output:
# [1]
# [1, 2]
# [1, 2, 3]
```

To avoid this, use `None`:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments

Functions can accept optional arguments as key-value pairs:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

Call in various ways:

```
parrot(1000) # positional
parrot(voltage=1000) # keyword
parrot(voltage=1000000, action='V00000M') # mixed
parrot('a million', 'bereft of life', 'jump') # all positional
parrot('a thousand', state='pushing up the daisies') # positional + keyword
```

Invalid calls (e.g., missing required, duplicate arg, unknown arg):

```
parrot()
parrot(voltage=5.0, 'dead')
parrot(110, voltage=220)
parrot(actor='John Cleese')
```

Arguments after positional are keywords only if specified. When using `**` in function args, it captures remaining keyword arguments into a dict:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Example call:

```
cheeseshop("Limburger",
    "It's very runny, sir.",
    "It's really very, VERY runny, sir.",
    shopkeeper="Michael Palin",
    client="John Cleese",
    sketch="Cheese Shop Sketch")
```

4.9.3. Special parameters¶

4.9.3.1. Positional-or-Keyword Arguments¶

No special markers; arguments can be positional or keyword.

4.9.3.2. Positional-Only Parameters¶

Use `/` in parameter list:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    pass
```

Positional-only parameters are before `/`.

4.9.3.3. Keyword-Only Arguments¶

Use `*` in parameter list:

```
def f(pos1, *, kwd1, kwd2):
    pass
```

4.9.3.4. Function Examples

```
def standard_arg(arg):
    print(arg)
```

```
def pos_only_arg(arg, /):
    print(arg)
```

```
def kwd_only_arg(*, arg):
    print(arg)
```

```
def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

4.9.3.5. Recap

Use `/` for positional-only, `*` for keyword-only. They can be combined to restrict argument passing modes.

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    pass
```

4.9.4. Arbitrary Argument Lists¶

Allow functions to accept any number of positional arguments:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Usage:

```
def concat(*args, sep='/'):
    return sep.join(args)
```

```
concat("earth", "mars", "venus") # 'earth/mars/venus'
concat("earth", "mars", "venus", sep='.') # 'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists¶

Unpack argument lists from a sequence/dict:

```
# List unpacking
args = [3, 6]
list(range(*args))
# Dict unpacking
d = {'voltage': 'four million', 'state': 'bleedin\ demised', 'action': 'VOOM'}
parrot(**d)
```

4.9.6. Lambda Expressions¶

Create anonymous functions:

```
def make_incrementor(n):  
    return lambda x: x + n
```

```
f = make_incrementor(42)  
f(0) # 42  
f(1) # 43
```

Use as key functions:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]  
pairs.sort(key=lambda pair: pair[1])  
# Sorted by second element
```

4.9.7. Documentation Strings¶

Conventions:

- Short summary as first line, capitalized, end with period.
- One blank line after summary.
- More details follow in paragraphs.

Indentation in multi-line strings is stripped based on the first non-blank line after the initial line.

```
def my_function():  
    """Do nothing, but document it."""  
    pass
```

```
print(my_function.__doc__)  
# Output:  
# Do nothing, but document it.  
#  
# No, really, it doesn't do anything.
```

4.9.8. Function Annotations¶

Optional metadata about arguments and return types:

```
def f(ham: str, eggs: str = 'eggs') -> str:  
    print("Annotations:", f.__annotations__)  
    print("Arguments:", ham, eggs)  
    return ham + ' and ' + eggs
```

Stored in the `__annotations__` dict; do not affect runtime behavior.

4.10. Intermezzo: Coding Style¶

Adopts [PEP 8](#) recommendations for readable code:

- 4 spaces per indentation level.
- Wrap lines exceeding 79 characters.
- Use blank lines to separate functions, classes, blocks.

- Use comments and docstrings.
- Spaces around operators and after commas.
- Consistent naming conventions (`UpperCamelCase` for classes, `lowercase_with_underscores` for functions).
- Use UTF-8 encoding; avoid non-ASCII identifiers in international contexts.

For more details, see the [PEP 8](#).

Navigation

- [index](#)
- [modules](#)
- [next: Data Structures](#)
- [previous: An Informal Introduction to Python](#)

This page

- [Report a bug](#)
- [Show source](#)

© [Python Software Foundation](#), 2001-2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples and code are additionally licensed under the Zero Clause BSD License.

[History and License](#) for more info.

Please [donate](#).

Last updated: Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created with [Sphinx](#) 8.2.3.

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword [‘elif’](#) is short for ‘else if’, and is useful to avoid excessive indentation. An `[if](../reference/compound_stmts.html#if)` ... `[elif](../reference/compound_stmts.html#elif)` ... `[else](../reference/compound_stmts.html#else)` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [Match Statements](#).

4.2. for Statements

The `[for](../reference/compound_stmts.html#for)` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `[for](../reference/compound_stmts.html#for)` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}
```

```
# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is *iterable*, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `[for](../reference/compound_stmts.html#for)` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. break and continue Statements

The `break` statement breaks out of the innermost enclosing `[for](../reference/compound_stmts.html#for)` or `[while](../reference/compound_stmts.html#while)` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `[continue](../reference/simple_stmts.html#continue)` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. else Clauses on Loops

In a `[for](../reference/compound_stmts.html#for)` or `[while](../reference/compound_stmts.html#while)` loop the `[break](../reference/simple_stmts.html#break)` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `[for](../reference/compound_stmts.html#for)` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred. In a `[while](../reference/compound_stmts.html#while)` loop, it's executed after the loop's condition becomes false.

In either case, the `else` clause is **not** executed if the loop was terminated by a `break`. Other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, *not* the `if` statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [Handling Exceptions](#).

4.6. pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

4.7. match Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a `switch` statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point(x=x, y=y):
            print(f"X={x}, Y={y}")
        case _:
            print("Not a point")

```

You can specify attributes to be captured using the class name followed by patterns with attributes, similar to unpacking:

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

Only the standalone names (like `var`) are assigned to by a `match` statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the parentheses (...), like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")

```

```

case Point(x, y):
    print(f"Not on the diagonal")

```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```

case (Point(x1, y1), Point(x2, y2) as p2):
    ...

```

which will capture the second element of the input as `p2` (as long as the input is a sequence of two points).

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variables:

```

from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

```

```

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

```

```

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")

```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```

>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> # Now call the function we just defined:

```



```
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `[def](../reference/compound_stmts.html#def)` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, they are named in a `global` statement, or, for variables of enclosing functions, they are named in a `nonlocal` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by object reference* [1] (where the *value* is always an object *reference*, not the value of the object). When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):    # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that ‘belongs’ to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods using [classes](#).) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument:

```
ask_ok('Do you really want to quit?')
```

- giving one of the optional arguments:

```
ask_ok('OK to overwrite the file?', 2)
```

- or even giving all arguments:

```
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

This example also introduces the [in](#) keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that:

```
i = 5
```

```
def f(arg=i):
    print(arg)
```

```
i = 6
f()

will print 5.
```

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

This will print:

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments

Functions can also be called using [keyword arguments](#) of the form `kwarg=value`. For example, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()
parrot(voltage=5.0, 'dead')
```

```
parrot(110, voltage=220)
parrot(actor='John Cleese')
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g., `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g., `parrot(voltage=1000)` is valid too). No argument may receive a value more than once.

Here's an example that fails due to this restriction:

```
def function(a):
    pass

function(0, a=0)
# Traceback (most recent call last):
# ...
# TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a [tuple](#) containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("--" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.9.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           | Positional or keyword |
    |           |           |
    |           |           | - Keyword only
    -- Positional only
```

4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.9.3.2. Positional-Only Parameters

Certain parameters can be marked as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. They are placed before a `/` (forward slash).

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

4.9.3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, place an `*` in the argument list just before the first *keyword-only* parameter.

4.9.3.4. Function Examples

Consider the following examples:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
...
>>> standard_arg(2)
2
>>> pos_only_arg(1)
1
>>> kwd_only_arg(arg=3)
```

```

3
>>> combined_example(1, 2, kwd_only=3)
1 2 3

```

and the following invalid calls:

```

>>> standard_arg()
# Error
>>> pos_only_arg(arg=1)
# Error
>>> kwd_only_arg(3)
# Error
>>> combined_example(1, 2, 3)
# Error
>>> combined_example(1, 2, kwd_only=3)
# OK
>>> combined_example(pos_only=1, standard=2, kwd_only=3)
# Error

```

4.9.4. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Zero or more normal arguments may occur before the variable argument list:

```

def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))

>>> def concat(*args, sep='/'):
...     return sep.join(args)
...
>>> concat('earth', 'mars', 'venus')
'earth/mars/venus'
>>> concat('earth', 'mars', 'venus', sep='.')
'earth.mars.venus'

```

4.9.5. Unpacking Argument Lists

Arguments that are already in a list or tuple can be unpacked for a function requiring separate positional arguments, using the `*` operator:

```

>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]

```

Similarly, dictionaries can deliver keyword arguments with the `**` operator:

```

>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...
>>> d = {'voltage': 'four million', 'state': 'bleedin\ demised', 'action': 'VOOM'}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' de

```

4.9.6. Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments:

```
lambda a, b: a + b
```

Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument. For instance, `list.sort()` takes a sorting key function `key` which can be a lambda function:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.9.7. Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything."""
...     pass
```

```
...
>>> print(my_function.__doc__)
Do nothing, but document it.
No, really, it doesn't do anything.
```

4.9.8. Function Annotations

[Function annotations](#) are optional metadata about parameter types and return values. They are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on the function itself. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a `->`, followed by an expression, between the parameter list and the colon ending the `def` statement. Example:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10. Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, [PEP 8](#) has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs, e.g., `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `UpperCamelCase` for classes and `lowercase_with_underscores` for functions and methods. Always use `self` as the name for the first method argument.
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best.
- Likewise, don't use non-ASCII characters in identifiers if there is a chance that people speaking a different language will read or maintain the code.

Footnotes

[1] Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword [‘elif’](#) is short for ‘else if’, and is useful to avoid excessive indentation. An `[if](../reference/compound_stmts.html#)` ... `[elif](../reference/compound_stmts.html#)` ... `[elif](../reference/compound_stmts.html#)` ... sequence is a substitute for the `[switch](../reference/compound_stmts.html#)` or `[case](../reference/compound_stmts.html#)` statements found in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, you may also find the `[match](#match-statements)` statement useful. For more details see [match Statements](#).

4.2. for Statements

The `[for](../reference/compound_stmts.html#)` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `[for](../reference/compound_stmts.html#)` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}
```

```
# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

In many ways, the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is `iterable`, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `[for](../reference/compound_stmts.html#)` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. break and continue Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. else Clauses on Loops

In a `[for](../reference/compound_stmts.html#)` or `[while](../reference/compound_stmts.html#)` loop, the `break` statement may be paired with an `[else](../reference/compound_stmts.html#)` clause. If the loop finishes without executing the `break`, the `else` clause executes:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
```

```

...         if n % x == 0:
...             print(f"{n}, 'equals', {x}, '*', {n//x}")
...             break
...     else:
...         # loop fell through without finding a factor
...         print(f"{n} is a prime number")
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3

```

This is exemplified in the following `for` loop, which searches for prime numbers. (Note: the `else` belongs to the `for`, not the `if`.)

Note: The provided code is correct; the `[else] (#)` belongs to the `[for] (#)` loop, not the `[if] (#)`.

4.6. pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```

>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...

```

This is commonly used for creating minimal classes:

```

>>> class MyEmptyClass:
...     pass
...

```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```

>>> def initlog(*args):
...     pass # Remember to implement this!
...

```

4.7. match Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java, or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed, and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a wildcard and never fails to match. If no case matches, none of the branches are executed.

You can combine several literals in a single pattern using `|` (“or”):

```
match status:
    case 401 | 403 | 404:
        return "Not allowed"
```

Patterns can look like unpacking assignments and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals and can be thought of as an extension of the literal pattern shown above. The next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, similar to the unpacking assignment `(x, y)=point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point(x=x, y=y):
            print(f"X={x}, Y={y}")
```

```

    case _:
        print("Not a point")

```

Patterns can be arbitrarily nested. For example, if we have a list of Points with `__match_args__` added, we could match it like this:

```

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 2), Point(0, 5), Point(3, 4)]

```

```

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

We can add an `[if] (#)` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")

```

Additional key features:

- Like unpacking assignments, tuple and list patterns match arbitrary sequences (except iterators or strings).
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` are similar to unpacking assignments.
- Mapping patterns like `{"bandwidth": b, "latency": l}` capture dictionary values, ignoring extra keys.
- Subpatterns can be captured with `as`, e.g., `case (Point(x, y) as p2):`.
- Most literals are compared by equality, except `True`, `False`, and `None`, which are compared by identity.
- Patterns may use named constants, denoted with dotted names to prevent interpretation as capture variables.
- Patterns can be nested arbitrarily, including lists of Points or other objects with `__match_args__`.

For a more detailed explanation and additional examples, see [PEP 636](#).

4.8. Defining Functions

We can create a function that writes the Fibonacci series up to an arbitrary boundary:

```
>>> def fib(n): # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to generate documentation, so it's good practice to include them.

The *execution* of a function creates a new symbol table for local variables. All variable assignments inside the function are stored in this local table, and variable references first look there. To access global variables, you can declare them with the `global` or `nonlocal` statements.

Calling a function passes arguments via *call by object reference*, meaning that mutable objects passed into a function can be modified by it, affecting the caller's object.

Functions are objects themselves. When you define a function, it is bound to its name in the current symbol table. You can assign that to other variable names:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Functions without a `return` statement implicitly return `None`.

4.9. More on Defining Functions

It's possible to define functions with variable arguments, default argument values, keyword arguments, and more.

4.9.1. Default Argument Values

You can specify default values for arguments, allowing the function to be called with fewer arguments:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries -= 1
        if retries < 0:
```



```
        raise ValueError('invalid user response')
    print(reminder)
```

This function can be called with just the prompt, or with one or both optional arguments specified explicitly.

Note: Default argument values are evaluated when the function is defined, not each time it is called. Using mutable objects as defaults can lead to shared state:

```
def f(a, L=[]):
    L.append(a)
    return L
```

Calling `f(1)` multiple times accumulates values in the same list. To avoid this, use `None` as default:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments

Functions can be called with arguments specified as `name=value`. For example:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state + "!")
```

Call it like:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot(action='V00000M', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

Invalid calls include missing required arguments or duplicate arguments.

Arguments passed as keyword arguments must follow positional arguments. All keyword arguments must match parameters in the function definition. Mixing positional and keyword arguments is allowed with restrictions.

Matching argument names with parameter names, especially with special parameters like `*` and `**`, enables flexible calling conventions.

4.9.3. Special parameters

Functions can restrict argument passing orders with:

- `/` to mark positional-only parameters
- `*` to mark keyword-only parameters

Example:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
```

- Parameters before `/` are positional-only; cannot be passed by keyword.
- Parameters after `*` are keyword-only.
- Parameters between `/` and `*` can be positional or keyword.

Various combinations are possible, including using `*args`, `**kwargs`, and `*` for flexible argument passing.

4.9.4. Arbitrary Argument Lists

A function can accept any number of positional arguments with `*args`, which are received as a tuple:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

It can be called with normal arguments followed by any number of extra positional arguments:

```
def concat(*args, sep='/'):
    return sep.join(args)

concat("earth", "mars", "venus")
# 'earth/mars/venus'
```

Arguments after `*args` are keyword-only.

4.9.5. Unpacking Argument Lists

Unpacking can be used to pass arguments stored in lists or tuples:

```
args = [3, 6]
list(range(*args))
# [3, 4, 5]

d = {'voltage': 'four million', 'state': 'bleedin' demised'}
parrot(**d)
# -- This parrot wouldn't VROOM if you put four million volts through it. E's bleedin'
```

4.9.6. Lambda Expressions

Small anonymous functions can be created with `lambda`:

```
# Returns the sum of its two arguments
lambda a, b: a + b
```

Useful for small functions passed as arguments, e.g., as `key` in `list.sort()`:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# sorts by second element
```

4.9.7. Documentation Strings

Docstrings should start with a short, concise summary, beginning with a capital letter and ending with a period. Followed by a blank line and further details if needed. Indentation should be consistent, and triple quotes (`"""`) are used.

```
def my_function():
    """Do nothing, but document it."""
    pass
```

Accessed as `__doc__` attribute.

4.9.8. Function Annotations

Annotations are optional metadata about parameters and return value, stored in the function's `__annotations__` dictionary.

```
def f(ham: str, eggs: str = 'eggs') -> str:
    """Example with annotations."""
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs
```

4.10. Intermezzo: Coding Style

To improve code readability, adhere to [PEP 8](#), Python's style guide, which recommends:

- Use 4 spaces for indentation, no tabs
 - Wrap lines to max 79 characters
 - Use blank lines to separate functions, classes, larger code blocks
 - Place comments on their own lines where possible
 - Use docstrings for documentation
 - Use spaces around operators, after commas
 - Name classes with `UpperCamelCase`, functions with `lowercase_with_underscores`
 - Always use `self` as first argument in methods
 - Avoid non-ASCII characters in identifiers
 - Prefer UTF-8 encoding
-

Related Links

- [Report a bug](#)
 - [Show source](#)
 - [Python](#)
 - [Index](#)
 - [Modules](#)
 - [Next](#)
 - [Previous](#)
 - [Chapter 3.13.5 Documentation](#)
 - [The Python Tutorial](#)
-

Created using [Sphinx](#) 8.2.3.

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

4. More Control Flow Tools — Python 3.13.5 documentation

4.1. `if` Statements¶

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for "else if". This sequence acts as a substitute for switch/case statements in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, the `match` statement is useful. See below.

4.2. `for` Statements ¶

The `for` statement in Python iterates over any sequence (list, string, etc.) in the order that they appear. Example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
# Output:
# cat 3
# window 6
# defenestrate 12
```

Modifying collections while iterating can be tricky. Better to iterate over a copy or create a new collection.

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function ¶

Useful for iterating over sequences of numbers:

```
for i in range(5):
    print(i)
# Output: 0 1 2 3 4
```

`range(10)` generates 10 values (0 to 9). Can specify start, stop, step:

```
list(range(5,10))      # [5, 6, 7, 8, 9]
list(range(0,10,3))    # [0, 3, 6, 9]
list(range(-10, -100, -30)) # [-10, -40, -70]
```

To iterate over sequence indices:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
# Output:
# 0 Mary
# 1 had
# 2 a
```

```
# 3 little
# 4 lamb
```

`enumerate()` is often more convenient.

`range()` objects are iterable but not lists, so they save space.

```
sum(range(4)) # 6
```

4.4. `break` and `continue` Statements ¶

`break` exits the innermost loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
```

`continue` skips to the next iteration:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
```

4.5. `else` Clauses on Loops ¶

In `for` or `while` loops, `else` executes if the loop completes normally (not via `break`):

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
    else:
        print(n, 'is a prime number')
# Results: 2, 3, 5, 7 identified as prime
```

The `else` runs only if no `break` is encountered.

4.6. `pass` Statements ¶

`pass` does nothing; used as placeholders:

```
while True:
    pass # Busy-wait for Ctrl+C
```

In class definitions:

```
class MyEmptyClass:
    pass
```

In functions:

```
def initlog(*args):
    pass # To be implemented later
```

4.7. match Statements

Pattern matching introduced in Python 3.10. It compares a value against multiple patterns:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

The `_` acts as a wildcard. Pattern matching supports literals, unpacking, class patterns, guards (`if`), etc. For more details, see [PEP 636](#).

4.8. Defining Functions

Create flexible functions, e.g., Fibonacci series:

```
def fib(n):
    """Print Fibonacci series less than n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()
```

Calling `fib(2000)` prints Fibonacci numbers less than 2000.

Functions have self-contained symbol tables; arguments are passed by object reference (call by object reference). Functions can be assigned to variables:

```
fib = fib
fib(100)
```

Even functions without explicit `return` return `None`. To return values:

```
def fib2(n):
    """Return Fibonacci series up to n as a list."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a + b
    return result
```

4.9. More on Defining Functions

4.9.1. Default Argument Values

Specify default values:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
```

```

reply = input(prompt)
if reply in {'y', 'ye', 'yes'}:
    return True
if reply in {'n', 'no', 'nop', 'nope'}:
    return False
retries -= 1
if retries < 0:
    raise ValueError('invalid user response')
print(reminder)

```

Defaults are evaluated at definition. Beware mutable default values:

```

def f(a, L=[]):
    L.append(a)
    return L

```

```

print(f(1)) # [1]
print(f(2)) # [1, 2]

```

To avoid sharing, use `None`:

```

def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

```

4.9.2. Keyword Arguments¶

Functions accept keyword arguments:

```

def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")

```

Call as:

```

parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot('a million', 'bereft of life', 'jump', type='Norwegian Blue')

```

Invalid calls:

```

parrot()
parrot(voltage=5.0, 'dead') # positional after keyword
parrot(110, voltage=220)    # multiple values for 'voltage'
parrot(actor='John Cleese') # unknown argument

```

Arguments after positional are prohibited.

Also, `**kwargs` collects extra keyword args as dict. Use `*args` for positional varargs:

```

def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:

```



```

    print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])

```

Call:

```

cheeseshop('Limburger', "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin', client='John Cleese',
           sketch='Cheese Shop Sketch')

```

4.9.3. Special Parameters ¶

- **Positional-only:** parameters before `/` cannot be passed by keyword.
- **Keyword-only:** parameters after `*` must be passed by keyword.

Example:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    pass

```

- `pos1`, `pos2`: positional-only
- `pos_or_kwd`: positional or keyword
- `kwd1`, `kwd2`: keyword-only

Examples:

```

def standard_arg(arg):
    print(arg)

```

```

def pos_only_arg(arg, /):
    print(arg)

```

```

def kwd_only_arg(*, arg):
    print(arg)

```

```

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)

```

Calling conventions:

```

standard_arg(2)
pos_only_arg(1)
kwd_only_arg(arg=3)

```

Prohibits:

```

pos_only_arg(arg=1)      # keyword in positional-only
kwd_only_arg(3)          # missing keyword

```

4.9.4. Arbitrary Argument Lists ¶

`*args` collects extra positional args as tuple:

```

def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))

```

Call:

```
def concat(*args, sep='/'):
    return sep.join(args)

concat('earth', 'mars', 'venus') # 'earth/mars/venus'
concat('earth', 'mars', 'venus', sep='.')
```

4.9.5. Unpacking Argument Lists¶

Unpack sequences when calling:

```
args = [3, 6]
list(range(*args)) # same as range(3,6)

d = {'voltage': 'four million', 'state': "bleedin' demised", 'action': 'VOOM'}
parrot(**d)
```

4.9.6. Lambda Expressions¶

Create anonymous functions:

```
add = lambda a, b: a + b
print(add(2, 3)) # 5

# Using with list.sort():
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Sorted by second element
```

4.9.7. Documentation Strings¶

First line: concise, start with capital, end with period.

Followed by optional description. Indentation is stripped based on first non-blank line after the first line.

Example:

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything."""
    pass

print(my_function.__doc__)
```

4.9.8. Function Annotations¶

Optional metadata:

```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

f('spam')
# Annotations: {'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class 'str'>}
# Arguments: spam eggs
# 'spam and eggs'
```

4.10. Intermezzo: Coding Style¶

Key points from [PEP 8](#):

- Use 4 spaces indentation, no tabs.
 - Wrap lines to max 79 characters.
 - Use blank lines to separate functions/classes.
 - Put comments on their own lines where possible.
 - Use docstrings.
 - Spaces around operators and after commas.
 - Class: `UpperCamelCase`, functions: `lowercase_with_underscores`.
 - Use `self` for method first parameter.
 - Avoid non-ASCII characters in identifiers.
 - Use UTF-8 or plain ASCII.
-

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```



- Hello
- World

1. Hello
2. World

Hello World

Example Data

```
console.log("Hello")
```

[Hello](#)



- Hello
- World

1. Hello
2. World

Hello World

Example Data

```
console.log("Hello")
```

[Hello](#)

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for “else if”, and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [match Statements](#).

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```

```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

for i in range(5):
    print(i)
# Output:
# 0
# 1
# 2
# 3
# 4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (step):

```

list(range(5, 10))
# Output: [5, 6, 7, 8, 9]

```

```

list(range(0, 10, 3))
# Output: [0, 3, 6, 9]

```

```

list(range(-10, -100, -30))
# Output: [-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```

a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
# Output:
# 0 Mary
# 1 had
# 2 a
# 3 little
# 4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the

list, thus saving space.

We say such an object is *iterable*, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
sum(range(4)) # 0 + 1 + 2 + 3
# Output:
# 6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. `break` and `continue` Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
# Output:
# 4 equals 2 * 2
# 6 equals 2 * 3
# 8 equals 2 * 4
# 9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
# Output:
# Found an even number 2
# Found an odd number 3
# Found an even number 4
# Found an odd number 5
# Found an even number 6
# Found an odd number 7
# Found an even number 8
# Found an odd number 9
```

4.5. `else` Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred.

In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the `else` clause is *not* executed if the loop was terminated by a `break`. Other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

Example: searching for primes

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')

# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2
# 5 is a prime number
# 6 equals 2 * 3
# 7 is a prime number
# 8 equals 2 * 4
# 9 equals 3 * 3
```

4.6. `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

This is commonly used for creating minimal classes:

```
class MyEmptyClass:
    pass
```

Another place `pass` can be used is as a placeholder for a function or conditional body when working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
def initlog(*args):
    pass # Remember to implement this!
```

4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

Example: HTTP error code

```
def http_error(status):
    match status:
```



```

case 400:
    return "Bad request"
case 404:
    return "Not found"
case 418:
    return "I'm a teapot"
case _:
    return "Something's wrong with the internet"

```

The last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

Patterns can look like unpacking assignments, and can be used to bind variables:

```

# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")

```

You can combine several literals in a single pattern using `|` (“or”):

```

case 401 | 403 | 404:
    return "Not allowed"

```

Patterns can look like unpacking assignments, with variables:

```

# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")

```

More complex patterns:

```

# class Point with __match_args__ set
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(0, 0):

```

```

        print("Origin")
    case Point(0, y):
        print(f"Y={y}")
    case Point(x, 0):
        print(f"X={x}")
    case Point(x, y):
        print(f"X={x}, Y={y}")
    case _:
        print("Not a point")

```

Using class patterns with attribute extraction and `[ ]` for sequences, including nested patterns, extended unpacking with `*rest`, mapping patterns `{"bandwidth": b, "latency": l}`, pattern guards with `if`, and constants.

For more detailed information and examples, see [PEP 636](#).

4.8. Defining Functions

Functions are defined using `def`:

```

def fib(n):
    """Print a Fibonacci series less than n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()

```

Calling `fib(2000)` will print Fibonacci numbers less than 2000.

The first string literal after `def` is the *docstring*. Functions have their own local symbol table. Arguments are passed via *call by object reference*. Functions are first-class objects, so you can assign them or pass them around:

```

fib = fib
fib(100)

```

Even functions without an explicit `return` return `None`. Use `return` to return a value, e.g., a list:

```

def fib2(n):
    """Return a list of Fibonacci numbers less than n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a + b
    return result

```

```

f100 = fib2(100)
# Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

Methods like `.append()` modify the list in-place. It's preferred over concatenation for efficiency.

4.9. More on Defining Functions

4.9.1. Default Argument Values

Default argument values are evaluated once at function definition time. Example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries -= 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Call examples:

```
ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

4.9.2. Keyword Arguments

Functions can be called with `kwarg=value`. For example:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

Calling:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot('a million', 'bereft of life', sketch='Cheese Shop Sketch') # invalid, as keyw
```

Invalid calls:

```
parrot()
parrot(voltage=5.0, 'dead')
parrot(110, voltage=220)
parrot(actor='John Cleese')
```

Keyword arguments must follow positional arguments and match parameter names.

4.9.3. Special Parameters

Using `/` for *positional-only*, `*` for *keyword-only*, and combinations:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    pass
```

- Positional-only parameters: before `/`
- Positional-or-keyword: between `/` and `*`
- Keyword-only: after `*`

Examples:

```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

Use positional-only to prevent parameter names from being used externally. Use keyword-only to enforce explicit naming.

4.9.4. Arbitrary Argument Lists

Using `*args`, e.g.:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Call:

```
def concat(*args, sep='/'):
    return sep.join(args)

concat('earth', 'mars', 'venus') # output: 'earth/mars/venus'
concat('earth', 'mars', 'venus', sep='.')
```

4.9.5. Unpacking Argument Lists

Unpack lists/tuples with `*`:

```
args = [3, 6]
list(range(*args))
# Output: [3, 4, 5]
```

And dictionaries with `**`:

```
d = {'voltage': 'four million', 'state': 'bleedin\ demised'}
parrot(**d)
# Output: -- This parrot wouldn't VROOM if you put four million volts through it. E's b
```

4.10. Lambda Expressions

Small anonymous functions:

```
make_incrementor = lambda n: (lambda x: x + n)
f = make_incrementor(42)
print(f(0)) # Output: 42
print(f(1)) # Output: 43
```

Used as arguments:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Output: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.11. Documentation Strings

Docstrings are used to document objects. First line: concise summary (capitalized, end with period). Followed by optional blank line and detailed description. Use indentation conventions; tools usually strip indentation:

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass
```

Print the docstring with `print(my_function.__doc__)`.

4.12. Function Annotations

Optional metadata about argument and return types. Stored in `__annotations__` as a dict.

```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

f('spam')
```

Output:

```
# Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
# Arguments: spam eggs
# 'spam and eggs'
```

4.13. Intermezzo: Coding Style

Follow [PEP 8](#):

- Use 4 spaces per indentation level; no tabs.
 - Limit lines to 79 characters.
 - Separate functions/classes with blank lines.
 - Put comments on their own line.
 - Use docstrings.
 - Spaces around operators and after commas.
 - Name classes in `UpperCamelCase` and functions/methods in `lowercase_with_underscores`.
 - Use `self` as first method argument.
 - Avoid non-ASCII characters in identifiers.
-

Note: For more resources, see the [Python documentation](#).

More Control Flow Tools — Python

3.13.5 documentation

4.1. `if` Statements¶

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for `else if`, often used to avoid excessive indentation. Sequence `if ... elif ... elif ...` serves as a substitute for switch or case statements in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, the `match` statement is useful. See [Match Statements](#).

4.2. `for` Statements¶

The `for` statement in Python iterates over the items of any sequence (list or string), in order:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Modifying a collection while iterating over it can be tricky; it's clearer to loop over a copy or a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]
```

```
# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function ¶

To iterate over a sequence of numbers, use `range()`:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

`range(10)` generates numbers 0 to 9. You can specify start, stop, and step (even negative):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over indices:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Using `enumerate()` simplifies this:

```
for i, value in enumerate(a):
    print(i, value)
...
0 Mary
1 had
2 a
3 little
4 lamb
```

`range()` objects are iterables, not lists, saving space. They can be used with functions like `sum()`:

```
sum(range(4)) # 0 + 1 + 2 + 3 = 6
```

In `Data Structures`, we will explore more about lists.

4.4. `break` and `continue` Statements ¶

`break` exits the innermost loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n // x}")
            break
    else:
        # loop fell through without finding a factor
        print(f"{n} is a prime number")
```

`continue` skips to the next iteration:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
```

4.5. `else` Clauses on Loops ¶

An `else` clause on a loop executes if the loop completes normally (no break):

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n // x}")
            break
    else:
        # no break, so n is prime
        print(f"{n} is a prime number")
```

Useful for algorithms like primality testing. It runs if no break occurs.

4.6. `pass` Statements ¶

`pass` does nothing; it is used as a placeholder:

```
while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

Minimal class:

```
class MyEmptyClass:
    pass
```

As a placeholder in function or conditional bodies:

```
def initlog(*args):
    pass # Remember to implement this!
```

4.7. `match` Statements ¶

`match` compares an expression against patterns, similar to switch in other languages but more powerful:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Wildcard `_` matches anything. Patterns can combine literals (using `|`), unpacked tuples, dictionaries, class patterns, and more, including guards (`if` conditions). For detailed info, see [PEP 636](#).

4.8. Defining Functions¶

Create Fibonacci series function as an example:

```
def fib(n):
    """Print Fibonacci series less than n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()
```

Call the function:

```
fib(2000)
```

Functions are defined with `def`, with an optional docstring, local symbol tables, and arguments passed by object reference (`call by object reference`). Functions can be assigned to other names:

```
fib
f = fib
f(100)
```

Functions without return statements return `None`, but you can explicitly return a value or list of values, e.g.:

```
def fib2(n):
    """Return a list containing Fibonacci numbers less than n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a + b
    return result
```

4.9. More on Defining Functions¶

4.9.1. Default Argument Values¶

Set default values for arguments:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries -= 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Call in various ways:

```
ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

Default parameters are evaluated at function definition time. Use a `None` default for mutable objects:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments ¶

Call functions with `kwarg=value`:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

Examples:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V000000M')
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

Invalid calls:

```
parrot()
parrot(voltage=5.0, 'dead') # non-keyword after keyword
parrot(110, voltage=220)    # duplicate value
parrot(actor='John Cleese') # unknown argument
```

Keyword arguments must follow positional ones and match in name.

4.9.3. Special Parameters ¶

Supports:

- Positional-only parameters (`/`)

- Keyword-only parameters (*)
- Combination

Examples:

```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

Use these to enforce call conventions.

4.9.4. Arbitrary Argument Lists¶

Use `*args` and `**kwargs` to accept variable positional and keyword arguments:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Call example:

```
def concat(*args, sep='/'):
    return sep.join(args)

concat("earth", "mars", "venus")
# 'earth/mars/venus'

concat("earth", "mars", "venus", sep='.')
# 'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists¶

Unpack arguments from sequences or dictionaries:

```
list(range(3, 6))
args = [3, 6]
list(range(*args))
# [3, 4, 5]

d = {'voltage': 'four million', 'state': 'bleedin' demised'}
parrot(**d)
# -- This parrot wouldn't VROOM if you put four million volts through it. E's bleedin'
```

4.9.6. Lambda Expressions¶

Create small anonymous functions:

```
add = lambda a, b: a + b
make_incrementor = lambda n: lambda x: x + n
```

```
f = make_incrementor(42)
f(0) # 42
f(1) # 43
```

Use in functions like `list.sort()`:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Sorted by second element
```

4.9.7. Documentation Strings¶

First line: short summary. Post a blank line, then detailed description. Use consistent indentation conventions.

Example:

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass
```

4.9.8. Function Annotations¶

Optional metadata for functions; stored in `__annotations__`. Define parameter and return annotations with `:` and `->`.

Example:

```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

f('spam')
# {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
```

4.10. Intermezzo: Coding Style¶

- Use 4 spaces per indentation level, no tabs.
- Wrap lines at 79 characters.
- Use blank lines to separate functions/classes.
- Put comments on their own lines when possible.
- Use docstrings.
- Use spaces around operators and after commas.
- Class names: `UpperCamelCase`, functions/methods: `lowercase_with_underscores`.
- Use `self` as the first argument in methods.
- Avoid non-ASCII characters unless necessary.

For more details, see [PEP 8](#).

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword ‘`elif`’ is short for ‘else if’, and is useful to avoid excessive indentation. An `[ if ... elif ... elif ... ]` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [Match Statements](#).

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```

```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
...
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is more convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways, the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is *iterable*, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. `break` and `continue` Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...     else:
...         print(n, "is a prime number")
# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2
# 5 is a prime number
# 6 equals 2 * 3
# 7 is a prime number
# 8 equals 2 * 4
# 9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. `else` Clauses on Loops

In a `for` or `while` loop, the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no break occurred.

In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the `else` clause is *not* executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2
# 5 is a prime number
# 6 equals 2 * 3
# 7 is a prime number
# 8 equals 2 * 4
# 9 equals 3 * 3
```

4.6. `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed, and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

You can use class names followed by an argument list resembling a constructor, with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
```

```

    case Point(x=x, y=y):
        print(f"X={x}, Y={y}")
    case _:
        print("Not a point")

```

Patterns may have a guard, a boolean expression evaluated after successful pattern matching:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")

```

Some key features:

- Tuple and list patterns support arbitrary sequences with extended unpacking, e.g., `[x, y, *rest]`.
- Mapping patterns extract values from dictionaries.
- Patterns can be nested arbitrarily.
- Pattern matching can be extended with class attributes and guard conditions.

4.8. Defining Functions

Here is an example of a Fibonacci series function:

```

>>> def fib(n):
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

```

- The `def` introduces a function, followed by its name and a list of parameters.
- The function body starts after an indented block.
- The first statement can be a string literal, which serves as the *docstring*.
- Variable scope: variables assigned in a function are local; references search local then enclosing then global variables.
- Arguments are passed by object reference (call by value).

Functions are objects, and assignment binds names to functions:

```

>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89

```

Functions without a return statement return `None` (a built-in name). Example:

```

>>> fib(0)
>>> print(fib(0))
None

```

A function can return a list:

```
def fib2(n):
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a) # see below
        a, b = b, a + b
    return result

f100 = fib2(100) # call it
# write the result
print(f100) # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Highlights:

- `return` with a value sends data back to the caller.
- Methods like `list.append()` are functions that belong to objects.
- You can assign a function to another variable.

4.9. More on Defining Functions

Arguments may be specified with default values:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries -= 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This can be called with different argument forms:

```
ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

`in` tests whether a sequence contains a value.

Default arguments are evaluated once at function definition time.

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f() # prints 5
```

Warning: default objects like lists, dictionaries, or instances are shared across calls:

```
def f(a, L=[]):
    L.append(a)
```

```

        return L

print(f(1))
print(f(2))
print(f(3))
# Output:
# [1]
# [1, 2]
# [1, 2, 3]

To avoid this, write:

def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

```

4.9.2. Keyword Arguments

Arguments can be passed explicitly by name:

```

def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")

```

Call it in various ways:

```

parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot(action='V00000M', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')

```

Invalid calls include missing required arguments, duplicate argument names, or unknown keywords:

```

parrot() # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate argument
parrot(actor='John Cleese') # unknown argument

```

Keyword arguments must follow positional arguments. All passed keyword arguments must match the function parameters, and no argument may be assigned more than once.

For functions with variable arguments:

```

def function(a, *b, **c):
    pass

```

- `*b` collects extra positional arguments into a tuple.
- `**c` collects extra keyword arguments into a dict.

4.9.3. Special Parameters

You can control how parameters are passed with `/` and `*`:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):  
    pass
```

- `/` indicates positional-only parameters.
- `*` indicates keyword-only parameters.

4.9.3.1. Positional-or-Keyword Arguments

If neither `/` nor `*` are present, arguments can be passed by position or keyword.

4.9.3.2. Positional-Only Parameters

Arguments placed before `/` can only be passed positionally.

```
def pos_only_arg(arg, /):  
    pass
```

4.9.3.3. Keyword-Only Arguments

Arguments after `*` can only be passed as keywords.

```
def kwd_only_arg(*, arg):  
    pass
```

4.9.3.4. Function Examples

```
def standard_arg(arg):  
    print(arg)
```

```
def pos_only_arg(arg, /):  
    print(arg)
```

```
def kwd_only_arg(*, arg):  
    print(arg)
```

```
def combined_example(pos_only, /, standard, *, kwd_only):  
    print(pos_only, standard, kwd_only)
```

4.9.3.5. Recap

Parameter passing modes depend on the function definition:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):  
    pass
```

Guidelines:

- Use *positional-only* when parameter names should not be exposed or to enforce argument order.
- Use *keyword-only* when you want clearer code and to prevent relying on argument order.
- Use *positional-only* for stable APIs.

4.9.4. Arbitrary Argument Lists

Functions can accept any number of arguments (often called *variadic*):

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Arguments after `*args` are keyword-only:

```
def concat(*args, sep='/'):
    return sep.join(args)

print(concat("earth", "mars", "venus")) # 'earth/mars/venus'
print(concat("earth", "mars", "venus", sep='.')) # 'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists

Lists and tuples can be unpacked into function arguments:

```
args = [3, 6]
list(range(*args))
# Output: [3, 4, 5]
```

Dictionaries can unpack keyword arguments:

```
d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
parrot(**d)
# Prints: -- This parrot wouldn't VOOM if you put four million volts through it. E's b
```

4.9.6. Lambda Expressions

Small anonymous functions are created with `lambda`. Example:

```
add = lambda a, b: a + b
print(add(3, 5)) # 8
```

Lambdas are useful for passing simple functions where function objects are required, e.g., as sorting keys:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Sorted by second element
```

4.9.7. Documentation Strings

Docstrings are short, concise descriptions of functions or classes.

- The first line is a summary, starting with a capital letter and ending with a period.
- Followed by a blank line and more detailed description.
- Use consistent indentation.

Example:

```
def my_function():
    """Do nothing, but document it."""
    pass
```

Access via `__doc__`:

```
print(my_function.__doc__)
# Output:
# Do nothing, but document it.
```

4.9.8. Function Annotations

Annotations are optional metadata about parameter and return types, stored in `__annotations__`.

Example:

```
def f(ham: str, eggs: str = 'eggs') -> str:
    """Function with annotations."""
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs

print(f('spam'))
# Output:
# Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
# Arguments: spam eggs
# 'spam and eggs'
```

4.10. Intermezzo: Coding Style

[PEP 8](#) is the style guide for Python, emphasizing readability:

- Use 4 spaces per indentation level; no tabs.
- Wrap lines at 79 characters.
- Use blank lines to separate functions and classes.
- Put comments on their own lines where possible.
- Use docstrings for documentation.
- Space around operators and after commas.
- Class names with `UpperCamelCase`; function names with `lowercase_with_underscores`.
- Avoid non-ASCII characters in identifiers.
- Use UTF-8 encoding and avoid exotic encodings.

[Further details here.](#)

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for ‘else if’, and is useful to avoid excessive indentation. An `if ... elif ... else` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [Match Statements](#).

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as in C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```



```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

for i in range(5):
    print(i)
# Output:
# 0
# 1
# 2
# 3
# 4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

list(range(5, 10))
# Output: [5, 6, 7, 8, 9]

```

```

list(range(0, 10, 3))
# Output: [0, 3, 6, 9]

```

```

list(range(-10, -100, -30))
# Output: [-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
# Output:
# 0 Mary
# 1 had
# 2 a
# 3 little
# 4 lamb

```

In most such cases, however, it is more convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

range(10)
# Output: range(0, 10)

```

In many ways, the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
sum(range(4)) # 0 + 1 + 2 + 3
# Output: 6
```

Later, we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. `break` and `continue` Statements

The `break` statement breaks out of the innermost enclosing `for` or `while` loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n // x}")
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

The `continue` statement continues with the next iteration of the loop:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
```

4.5. `else` Clauses on Loops

In a `for` or `while` loop, the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred.

In a `while` loop, it's executed after the loop's condition becomes false.

In either case, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following prime number search:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n // x)
            break
    else:
```

```
# loop fell through without finding a factor
print(n, 'is a prime number')
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, not the `if` statement.)

4.6. `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

This is commonly used for creating minimal classes:

```
class MyEmptyClass:
    pass
```

Another place `pass` can be used is as a placeholder for a function or conditional body when working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
def initlog(*args):
    pass # Remember to implement this!
```

4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java, or JavaScript, but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
```

```

        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")

```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g., dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to ("x", "y"), the following patterns are all equivalent (and all bind the `y` attribute to the variable):

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (`foo.bar`), attribute names (`x=`, `y=`) or class names (recognized by the "(...)" next to them like `Point`) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of `Point` s, with `__match_args__` added, we could match it like this:

```

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x

```

```

        self.y = y

points = [Point(0, 0), Point(1, 1), Point(0, 2)]
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

We can add a `guard` to a pattern, which is an `if` clause. If the guard is false, `match` goes on to try the next case block. Note that value capture happens **before** the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")

```

Other Key Features

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similarly to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Extra keys are ignored. `**rest` is also supported (except `_`).
- Subpatterns may be captured using `as`:

```

case (Point(x1, y1), Point(x2, y2) as p2):
    ...

```

- Most literals are compared by equality, but the singletons `True`, `False`, and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variables:

```

from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

```

```

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

```

```

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")

```

```
case Color.BLUE:
    print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#), which is written in a tutorial format.

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
>>>
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function **definition**, followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or **docstring**. (More about docstrings can be found in the section [Documentation Strings](#).) Including docstrings is a good practice.

The execution of a function creates a new symbol table used for local variables. Variable assignments in a function store the value in the local symbol table; references first search locally, then in enclosing functions, then globally, then in built-in names. Global variables and variables of enclosing functions cannot be directly assigned within a function unless explicitly declared with `global` or `nonlocal`.

Arguments are passed using **call by object reference**: the actual parameters to the call are objects, and the function gets references to those objects. When a function calls another, a new local symbol table is created.

A function definition associates a name with the function object in the current symbol table. Other names can point to the same function object and are used to access it:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Even functions without a `return` statement do return a value, namely `None`. You can see this explicitly:

```
>>> fib(0)
>>> print(fib(0))
None
```

To return a value, use `return`:

```
def fib2(n):
    """Return a list containing the Fibonacci series up to n."""
    result = []
```

```

a, b = 0, 1
while a < n:
    result.append(a)
    a, b = b, a + b
return result

f100 = fib2(100) # call it
print(f100)      # write the result
# Output:
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

Key points:

- The `return` statement returns a value from a function. Without an argument, it returns `None`.
- Methods like `list.append()` are used to modify objects in place.
- Default argument values are evaluated only once when the function is defined. Mutable default arguments can lead to unexpected behavior if not handled carefully; see [Default Argument Values](#).

4.9. More on Defining Functions

Functions can have a variable number of arguments, specified with `*args`, `**kwargs`, and default values.

4.9.1. Default Argument Values

Default arguments allow functions to be called with fewer arguments than defined:

```

def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries -= 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)

```

Calling:

```

ask_ok('Do you really want to quit?') # default retries and reminder
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

```

Note: Default values are evaluated at function definition time, which can be problematic with mutable objects:

```

def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
# Output:
# [1]

```

```
# [1, 2]
# [1, 2, 3]
```

To avoid this, use `None` as default:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments

Functions can be called with `kwarg=value` syntax, allowing explicit specification of arguments:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

Examples:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

Invalid calls:

```
parrot()
# missing required argument
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220)    # duplicate argument
parrot(actor='John Cleese') # unknown argument
```

Arguments with `*` and `**` allow flexible parameter specifications, as explained in the [Special parameters](#) section.

4.9.3. Special parameters

Parameters can be marked as:

- Positional-only (`/`)
- Keyword-only (`*`)
- Both

for clarity and API stability. Examples:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    ...
```

4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are absent, arguments can be passed positionally or by keyword.

4.9.3.2. Positional-Only Parameters

Parameters before `/` are positional-only and cannot be passed by keyword.

4.9.3.3. Keyword-Only Arguments

Parameters after `*` are keyword-only.

4.9.3.4. Function Examples

```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

Calling:

```
standard_arg(2)
pos_only_arg(1)
kwd_only_arg(arg=3)
combined_example(1, 2, kwd_only=3)
```

4.9.4. Arbitrary Argument Lists

Functions can accept any number of positional arguments:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Example:

```
def concat(*args, sep='/'):
    return sep.join(args)

concat("earth", "mars", "venus")
# Output: 'earth/mars/venus'

concat("earth", "mars", "venus", sep='.')
# Output: 'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists

You can unpack lists/tuples in function calls with `*`:

```
args = [3, 6]
list(range(*args)) # equivalent to range(3, 6)
# Output: [3, 4, 5]
```

Dictionaries can be used for keyword arguments with `**`:

```
d = {'voltage': 'four million', 'state': 'bleeding demised', 'action': 'VOOM'}
parrot(**d)
# Output: -- This parrot wouldn't VOOM if you put four million volts through it. E's b
```

4.9.6. Lambda Expressions

Small anonymous functions can be created with `lambda`:

```
lambda a, b: a + b
```

Example:

```
def make_incrementor(n):
    return lambda x: x + n
```

```
f = make_incrementor(42)
print(f(0)) # 42
print(f(1)) # 43
```

Lambda functions are useful for small functions or as arguments to higher-order functions like

```
list.sort() :

pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Sorts by second element in each tuple
```

4.9.7. Documentation Strings

Documentation strings (docstrings) should start with a short, concise summary, ending with a period and starting with a capital letter. Followed by a blank line, then more detailed description as needed.

```
def my_function():
    """Do nothing, but document it."""
    pass
```

Indentation is preserved; tools can strip common leading whitespace for processing.

4.9.8. Function Annotations

Annotations are optional metadata stored in `__annotations__`. Parameter annotations follow the parameter with a colon; return annotations use `->`:

```
def f(ham: str, eggs: str = 'eggs') -> str:
    """Example with annotations."""
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs
```

Annotations do not affect function behavior.

4.10. Intermezzo: Coding Style

To write readable Python code:

- Use 4 spaces for indentation (no tabs)
- Wrap lines at 79 characters
- Separate functions/classes with blank lines
- Write comments on separate lines
- Use docstrings
- Put spaces around operators, after commas, but not inside brackets
- Name classes `UpperCamelCase`, functions `lowercase_with_underscores`
- Always use `self` as the first argument of methods
- Use UTF-8 or ASCII, avoid non-ASCII in identifiers

For more details, see [PEP 8](#).

4. More Control Flow Tools¶

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1.. [if](#) Statements¶

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An `if ... elif ... else` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [match Statements](#).

4.2.. [for](#) Statements¶

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s [for](#) statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```

```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3.. The `range()` Function ¶

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
# Output:
# 0
# 1
# 2
# 3
# 4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the indices for items of a sequence of length 10. It's possible to let the range start at another number, or to specify a different increment (even negative):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
# Output:
# 0 Mary
# 1 had
# 2 a
# 3 little
# 4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the

list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](#) statement is such a construct, while an example of a function that takes an iterable is [sum\(\)](#):

```
>>> sum(range(4))
6 # 0 + 1 + 2 + 3
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about [list\(\)](#).

4.4.. [break](#) and [continue](#) Statements¶

The [break](#) statement breaks out of the innermost enclosing [for](#) or [while](#) loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
# Output:
# 4 equals 2 * 2
# 6 equals 2 * 3
# 8 equals 2 * 4
# 9 equals 3 * 3
```

The [continue](#) statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
# Output:
# Found an even number 2
# Found an odd number 3
# Found an even number 4
# Found an odd number 5
# Found an even number 6
# Found an odd number 7
# Found an even number 8
# Found an odd number 9
```

4.5.. [else](#) Clauses on Loops¶

In a [for](#) or [while](#) loop, the [break](#) statement may be paired with an [else](#) clause. If the loop finishes without executing the [break](#), the [else](#) clause executes.

In a [for](#) loop, the [else](#) clause is executed after the loop finishes its final iteration, that is, if no [break](#) occurred.

In a [while](#) loop, it's executed after the loop's condition becomes false.

In either kind of loop, the [else](#) clause is **not** executed if the loop was terminated by a [break](#). Of course, other ways of ending the loop early, such as a [return](#) or a raised exception, will also skip execution of the [else](#) clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2
# 5 is a prime number
# 6 equals 2 * 3
# 7 is a prime number
# 8 equals 2 * 4
# 9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for`, not the `[if]` ([../reference/compound_stmts.html# if](http://reference.compound_stmts.html# if)) statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [Handling Exceptions](#).

4.6.. `pass` Statements ¶

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
# (This is an infinite loop that does nothing, awaiting interruption)
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
```

4.7.. `match` Statements ¶

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a `switch` statement in C, Java or JavaScript, but it's more like pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a wildcard and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Not a point")
```



```

        print("Somewhere else")
    case _:
        print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `(x, y)`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what.

Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names, or class names (recognized by the `"(...)"` next to them, like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 2)]
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")

```

Other key features of the `match` statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments.

- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Extra keys are ignored. `**rest` is also supported.
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
    ...
```

- Most literals are compared by equality; `True`, `False`, and `None` are compared by identity.
- Patterns may use named constants, which must be dotted names to prevent them from being interpreted as capture variables:

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'
color = Color(input("Enter your choice of 'red', 'green', or 'blue': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

4.8.. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of

enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a `global` statement, or, for variables of enclosing functions, named in a `nonlocal` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 0x7f8e91770280>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:

```
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a + b
...     return result
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `<obj>.methodname`, where `<obj>` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using [classes](#).) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example, it is equivalent to `<result> = <result> + [a]`, but more efficient.

4.9.. More on Defining Functions¶

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.. 1. Default Argument Values¶

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
>>> def ask_ok(prompt, retries=4, reminder='Please try again!'):
...     while True:
...         reply = input(prompt)
...         if reply in {'y', 'ye', 'yes'}:
...             return True
...         if reply in {'n', 'no', 'nop', 'nope'}:
...             return False
...         retries = retries - 1
...         if retries < 0:
...             raise ValueError('invalid user response')
...         print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that:

```
>>> i = 5
>>> def f(arg=i):
...     print(arg)
>>> i = 6
>>> f()
5
```

will print `5`.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
>>> def f(a, L=[]):
...     L.append(a)
...     return L
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
```

This will print:

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
>>> def f(a, L=None):
...     if L is None:
...         L = []
...     L.append(a)
...     return L
```

4.9.. 2. Keyword Arguments ¶

Functions can also be called using [keyword arguments](#) of the form `kwarg=value`. For instance, the following function:

```
>>> def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...     print("-- Lovely plumage, the", type)
...     print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
>>> parrot(1000) # 1 positional argument
>>> parrot(voltage=1000) # 1 keyword argument
>>> parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
>>> parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments, order doesn't ma
>>> parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
>>> parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
>>> parrot() # required argument missing
>>> parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
>>> parrot(110, voltage=220) # duplicate value for the same argument
>>> parrot(actor='John Cleese') # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g., `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (`parrot(voltage=1000)` is valid too). No argument may receive a value more than once.

Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a [tuple](#) containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`). For example:

```
def cheeseshop(kind, *, shopkeeper='John', **kwargs):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
```

```

for arg in kwargs:
    print(arg)
    print("-" * 40)
for kw in kwargs:
    print(kw, ":", kwargs[kw])

```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.", shopkeeper="Michael Palin", client="J
```

and of course it would print:

```

-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch

```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.9.. 3. Special parameters¶

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```

>>> def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
...     # pos1 and pos2 are positional-only
...     # pos_or_kwd can be positional or keyword
...     # kwd1 and kwd2 are keyword-only
...     pass

```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also called named parameters.

4.9.. 3.1. Positional-or-Keyword Arguments¶

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.9.. 3.2.. Positional-Only Parameters¶

Certain parameters can be marked as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a `/`. The `/` is used to logically separate the positional-only parameters from the rest:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    # ...

```

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

4.9.. 3.3. Keyword-Only Arguments¶

To mark parameters as *keyword-only* (must be passed by keyword), place a `*` in the arguments list just before the first *keyword-only* parameter:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    # ...
```

4.9.. 3.4. Function Examples¶

Consider these functions:

```
>>> def standard_arg(arg):
...     print(arg)
>>> def pos_only_arg(arg, /):
...     print(arg)
>>> def kwd_only_arg(*, arg):
...     print(arg)
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

- The most familiar, `standard_arg()`, places no restrictions.
- `pos_only_arg()` restricts to only positional arguments.
- `kwd_only_arg()` restricts to only keyword arguments.
- `combined_example()` uses all three.

Calling these functions:

```
>>> standard_arg(2)
2
>>> pos_only_arg(1)
1
>>> kwd_only_arg(arg=3)
3
>>> combined_example(1, 2, kwd_only=3)
1 2 3
>>> combined_example(1, standard=2, kwd_only=3)
1 2 3
>>> combined_example(pos_only=1, standard=2, kwd_only=3)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: combined_example() got some positional-only arguments passed as keyword arg

And this example which mixes the conventions:

```
>>> def foo(name, **kws):
...     return 'name' in kws
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
```

If you want to allow `name` as a keyword argument without ambiguity, define `foo()` as:

```
>>> def foo(name, /, **kwds):  
...     return 'name' in kwds  
>>> foo(1, **{'name': 2})  
True
```

Finally, consider this function with a potential collision:

```
def foo(name, **kwds):  
    return 'name' in kwds
```

which has a call:

```
>>> foo(1, name=2)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: foo() got multiple values for argument 'name'
```

but with `/`:

```
>>> def foo(name, /, **kwds):  
...     return 'name' in kwds  
>>> foo(1, **{'name': 2})  
True
```


4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An ‘[if ... elif ... elif ...](#)’ sequence is a substitute for the [switch](#) or [case](#) statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [match Statements](#).

4.2. for Statements

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s [for](#) statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
```

```

del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The range() Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
...
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](#) statement is such a construct, while an example of a function that takes an iterable is [sum\(\)](#):

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about [list\(\)](#).

4.4. break and continue Statements

The [break](#) statement breaks out of the innermost enclosing [for](#) or [while](#) loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The [continue](#) statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. else Clauses on Loops

In a [for](#) or [while](#) loop the [break](#) statement may be paired with an [else](#) clause. If the loop finishes without executing the [break](#), the [else](#) clause executes.

In a [for](#) loop, the [else](#) clause is executed after the loop finishes its final iteration, that is, if no break occurred.

In a [while](#) loop, it's executed after the loop's condition becomes false.

In either kind of loop, the [else](#) clause is **not** executed if the loop was terminated by a [break](#). Of course, other ways of ending the loop early, such as a [return](#) or a raised exception, will also skip execution of the [else](#) clause.

This is exemplified in the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...         else:
...             # loop fell through without finding a factor
...             print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, not the `if` statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [Handling Exceptions](#).

4.6. pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
...
```

4.7. match Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a `switch` statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point(x=x, y=y):
```

```

        print(f"X={x}, Y={y}")
    case _:
        print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `("x", "y")`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (`foo.bar`), attribute names (`x=`, `y=` above), or class names (`Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```

class Point:
    __match_args__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 2)]

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")

```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.

- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns, e.g., `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported.
- Subpatterns may be captured using the `as` keyword: `case (Point(x1, y1), Point(x2, y2) as p2):` will capture the second element as `p2`.
- Most literals are compared by equality; however, the singletons `True`, `False`, and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variables:

```
from enum import Enum
```

```
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'
```

```
color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))
```

```
match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a + b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function **definition**. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or [docstring](#). (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The [execution](#) of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a [global](#) statement, or, for variables of enclosing functions, named in a [nonlocal](#) statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using [call by object reference](#) (where the [value](#) is always an object [reference](#), not the value of the object). When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a [return](#) statement do return a value, albeit a rather boring one. This value is called [None](#) (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using [print\(\)](#):

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The [return](#) statement returns with a value from a function. [return](#) without an expression argument returns [None](#). Falling off the end of a function also returns [None](#).

- The statement `[result].append(a)` calls a [method](#) of the list object `[result]`. A method is a function that 'belongs' to an object and is named `[obj].methodname`(#), where `[obj]`(#) is some object (this may be an expression), and `[methodname]`(#) is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using [classes](#).)
- Methods of list objects like `[append]`(#) add a new element at the end of the list. It is more efficient than `[result](#) = [result](#) + [[a](#)]`, but accomplishes the same task.

4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument:

```
>>> ask_ok('Do you really want to quit?')
```

- giving one of the optional arguments:

```
>>> ask_ok('OK to overwrite the file?', 2)
```

- or even giving all arguments:

```
>>> ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

but all the following calls would be invalid:

```
>>> ask_ok()
# required argument missing
>>> ask_ok('YES', 'NO')
# non-keyword argument after a keyword argument
>>> ask_ok(110, voltage=220)
# duplicate value for the same argument
>>> ask_ok(actor='John Cleese')
# unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not valid for the `ask_ok` function), and their order is not important. This also includes non-optional arguments (e.g., `ask_ok(voltage=1000)` is valid too). No argument may receive a value more than once.

Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection), which receives a [tuple](#) containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example:

```
def cheeseshop(kind, *, shopkeeper="Michael Palin", **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in keywords:
        print(arg, ":", keywords[arg])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
            "It's really very, VERY runny, sir.",
            shopkeeper="Michael Palin",
            client="John Cleese",
            sketch="Cheese Shop Sketch")
```

and it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

The order of the keyword arguments in the output is guaranteed to match the order in the function call.

4.9.2. Keyword Arguments

Functions can also be called using [keyword arguments](#) of the form `kwarg=value`. For example:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- E's", state, "!")
```

It can be called like:

```

parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword

```

But the following calls are invalid:

```

parrot()
# required argument missing
parrot(5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate value for argument 'voltage'
parrot(actor='John Cleese') # unknown keyword argument

```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g., `actor` is not valid for the `parrot` function), and their order is not important. This also includes non-optional arguments.

In addition, with [kwargs](#), the order of keyword arguments is guaranteed to match the order in which they are provided in the call.

4.9.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict how arguments can be passed so that a developer only needs to look at the function definition to determine if items are passed by position, by position or keyword, or only by keyword.

A function may look like:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    # pos1, pos2 are positional-only
    # pos_or_kwd can be passed by position or keyword
    # kwd1, kwd2 are keyword-only

```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter based on how they can be passed:

- Positional-only
- Positional-or-keyword
- Keyword-only

4.9.3.1. Positional-or-keyword Arguments

If `/` and `*` are not present, arguments can be passed either positionally or as keywords.

4.9.3.2. Positional-Only Parameters

Parameters before `/` are positional-only. Their order matters; they cannot be passed as keywords.

```

def pos_only_arg(arg, /):
    print(arg)

```

Attempting to pass keyword argument will result in a `TypeError`:

```

>>> pos_only_arg(1)
1
>>> pos_only_arg(arg=1)
Traceback (most recent call last):

```

```
File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword argumen
```

4.9.3.3. Keyword-Only Arguments

Parameters after `*` are keyword-only:

```
def kwd_only_arg(*, arg):
    print(arg)
```

Calling via positional argument will cause error:

```
>>> kwd_only_arg(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() missing 1 required keyword-only argument: 'arg'
>>> kwd_only_arg(arg=1)
1
```

4.9.3.4. Function Examples

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
...
```

Calls:

```
>>> standard_arg(2)
2
>>> pos_only_arg(1)
1
>>> kwd_only_arg(arg=3)
3
>>> combined_example(1, 2, kwd_only=3)
1 2 3
>>> combined_example(1, standard=2, kwd_only=3)
1 2 3
>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword arg
```

4.9.4. Arbitrary Argument Lists

Using `*args`, functions can accept any number of positional arguments, which are collected into a tuple:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

In typical use:

```
>>> def concat(*args, sep='/'):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep='.')
'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists

Arguments in a list or tuple can be unpacked when calling a function with `*`:

```
>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]
```

Similarly, dictionaries can be unpacked for keyword arguments with `**`:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.")
...
>>> d = {'voltage': 'four million', 'state': 'bleedin\ demised', 'action': 'VOOM'}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' de
```

Hello

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello



- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

[Report a bug](#)

[Show source](#)

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An [if ... elif ... elif ...](#) sequence is a substitute for the [switch](#) or [case](#) statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [Match Statements](#).

4.2. for Statements

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as in C), Python’s [for](#) statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
```

```

del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The range() Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

for i in range(5):
    print(i)
# Output:
# 0
# 1
# 2
# 3
# 4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

list(range(5, 10))
# Output: [5, 6, 7, 8, 9]

list(range(0, 10, 3))
# Output: [0, 3, 6, 9]

list(range(-10, -100, -30))
# Output: [-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```

a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
# Output:
# 0 Mary
# 1 had
# 2 a
# 3 little
# 4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

range(10)
# Output: range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](#) statement is such a construct, while an example of a function that takes an iterable is [sum\(\)](#):

```
sum(range(4))  # 0 + 1 + 2 + 3
# Output: 6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about [list\(\)](#).

4.4. break and continue Statements

The [break](#) statement breaks out of the innermost enclosing [for](#) or [while](#) loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break
# Output:
# 4 equals 2 * 2
# 6 equals 2 * 3
# 8 equals 2 * 4
# 9 equals 3 * 3
```

The [continue](#) statement continues with the next iteration of the loop:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")
# Output:
# Found an even number 2
# Found an odd number 3
# Found an even number 4
# Found an odd number 5
# Found an even number 6
# Found an odd number 7
# Found an even number 8
# Found an odd number 9
```

4.5. else Clauses on Loops

In a [for](#) or [while](#) loop, the [break](#) statement may be paired with an [else](#) clause. If the loop finishes without executing the [break](#), the [else](#) clause executes.

The [else](#) clause is executed after the loop finishes its final iteration, that is, if no break occurred. In a [while](#) loop, it's executed after the loop's condition becomes false.

In either case, the [else](#) clause is **not** executed if the loop was terminated by a [break](#). Other ways of ending the loop early, such as a [return](#) or a raised exception, will also skip execution of the [else](#) clause.

This is exemplified in the following [for](#) loop, which searches for prime numbers:

```
for n in range(2, 10):
    for x in range(2, n):
```

```

        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
        else:
            # loop fell through without finding a factor
            print(n, 'is a prime number')
# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2
# 5 is a prime number
# 6 equals 2 * 3
# 7 is a prime number
# 8 equals 2 * 4
# 9 equals 3 * 3

```

4.6. pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```

while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)

```

This is commonly used for creating minimal classes:

```

class MyEmptyClass:
    pass

```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```

def initlog(*args):
    pass # Remember to implement this!

```

4.7. match Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```

def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"

```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y)=point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

You can use positional parameters with some built-in classes that provide an ordering for their attributes (e.g., dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to (`"x"`, `"y"`), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what.

Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (`foo.bar`), attribute names (`x=` , `y=` above) or class names (recognized by the “(…)” next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 2)]
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print("Not on the diagonal")
```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (but `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
    ...
```

This code captures the second element of a sequence as `p2` .

- Most literals are compared by equality; however, the singleton `True` , `False` and `None` are compared by identity.

- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variables:

```
from enum import Enum

class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For more detailed explanations and additional examples, see [PEP 636](#).

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
def fib(n):
    """Print a Fibonacci series less than n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()

# Now call the function we just defined:
fib(2000)

# Output:
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function [definition](#). It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or [docstring](#). There are tools which use docstrings to automatically produce documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The [execution](#) of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in enclosing functions, then in the global table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a [global](#) statement, or, for variables of enclosing functions, named in a [nonlocal](#) statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; arguments are passed using *call by object reference* (see [Note 1](#)). When a function calls another or itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can be used to access the function:

```
>>> fib
<function fib at 0x7f8e4dcd0f70>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In Python, functions without a `return` statement do return a value, namely `None`. Writing the value `None` is normally suppressed by the interpreter if it is the only value to be displayed. You can see it explicitly:

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of Fibonacci numbers instead of printing:

```
def fib2(n):
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a) # see below
        a, b = b, a + b
    return result

f100 = fib2(100) # call it
# write the result
print(f100)
# Output:
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example demonstrates some Python features:

- The `return` statement returns a value from a function. `return` without an argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a `method` of the list object `result`. A method is a function belonging to an object, called with dot notation: `obj.methodname`. Different object types define different methods; methods of different types may have the same name. In this case, `append()` adds an element to the list.

4.9. More on Defining Functions

Functions can have variable arguments. Three forms exist, which can be combined.

4.9.1. Default Argument Values

Default argument values are specified in the function definition and allow calling the function with fewer arguments than it is defined for:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
```

```

    if reply in {'y', 'ye', 'yes'}:
        return True
    if reply in {'n', 'no', 'nop', 'nope'}:
        return False
    retries -= 1
    if retries < 0:
        raise ValueError('invalid user response')
    print(reminder)

```

This function can be called as:

```

ask_ok('Do you really want to quit?') # Only mandatory argument
ask_ok('OK to overwrite the file?', 2) # One optional argument
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!') # All arguments

```

It also introduces the [in](#) keyword, which tests whether a sequence contains a value.

Default values are evaluated at function definition time, in the defining scope:

```

i = 5

def f(arg=i):
    print(arg)

i = 6
f() # Prints 5

```

Important: The default value is evaluated only once. This is significant when the default is a mutable object like a list, dict, or class instance:

```

def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
# Output:
# [1]
# [1, 2]
# [1, 2, 3]

```

To avoid sharing the default, write instead:

```

def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

```

4.9.2. Keyword Arguments

Functions can be called with explicit keyword arguments:

```

def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")

```

```
print("-- Lovely plumage, the", type)
print("-- It's", state, "!")
```

Calls:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # positional + keyword
```

Invalid calls:

```
parrot() # missing required argument
parrot(voltage=5.0, 'dead') # non-keyword argument after keyword argument
parrot(110, voltage=220) # duplicate argument value
parrot(actor='John Cleese') # unknown keyword argument
```

Note: In calls, keyword arguments follow positional arguments, and must match the parameter names, but their order doesn't matter. Arguments cannot be assigned more than once.

4.9.3. Special parameters

Functions may specify parameters as positional-only, keyword-only, or both, using `/` and `*` in the parameter list.

4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are absent, arguments can be passed either way:

```
def standard_arg(arg):
    print(arg)

standard_arg(2)
standard_arg(arg=2)
```

4.9.3.2. Positional-Only Parameters

Parameters before `/` are positional-only:

```
def pos_only_arg(arg, /):
    print(arg)

pos_only_arg(1)
pos_only_arg(arg=1) # Error
```

4.9.3.3. Keyword-Only Arguments

Parameters after `*` are keyword-only:

```
def kwd_only_arg(*, arg):
    print(arg)

kwd_only_arg(arg=3)
kwd_only_arg(3) # Error
```

4.9.3.4. Function Examples


```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

Examples of calling:

```
standard_arg(2)
pos_only_arg(1)
# pos_only_arg(arg=1) # Error
kwd_only_arg(arg=3)
combined_example(1, 2, kwd_only=3)
# combined_example(pos_only=1, standard=2, kwd_only=3) # Error for pos_only
```

4.9.4. Arbitrary Argument Lists

Functions can accept arbitrary arguments with `*args` :

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

called as:

```
def concat(*args, sep='/'):
    return sep.join(args)

concat("earth", "mars", "venus")
# Output: 'earth/mars/venus'

concat("earth", "mars", "venus", sep='.')
# Output: 'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists

Arguments in a list or tuple can be unpacked into separate parameters:

```
list(range(3, 6))
# Output: [3, 4, 5]

args = [3, 6]
list(range(*args))
# Output: [3, 4, 5]
```

Similarly, dictionaries can be unpacked into keyword arguments:

```
def parrot(voltage, state='a stiff', action='vroom'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.", end=' ')
    print("E's", state, "!")
```

```
d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
parrot(**d)
```

4.9.6. Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. It returns the sum of its two arguments:

```
lambda a, b: a + b
```

Lambda functions are syntactic sugar for normal functions, restricted to a single expression, possibly referencing variables from the containing scope:

```
def make_incrementor(n):
    return lambda x: x + n
```

```
f = make_incrementor(42)
print(f(0))
# Output: 42
print(f(1))
# Output: 43
```

They can be used where function objects are required, e.g., in list sorting:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
# Sorted based on second element
```

4.9.7. Documentation Strings

Collection of conventions around docstrings, using a brief summary, blank line, and descriptive paragraphs. The parser does not strip indentation, so tools processing docstrings usually do. Example:

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass

print(my_function.__doc__)
# Output:
# Do nothing, but document it.
#
# No, really, it doesn't do anything.
```

4.9.8. Function Annotations

Annotations are optional metadata about function parameters and return value. They are stored in the `__annotations__` attribute:

```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs
```

```
f('spam')
# Output:
# Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
# Arguments: spam eggs
# 'spam and eggs'
```

4.10. Intermezzo: Coding Style

(A brief guide on Python coding style based on PEP 8, including indentation, line length, comments, naming conventions, and more.)

4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. `if` Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the [switch](#) or `case` statements found in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [match Statements](#).

4.2. `for` Statements

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
# Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))

# Output:
# cat 3
# window 6
# defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy
for user, status in users.copy().items():
```

```

    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status

```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
...     print(i)
# Output:
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
# Output:
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the

list, thus saving space.

We say such an object is **iterable**, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the **for** statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

4.4. **break** and **continue** Statements

The **break** statement breaks out of the innermost enclosing `for` or `while` loop:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} equals {x} * {n//x}")
            break

# Output:
# 4 equals 2 * 2
# 6 equals 2 * 3
# 8 equals 2 * 4
# 9 equals 3 * 3
```

The **continue** statement continues with the next iteration of the loop:

```
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number {num}")

# Output:
# Found an even number 2
# Found an odd number 3
# Found an even number 4
# Found an odd number 5
# Found an even number 6
# Found an odd number 7
# Found an even number 8
# Found an odd number 9
```

4.5. **else** Clauses on Loops

In a `for` or `while` loop the **break** statement may be paired with an **else** clause. If the loop finishes without executing the **break**, the **else** clause executes.

In a `for` loop, the **else** clause is executed after the loop finishes its final iteration, that is, if no **break** occurred.

In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the **else** clause is **not** executed if the loop was terminated by a **break**. Of course, other ways of ending the loop early, such as a **return** or a raised exception, will also skip execution of the

`else` clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
        else:
            # loop fell through without finding a factor
            print(n, 'is a prime number')
# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2
# 5 is a prime number
# 6 equals 2 * 3
# 7 is a prime number
# 8 equals 2 * 4
# 9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, not the `if` statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [Handling Exceptions](#).

4.6. `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

This is commonly used for creating minimal classes:

```
class MyEmptyClass:
    pass
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
def initlog(*args):
    pass # Remember to implement this!
```

4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
```



```

        print("Origin")
    case Point(x=0, y=y):
        print(f"Y={y}")
    case Point(x=x, y=0):
        print(f"X={x}")
    case Point():
        print("Somewhere else")
    case _:
        print("Not a point")

```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to ("x", "y"), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the "(...)" next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```

class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")

```

We can add an `if` clause to a pattern, known as a "guard". If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")

```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
    ...
```

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'
```

```
color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))
```

```
match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#) which is written in a tutorial format.

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
def fib(n):
    """Print a Fibonacci series less than n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()
# Now call the function we just defined:
fib(2000)
# Output:
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a `global` statement, or, for variables of enclosing functions, named in a `nonlocal` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by object reference* (where the *value* is always an object *reference*, not the value of the object). [1] When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 0x10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
def fib2(n):
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a) # see below
        a, b = b, a + b
    return result

f100 = fib2(100) # call it
# write the result
f100 # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be

an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see [Classes](#)) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
```python
def ask_ok(prompt, retries=4, reminder='Please try again!'):
 while True:
 reply = input(prompt)
 if reply in {'y', 'ye', 'yes'}:
 return True
 if reply in {'n', 'no', 'nop', 'nope'}:
 return False
 retries = retries - 1
 if retries < 0:
 raise ValueError('invalid user response')
 print(reminder)
```

This function can be called in several ways:

- Giving only the mandatory argument:

```
ask_ok('Do you really want to quit?')
```

- Giving one of the optional arguments:

```
ask_ok('OK to overwrite the file?', 2)
```

- Or even giving all arguments:

```
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that:

```
i = 5
def f(arg=i):
 print(arg)
i = 6
f()
Outputs: 5
```

*Important warning:* The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
 L.append(a)
 return L
```

```
print(f(1))
print(f(2))
print(f(3))
Outputs:
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
 if L is None:
 L = []
 L.append(a)
 return L
```

## 4.9.2. Keyword Arguments

Functions can also be called using [keyword arguments](#) of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.")
 print("-- Lovely plumage, the", type)
 print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments in different order
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot() # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g., `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g., `parrot(voltage=1000)` is valid too). No argument may receive a value more than once.

Here's an example that fails due to this restriction:

```
def function(a):
 pass
```

```
function(0, a=0)
Error:
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`). For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
 print("-- Do you have any", kind, "?")
 print("-- I'm sorry, we're all out of", kind)
 for arg in arguments:
 print(arg)
 print("-" * 40)
 for kw in keywords:
 print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger",
 "It's very runny, sir.",
 "It's really very, VERY runny, sir.",
 shopkeeper="Michael Palin",
 client="John Cleese",
 sketch="Cheese Shop Sketch")
```

and it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.

shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

## 4.9.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
 # -----
 # Positional or keyword # Keyword only
 # Positional only
```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as *named parameters*.

### 4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

### 4.9.3.2. Positional-Only Parameters

Looking at this in a bit more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a `/` (forward-slash). The `/` is used to logically separate the positional-only parameters from the rest of the parameters. If there is no `/` in the function definition, there are no positional-only parameters.

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

### 4.9.3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an `*` in the arguments list just before the first *keyword-only* parameter.

### 4.9.3.4. Function Examples

Consider the following example function definitions paying close attention to the markers `/` and `*`:

```
>>> def standard_arg(arg):
... print(arg)
>>> def pos_only_arg(arg, /):
... print(arg)
>>> def kwd_only_arg(*, arg):
... print(arg)
>>> def combined_example(pos_only, /, standard, *, kwd_only):
... print(pos_only, standard, kwd_only)
```

The first function definition, `standard_arg`, the most familiar form, places no restrictions on the calling convention and arguments may be passed by position or keyword:

```
>>> standard_arg(2)
2
```

```
>>> standard_arg(arg=2)
2
```

The second function `pos_only_arg` is restricted to only use positional parameters as there is a `/` in the function definition:

```
>>> pos_only_arg(1)
1
```

```
>>> pos_only_arg(arg=1)
Error:
TypeError: pos_only_arg() got some positional-only arguments passed as keyword argum
```

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition:

```
>>> kwd_only_arg(3)
Error:
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

And the last uses all three calling conventions in the same function definition:

```
>>> combined_example(1, 2, 3)
Error:
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Error:
TypeError: combined_example() got some positional-only arguments passed as keyword a
```

Finally, consider this function definition which has a potential collision between the positional argument `name` and `**kwargs` which has `name` as a key:

```
def foo(name, **kwargs):
 return 'name' in kwargs
```

There is no possible call that will make it return `True` as the keyword `'name'` will always bind to the first parameter. For example:

```
foo(1, **{'name': 2})
Error:
TypeError: foo() got multiple values for argument 'name'
```

But using `/` (positional only arguments), it is possible since it allows `name` as a positional argument and `'name'` as a key in the keyword arguments:

```
>>> def foo(name, /, **kwargs):
... return 'name' in kwargs
>>> foo(1, **{'name': 2})
True
```

In other words, the names of positional-only parameters can be used in `**kwargs` without ambiguity.

### 4.9.3.5. Recap

The use case will determine which parameters to use in the function definition:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

As guidance:

- Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
- Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being



passed.

- For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

## 4.9.4. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Before the variable number of arguments, zero or more normal arguments may occur:

```
def write_multiple_items(file, separator, *args):
 file.write(separator.join(args))
```

Normally, these *variadic* arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep='/'):
... return sep.join(args)
>>> concat('earth', 'mars', 'venus')
'earth/mars/venus'
>>> concat('earth', 'mars', 'venus', sep='.')
'earth.mars.venus'
```

## 4.9.5. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
... print("-- This parrot wouldn't", action, end=' ')
... print("if you put", voltage, "volts through it.")
>>> d = {'voltage': 'four million', 'state': 'bleedin' demised', 'action': 'VOOM'}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' de
```

## 4.9.6. Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):
... return lambda x: x + n
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a `lambda` expression to return a function. Another use is to pass a small function as an argument. For instance, `list.sort()` takes a sorting key function *key* which can be a lambda function:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
Now sorted:
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

## 4.9.7. Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
>>> def my_function():
... """Do nothing, but document it.
...
... No, really, it doesn't do anything.
... """
... pass
>>> print(my_function.__doc__)
Do nothing, but document it.
```

```
No, really, it doesn't do anything.
```

## 4.9.8. Function Annotations

[Function annotations](#) are completely optional metadata information about the types used by user-defined functions (see [PEP 3107](#) and [PEP 484](#) for more information).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
... print("Annotations:", f.__annotations__)
... print("Arguments:", ham, eggs)
... return ham + ' and ' + eggs
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

## 4.10. Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, [PEP 8](#) has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `UpperCamelCase` for classes and `lowercase_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see [A First Look at Classes](#) for more on classes and methods).
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

Footnotes:

[1] Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

# More Control Flow Tools — Python

## 3.13.5 documentation

### 4.1. `if` Statements¶

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
... x = 0
... print('Negative changed to zero')
... elif x == 0:
... print('Zero')
... elif x == 1:
... print('Single')
... else:
... print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for "else if" and helps avoid excessive indentation. An `if ... elif ...` sequence is a substitute for `switch` or `case` statements in other languages.

If comparing the same value to several constants, or checking for specific types or attributes, the `match` statement can be useful. For more details, see [match Statements](#).

### 4.2. `for` Statements¶

The `for` statement in Python iterates over the items of any sequence (list or string), in the order they appear:

```
Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
 print(w, len(w))
Output:
cat 3
window 6
defenestrate 12
```

Modifying a collection while iterating over it can be tricky; consider iterating over a copy or creating a new collection:

```
Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

Strategy: Iterate over a copy
for user, status in users.copy().items():
 if status == 'inactive':
 del users[user]
```

```
Strategy: Create a new collection
active_users = {}
for user, status in users.items():
 if status == 'active':
 active_users[user] = status
```

## 4.3. The `range()` Function ¶

If you need to iterate over a sequence of numbers, use `range()` :

```
for i in range(5):
 print(i)
Output:
0
1
2
3
4
```

`range()` generates arithmetic progressions; the end point is never included. It can start at another number, include a different step (even negative):

```
list(range(5, 10)) # [5, 6, 7, 8, 9]
list(range(0, 10, 3)) # [0, 3, 6, 9]
list(range(-10, -100, -30)) # [-10, -40, -70]
```

To iterate over sequence indices:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
 print(i, a[i])
Output:
0 Mary
1 had
2 a
3 little
4 lamb
```

It's often more convenient to use `enumerate()` for such cases.

## Iterable objects

`range()` objects are iterable but not lists. They save space compared to actual lists but behave like iterables:

```
sum(range(4)) # 0 + 1 + 2 + 3 = 6
```

## 4.4. `break` and `continue` Statements ¶

`break` exits the innermost enclosing loop:

```
for n in range(2, 10):
 for x in range(2, n):
 if n % x == 0:
 print(f"{n} equals {x} * {n//x}")
 break
```

```

else:
 # Loop fell through without finding a factor
 print(f"{n} is a prime number")

```

`continue` skips to the next iteration:

```

for num in range(2, 10):
 if num % 2 == 0:
 print(f"Found an even number {num}")
 continue
 print(f"Found an odd number {num}")

```

## 4.5. `else` Clauses on Loops¶

The `break` statement can be paired with an `else` clause; the `else` executes only if the loop completes normally (without `break`):

```

for n in range(2, 10):
 for x in range(2, n):
 if n % x == 0:
 print(f"{n} equals {x} * {n//x}")
 break
 else:
 print(f"{n} is a prime number")

```

The `else` element is similar to the `try / except else` clause, running if no exception occurs or no `break` is executed.

## 4.6. `pass` Statements¶

The `pass` statement does nothing; it's used where syntactically required but no action is needed:

```

while True:
 pass # Busy-wait for keyboard interrupt

```

It's often used for minimal classes:

```

class MyEmptyClass:
 pass

```

Also as a placeholder:

```

def initlog(*args):
 pass # Remember to implement this!

```

## 4.7. `match` Statements¶

`match` takes a value and compares it against patterns introduced as `case` blocks; similar to `switch / case` but with pattern matching capabilities.

Basic literal match:

```

def http_error(status):
 match status:
 case 400:
 return "Bad request"

```

```

case 404:
 return "Not found"
case 418:
 return "I'm a teapot"
case _:
 return "Something's wrong with the internet"

```

- The `_` acts as a wildcard.
- Patterns can include multiple literals separated by `|`:

```

case 401 | 403 | 404:
 return "Not allowed"

```

- Patterns can unpack data, bind variables, or match class attributes:

```

match point:
 case (0, 0):
 print("Origin")
 case (0, y):
 print(f"Y={y}")
 case (x, 0):
 print(f"X={x}")
 case (x, y):
 print(f"X={x}, Y={y}")
 case _:
 raise ValueError("Not a point")

```

Using class patterns:

```

class Point:
 def __init__(self, x, y):
 self.x = x
 self.y = y

def where_is(point):
 match point:
 case Point(0, 0):
 print("Origin")
 case Point(0, y):
 print(f"Y={y}")
 case Point(x, 0):
 print(f"X={x}")
 case Point(x, y):
 print(f"X={x}, Y={y}")
 case _:
 print("Not a point")

```

`match` patterns support nested and extended unpacking, mapping patterns, subpatterns, constants, and guards for complex data structures.

## 4.8. Defining Functions¶

To define a function:

```

def fib(n):
 """Print a Fibonacci series less than n."""
 a, b = 0, 1

```

```

while a < n:
 print(a, end=' ')
 a, b = b, a + b
print()

```

- The first string (if present) is a docstring.
- Variables inside are local.
- Arguments are passed by object reference.
- Functions are objects and can be assigned or referenced:

```

fib = fib
fib(100)

```

Functions without a `return` explicitly return `None`.

Examples with return values:

```

def fib2(n):
 """Return a list containing the Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result

```

`return` can optionally send back a value; if omitted, `None` is returned.

Methods like `list.append()` modify objects in place.

## 4.9. More on Defining Functions¶

### 4.9.1. Default Argument Values¶

Examples:

```

def ask_ok(prompt, retries=4, reminder='Please try again!'):
 while True:
 reply = input(prompt)
 if reply in {'y', 'ye', 'yes'}:
 return True
 if reply in {'n', 'no', 'nop', 'nope'}:
 return False
 retries -= 1
 if retries < 0:
 raise ValueError('invalid user response')
 print(reminder)

```

### 4.9.2. Keyword Arguments¶

Functions can be called with `kwarg=value` syntax to specify arguments explicitly:

```

def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.")
 print("-- Lovely plumage, the", type)
 print("-- It's", state, "!")

```



Calling:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot('a million', 'bereft of life', shopkeeper='Michael Palin', client='John Cleese')
```

Guards (if clauses):

```
match point:
 case Point(x, y) if x == y:
 print(f"Y=X at {x}")
 case Point(x, y):
 print("Not on the diagonal")
```

### 4.9.3. Special Parameters

Parameters can be marked as:

- **Positional-only** with `/`.
- **Keyword-only** with `*`.

Examples:

```
def standard_arg(arg):
 print(arg)

def pos_only_arg(arg, /):
 print(arg)

def kwd_only_arg(*, arg):
 print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
 print(pos_only, standard, kwd_only)
```

### 4.9.4. Arbitrary Argument Lists

`*args` captures extra positional arguments:

```
def write_multiple_items(file, separator, *args):
 file.write(separator.join(args))
```

`*kwargs` captures extra keyword arguments:

```
def parrot(voltage, state='a stiff', action='vroom'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.")
```

### 4.9.5. Unpacking Argument Lists

Unpack list/tuple for separate arguments:

```
args = [3, 6]
list(range(*args))
```

Unpack dict for keyword arguments:

```
d = {'voltage': 'four million', 'state': 'bleedin\'' demised'}
parrot(**d)
```

## 4.10. Intermezzo: Coding Style ¶

Follow [PEP 8](#) for readable Python code:

- Use 4 spaces, no tabs.
- Wrap lines >79 characters.
- Separate functions/classes with blank lines.
- Place comments on their own line.
- Use docstrings.
- Spaces around operators but not inside brackets.
- Name classes with `UpperCamelCase`, functions with `lowercase_with_underscores`.
- Use ASCII names for international compatibility.

## Footer

[Report a bug](#) | [Source](#)

© [Python Software Foundation](#), 2001-2025. Licensed under PSF License Version 2.

Examples licensed under the BSD License.

[Donate](#) | Last updated on Jul 05, 2025 (07:10 UTC).

## 4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

### 4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
... x = 0
... print('Negative changed to zero')
... elif x == 0:
... print('Zero')
... elif x == 1:
... print('Single')
... else:
... print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An [if ... elif ... elif ...](#) sequence is a substitute for the [switch](#) or [case](#) statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the [match](#) statement useful. For more details see [match Statements](#).

### 4.2. for Statements

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s [for](#) statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
 print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straightforward to loop over a copy of the collection or to create a new collection:

```
Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

Strategy: Iterate over a copy
for user, status in users.copy().items():
 if status == 'inactive':
```

```

del users[user]

Strategy: Create a new collection
active_users = {}
for user, status in users.items():
 if status == 'active':
 active_users[user] = status

```

## 4.3. The range() Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

>>> for i in range(5):
... print(i)
...
0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
... print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```

>>> range(10)
range(0, 10)

```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is [iterable](#), that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the [for](#) statement is such a construct, while an example of a function that takes an iterable is [sum\(\)](#):

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about [list\(\)](#).

## 4.4. break and continue Statements

The [break](#) statement breaks out of the innermost enclosing [for](#) or [while](#) loop:

```
>>> for n in range(2, 10):
... for x in range(2, n):
... if n % x == 0:
... print(f"{n} equals {x} * {n//x}")
... break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The [continue](#) statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
... if num % 2 == 0:
... print(f"Found an even number {num}")
... continue
... print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

## 4.5. else Clauses on Loops

In a [for](#) or [while](#) loop the [break](#) statement may be paired with an [else](#) clause. If the loop finishes without executing the [break](#), the [else](#) clause executes.

In a [for](#) loop, the [else](#) clause is executed after the loop finishes its final iteration, that is, if no break occurred.

In a [while](#) loop, it's executed after the loop's condition becomes false.

In either kind of loop, the [else](#) clause is *not* executed if the loop was terminated by a [break](#). Of course, other ways of ending the loop early, such as a [return](#) or a raised exception, will also skip execution of the [else](#) clause.

This is exemplified in the following for loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
... for x in range(2, n):
... if n % x == 0:
... print(n, 'equals', x, '*', n//x)
... break
... else:
... # loop fell through without finding a factor
... print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, *not* the `if` statement.)

One way to think of the `else` clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like `if/if/else`. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see [Handling Exceptions](#).

## 4.6. pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
... pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
... pass
...
```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
... pass # Remember to implement this!
...
```

## 4.7. match Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a `switch` statement in C, Java or JavaScript (and many other languages), but it's more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
 match status:
 case 400:
 return "Bad request"
 case 404:
 return "Not found"
 case 418:
 return "I'm a teapot"
 case _:
 return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
 return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
point is an (x, y) tuple
match point:
 case (0, 0):
 print("Origin")
 case (0, y):
 print(f"Y={y}")
 case (x, 0):
 print(f"X={x}")
 case (x, y):
 print(f"X={x}, Y={y}")
 case _:
 raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject ( `point` ). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
 def __init__(self, x, y):
 self.x = x
 self.y = y

def where_is(point):
 match point:
 case Point(x=0, y=0):
 print("Origin")
 case Point(x=0, y=y):
 print(f"Y={y}")
 case Point(x=x, y=0):
 print(f"X={x}")
 case Point():
 print("Not a point")
```

```

 print("Somewhere else")
 case _:
 print("Not a point")

```

You can use positional parameters with some built-in classes that provide an ordering for their attributes (e.g., dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `("x", "y")`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (like `x=`, `y=` above) or class names (recognized by the `"(...)"` next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```

class Point:
 __match_args__ = ('x', 'y')
 def __init__(self, x, y):
 self.x = x
 self.y = y

match points:
 case []:
 print("No points")
 case [Point(0, 0)]:
 print("The origin")
 case [Point(x, y)]:
 print(f"Single point {x}, {y}")
 case [Point(0, y1), Point(0, y2)]:
 print(f"Two on the Y axis at {y1}, {y2}")
 case _:
 print("Something else")

```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```

match point:
 case Point(x, y) if x == y:
 print(f"Y=X at {x}")
 case Point(x, y):
 print(f"Not on the diagonal")

```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings.
- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking



like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)

- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2):
 ...
```

will capture the second element of the input as `p2` (as long as the input is a sequence of two points)

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

```
from enum import Enum
class Color(Enum):
 RED = 'red'
 GREEN = 'green'
 BLUE = 'blue'
```

```
color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))
```

```
match color:
 case Color.RED:
 print("I see red!")
 case Color.GREEN:
 print("Grass is green")
 case Color.BLUE:
 print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into [PEP 636](#).

## 4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n): # write Fibonacci series less than n
... """Print a Fibonacci series less than n."""
... a, b = 0, 1
... while a < n:
... print(a, end=' ')
... a, b = b, a + b
... print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function [definition](#). It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or [docstring](#). (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The [execution](#) of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of [built-in names](#). Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a [global](#) statement, or, for variables of enclosing functions, named in a [nonlocal](#) statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using [call by object reference](#) (where the [reference](#) is always to an object, not a copy of the object). When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 0x104084bd0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a [return](#) statement do return a value, albeit a rather boring one. This value is called [None](#), which is a built-in name. Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using [print\(\)](#):

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
... """Return a list containing the Fibonacci series up to n."""
... result = []
... a, b = 0, 1
... while a < n:
... result.append(a) # see below
... a, b = b, a + b
... return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The [return](#) statement returns with a value from a function. [return](#) without an expression argument returns `[None]` ([../reference/variable-dtypes.html#none](#)). Falling off the end of a function also returns `[None]` ([../reference/variable-dtypes.html#none](#)).
- The statement `[result.append(a)]` calls a [method](#) of the list object `[result]`. A method is a function that 'belongs' to an object and is named `[obj.methodname]` ([#methods-and-attributes](#)), where `[obj]` ([#objects-and-classes](#)) is some object (this may be an expression), and `[methodname]` ([#methods-and-attributes](#)) is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and

methods, using [classes](#), see [Classes](#)). The method `[append()]` (`#methods-and-attributes`) shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `[result = result + [a]]`, but more efficient.

## 4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

### 4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
 while True:
 reply = input(prompt)
 if reply in {'y', 'ye', 'yes'}:
 return True
 if reply in {'n', 'no', 'nop', 'nope'}:
 return False
 retries = retries - 1
 if retries < 0:
 raise ValueError('invalid user response')
 print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument:

```
ask_ok('Do you really want to quit?')
```

- giving one of the optional arguments:

```
ask_ok('OK to overwrite the file?', 2)
```

- or even giving all arguments:

```
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

This example also introduces the [in](#) keyword. This tests whether a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that:

```
i = 5
```

```
def f(arg=i):
 print(arg)
```

```
i = 6
```

```
f()
```

will print `5`.

**Important warning:** The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
 L.append(a)
 return L
```

```
print(f(1))
print(f(2))
print(f(3))
```

This will print:

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
 if L is None:
 L = []
 L.append(a)
 return L
```

## 4.9.2. Keyword Arguments

Functions can also be called using [keyword arguments](#) of the form `[kwarg=value]`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.")
 print("-- Lovely plumage, the", type)
 print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='V00000M') # 2 keyword arguments
parrot(action='V00000M', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()
required argument missing

parrot(voltage=5.0, 'dead')
non-keyword argument after a keyword argument

parrot(110, voltage=220)
duplicate value for the same argument

parrot(actor='John Cleese')
unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g., `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g., `parrot(voltage=1000)` is valid too). No argument may receive a value more than once.

Here's an example that fails due to this restriction:

```
>>> def function(a):
... pass
...
>>> function(0, a=0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a [tuple](#) containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example:

```
def cheeseshop(kind, *arguments, **keywords):
 print("-- Do you have any", kind, "?")
 print("-- I'm sorry, we're all out of", kind)
 for arg in arguments:
 print(arg)
 print("-" * 40)
 for kw in keywords:
 print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
 "It's really very, VERY runny, sir.",
 shopkeeper="Michael Palin",
 client="John Cleese",
 sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.

shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

## 4.9.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. To restrict this, parameters can be marked with special symbols:

- `[ / ]` indicates positional-only parameters (parameters that can only be supplied positionally).

- `[*]` indicates keyword-only parameters (parameters that can only be supplied as keywords).

### 4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

### 4.9.3.2. Positional-Only Parameters

You can mark certain parameters as *positional-only* by placing a `/` in the parameter list before their position. These parameters can only be supplied positionally, not as keywords:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
 # Positional-only / ----- * Keyword-only
 # parameters
```

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*. If no `/` in the list, there are no positional-only parameters.

### 4.9.3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, place `*` in the arguments list just before the first keyword-only parameter:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
 ...
```

### 4.9.3.4. Function Examples

```
>>> def standard_arg(arg):
... print(arg)
...
>>> def pos_only_arg(arg, /):
... print(arg)
...
>>> def kwd_only_arg(*, arg):
... print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
... print(pos_only, standard, kwd_only)
...
>>> standard_arg(2)
2
>>> pos_only_arg(1)
1
>>> kwd_only_arg(arg=3)
3
>>> combined_example(1, 2, kwd_only=3)
1 2 3
```

## 4.9.4. Arbitrary Argument Lists

Finally, functions can be defined to accept an arbitrary number of positional arguments:

```
def write_multiple_items(file, separator, *args):
 file.write(separator.join(args))
```

Any formal parameters following `*args` are keyword-only. Example:

```
def concat(*args, sep='/'):
 return sep.join(args)

concat("earth", "mars", "venus") # 'earth/mars/venus'
concat("earth", "mars", "venus", sep='.') # 'earth.mars.venus'
```

## 4.9.5. Unpacking Argument Lists

Arguments stored in a list or tuple can be unpacked into separate positional arguments when calling a function:

```
>>> list(range(3, 6))
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))
[3, 4, 5]
```

Similarly, for keyword arguments:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
... print("-- This parrot wouldn't", action, end=' ')
... print("if you put", voltage, "volts through it.")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' de
```

## 4.9.6. Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments:

```
lambda a, b: a + b
```

Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. They can reference variables from the surrounding scope:

```
>>> def make_incrementor(n):
... return lambda x: x + n
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda to return a function. It's also used to pass small functions as arguments, such as sorting key functions:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
```

```
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

## 4.9.7. Documentation Strings

Here are some conventions about the content and formatting of documentation strings:

- The first line should always be a short, concise summary of the object's purpose. It should begin with a capital letter and end with a period. It should not explicitly state the object's name or type.
- If there are more lines, the second line should be blank, separating the summary from the rest. The following lines should be paragraphs describing calling conventions, side effects, etc.
- Indentation is preserved, but conventions recommend stripping common leading whitespace:

```
>>> def my_function():
... """Do nothing, but document it.
...
... No, really, it doesn't do anything.
... """
... pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.
```

No, really, it doesn't do anything.

## 4.9.8. Function Annotations

Function annotations are optional metadata about types used by functions:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
... print("Annotations:", f.__annotations__)
... print("Arguments:", ham, eggs)
... return ham + ' and ' + eggs
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

Annotations are stored in the `__annotations__` attribute of the function as a dictionary; they do not affect execution.

---

## 4.10. Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about [coding style](#). Most languages have different styles; some are more readable. Adopting a consistent style makes your code easier for others to read.

Important points:

- Use 4 spaces per indentation level; no tabs.
- Wrap lines so they don't exceed 79 characters.
- Separate functions and classes with blank lines.
- Put comments on their own lines when possible.



- Use docstrings for modules, functions, classes, and methods.
- Use spaces around operators and after commas.
- Name classes with `UpperCamelCase`, functions with `lowercase_with_underscores`.
- Always use `self` as the first argument to instance methods.
- Use UTF-8 encoding, avoid non-ASCII characters in identifiers unless necessary.

**Footnote:**

[1] Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes made to it by the callee (items inserted into a list).

## 4. More Control Flow Tools

As well as the [while](#) statement just introduced, Python uses a few more that we will encounter in this chapter.

### 4.1. if Statements

Perhaps the most well-known statement type is the [if](#) statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
... x = 0
... print('Negative changed to zero')
... elif x == 0:
... print('Zero')
... elif x == 1:
... print('Single')
... else:
... print('More')
...
More
```

There can be zero or more [elif](#) parts, and the [else](#) part is optional. The keyword ‘[elif](#)’ is short for ‘else if’, and is useful to avoid excessive indentation. An `[if](../reference/compound_stmts.html#if)` ... `[elif](../reference/compound_stmts.html#elif)` ... `[elif](../reference/compound_stmts.html#elif)` ... sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [match Statements](#).

### 4.2. for Statements

The [for](#) statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
Measure some strings:
words = ['cat', 'window', 'defenestrate']
for w in words:
 print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}
```

```
Strategy: Iterate over a copy
for user, status in users.copy().items():
 if status == 'inactive':
 del users[user]

Strategy: Create a new collection
active_users = {}
for user, status in users.items():
 if status == 'active':
 active_users[user] = status
```

## 4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
for i in range(5):
 print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
list(range(5, 10))
[5, 6, 7, 8, 9]

list(range(0, 10, 3))
[0, 3, 6, 9]

list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()`:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
 print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see [Looping Techniques](#).

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is *iterable*, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `[for](../reference/compound_stmts.html#for)` statement is such a construct, while an example of a function that takes an iterable is `sum()`:

```
sum(range(4)) # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter [Data Structures](#), we will discuss in more detail about `list()`.

## 4.4. break and continue Statements

The *break* statement breaks out of the innermost enclosing *for* or *while* loop:

```
for n in range(2, 10):
 for x in range(2, n):
 if n % x == 0:
 print(f"{n} equals {x} * {n//x}")
 break
 ...
2 equals 2 * 1
3 equals 3 * 1
4 equals 2 * 2
5 equals 5 * 1
...
```

The *continue* statement continues with the next iteration of the loop:

```
for num in range(2, 10):
 if num % 2 == 0:
 print(f"Found an even number {num}")
 continue
 print(f"Found an odd number {num}")
 ...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

## 4.5. else Clauses on Loops

In a `for` or `while` loop the `break` statement may be paired with an `else` clause. If the loop finishes without executing the `break`, the `else` clause executes.

In a `for` loop, the `else` clause is executed after the loop finishes its final iteration, that is, if no `break` occurred. In a `while` loop, it's executed after the loop's condition becomes false.

In either kind of loop, the `else` clause is *not* executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following `for` loop, which searches for prime numbers:

```
for n in range(2, 10):
 for x in range(2, n):
 if n % x == 0:
 print(n, 'equals', x, '*', n//x)
 break
 else:
 # loop fell through without finding a factor
 print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

## Explanation

The `else` clause belongs to the `for` loop, **not** to the `if` statement. When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs.

---

## 4.6. `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action:

```
while True:
 pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

This is commonly used for creating minimal classes:

```
class MyEmptyClass:
 pass
```

Another place `pass` can be used is as a placeholder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
def initlog(*args):
 pass # Remember to implement this!
```

---

## 4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript, but it's more similar to

pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed, and it can also extract components (sequence elements or object attributes) from the value into variables.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
 match status:
 case 400:
 return "Bad request"
 case 404:
 return "Not found"
 case 418:
 return "I'm a teapot"
 case _:
 return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match. If no case matches, none of the branches is executed.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
 return "Not allowed"
```

Patterns can look like unpacking assignments and can be used to bind variables:

```
point is an (x, y) tuple
match point:
 case (0, 0):
 print("Origin")
 case (0, y):
 print(f"Y={y}")
 case (x, 0):
 print(f"X={x}")
 case (x, y):
 print(f"X={x}, Y={y}")
 case _:
 raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data, you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
 def __init__(self, x, y):
 self.x = x
 self.y = y

def where_is(point):
 match point:
 case Point(x=0, y=0):
 print("Origin")
 case Point(x=0, y=y):
 print(f"Y={y}")
```

```

case Point(x=x, y=0):
 print(f"X={x}")
case Point():
 print("Somewhere else")
case _:
 print("Not a point")

```

You can use positional parameters with some built-in classes that provide an ordering for their attributes (e.g., dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `("x", "y")`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```

Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)

```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what.

Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the `(...)` next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```

class Point:
 __match_args__ = ('x', 'y')
 def __init__(self, x, y):
 self.x = x
 self.y = y

points = [Point(0, 0), Point(1, 2), Point(3, 4)]
match points:
 case []:
 print("No points")
 case [Point(0, 0)]:
 print("The origin")
 case [Point(x, y)]:
 print(f"Single point {x}, {y}")
 case [Point(0, y1), Point(0, y2)]:
 print(f"Two on the Y axis at {y1}, {y2}")
 case _:
 print("Something else")

```

We can add an `[if](../reference/compound_stmts.html#match)` guard to a pattern. If the guard is false, `match` proceeds to the next case block. Note that value capture happens *before* the guard is evaluated:

```

match point:
 case Point(x, y) if x == y:
 print(f"Y=X at {x}")
 case Point(x, y):
 print("Not on the diagonal")

```

## Additional features:

- Tuple and list patterns with the same meaning, matching arbitrary sequences. `[...]` and `(...)` can include extended unpacking: `[x, y, *rest]` or `(x, y, *rest)`. The name after `*` can be `_` to ignore remaining items.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures those from a dictionary. Extra keys are ignored. `**rest` pattern also supported (but `**_` is redundant).
- Subpatterns with `as`: `case (Point(x1, y1), Point(x2, y2) as p2): ...`
- Literals generally compared by equality; `True`, `False`, and `None` compare by identity.
- Patterns may use named constants with dotted names, e.g.:

```
from enum import Enum
class Color(Enum):
 RED = 'red'
 GREEN = 'green'
 BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))
match color:
 case Color.RED:
 print("I see red!")
 case Color.GREEN:
 print("Grass is green")
 case Color.BLUE:
 print("I'm feeling the blues :(")
```

For more details, see [PEP 636](#).

## 4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
def fib(n):
 """Print a Fibonacci series less than n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()
```

Now call the function just defined:

```
fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body start at the next line and are indented.

The first statement of the function body can optionally be a string literal; this string is the function's documentation string, or *docstring*. More about docstrings can be found in [Documentation Strings](#). There are tools which use docstrings for documentation or interactive browsing, so it's good practice to include docstrings.

The execution of a function creates a new symbol table for local variables. All variable assignments in the function go into this local scope; references first look there, then in enclosing functions, then in the global scope, and finally in built-ins. Variables of enclosing functions cannot be assigned directly (unless declared `nonlocal`) and global variables need an explicit `global` statement.



Arguments are passed *by object reference*. When a function calls another, a new local symbol table is created. A function object is associated with its name, but can also be assigned to other names:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

A function without an explicit `return` statement returns a built-in `None`. Even functions with no explicit return statement return `None`, which is normally suppressed in output, but can be printed:

```
>>> fib(0)
None
>>> print(fib(0))
None
```

You can also write functions that return data, such as a list:

```
def fib2(n):
 """Return a list containing the Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a) # see below
 a, b = b, a + b
 return result
```

Calling `fib2(100)`:

```
f100 = fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

- The `return` statement exits a function and optionally returns a value.
- The method `append()` adds an element to a list and is a common way to build lists incrementally; it's more efficient than `result = result + [a]`.

---

## 4.9. More on Defining Functions

Arguments can be flexible, with defaults, variable args, and keyword args.

### 4.9.1. Default Argument Values

Specify default values to create optional arguments:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
 while True:
 reply = input(prompt)
 if reply in {'y', 'ye', 'yes'}:
 return True
 if reply in {'n', 'no', 'nop', 'nope'}:
 return False
 retries -= 1
 if retries < 0:
 raise ValueError('invalid user response')
 print(reminder)
```

This function can be called with different combinations:

```
ask_ok('Do you really want to quit?') # only mandatory argument
ask_ok('OK to overwrite the file?', 2) # one optional argument
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!') # all arguments
```

Default argument values are evaluated at the point of function definition. Beware of mutable default arguments:

```
def f(a, L=[]):
 L.append(a)
 return L
```

```
print(f(1))
print(f(2))
print(f(3))
Output:
[1]
[1, 2]
[1, 2, 3]
```

To avoid shared mutable defaults, use:

```
def f(a, L=None):
 if L is None:
 L = []
 L.append(a)
 return L
```

## 4.9.2. Keyword Arguments

Functions accept **keyword arguments**:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.")
 print("-- Lovely plumage, the", type)
 print("-- It's", state, "!")
```

Call with keywords:

```
parrot(1000)
parrot(voltage=1000)
parrot(voltage=1000000, action='V00000M')
parrot(action='V00000M', voltage=1000000)
parrot('a million', 'bereft of life', 'jump')
parrot('a thousand', state='pushing up the daisies')
```

Invalid calls:

```
parrot() # missing required argument
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword
parrot(110, voltage=220) # duplicate argument
parrot(actor='John Cleese') # unknown argument
```

In a call, keyword arguments follow positional arguments, and all must match parameters of the function.

## 4.9.3. Special parameters

#### 4.9.3.1. Positional-only arguments

Marked with `/` in function definition:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
 ...
```

Arguments before `/` are positional-only and cannot be passed by keyword.

#### 4.9.3.2. Positional-only

Use `/` to enforce positional-only arguments.

#### 4.9.3.3. Keyword-only arguments

Use `*` before argument list:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
 ...
```

Arguments after `*` are keyword-only.

#### 4.9.3.4. Function examples

```
def standard_arg(arg):
 print(arg)
```

```
def pos_only_arg(arg, /):
 print(arg)
```

```
def kwd_only_arg(*, arg):
 print(arg)
```

```
def combined_example(pos_only, /, standard, *, kwd_only):
 print(pos_only, standard, kwd_only)
```

Call examples and restrictions are shown in the documentation.

#### 4.9.3.5. Recap

Parameter options depend on the symbols `/` and `*`. Use:

- positional-only with `/`
- keyword-only with `*`
- or neither for unrestricted.

---

## 4.9.4. Arbitrary Argument Lists

Functions can accept an arbitrary number of positional arguments:

```
def write_multiple_items(file, separator, *args):
 file.write(separator.join(args))
```

`*args` collects extra positional arguments into a tuple, which can be used as:

```
def concat(*args, sep='/'):
 return sep.join(args)
```

```
concat('earth', 'mars', 'venus') # 'earth/mars/venus'
concat('earth', 'mars', 'venus', sep='.') # 'earth.mars.venus'
```

---

## 4.9.5. Unpacking Argument Lists

Unpack lists/tuples in function calls:

```
args = [3, 6]
list(range(*args))
[3, 4, 5]

d = {'voltage': 'four million', 'state': 'bleedin\ demised'}
parrot(**d)
-- This parrot wouldn't VROOM if you put four million volts through it. E's bleedin'
```

---

## 4.9.6. Lambda Expressions

Small anonymous functions:

```
lambda a, b: a + b
```

Example of returning a function:

```
def make_incrementor(n):
 return lambda x: x + n
f = make_incrementor(42)
```

```
f(0) # 42
f(1) # 43
```

Use in `list.sort()`:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
[('4', 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

---

## 4.9.7. Documentation Strings

Docstrings start with a short summary. Followed optionally by blank line and detailed description. Indentation is preserved. Conventionally:

```
def my_function():
 """Do nothing, but document it.

 No, really, it doesn't do anything.
 """
 pass
```

The first line begins with a capital letter and ends with a period. Use triple quotes.

---

## 4.9.8. Function Annotations

Optional metadata about parameter and return types, stored in `__annotations__`. Example:

```
def f(ham: str, eggs: str = 'eggs') -> str:
 """Example with annotations."""
```

```
print("Annotations:", f.__annotations__)
print("Arguments:", ham, eggs)
return ham + ' and ' + eggs
```

---

## 4.10. Intermezzo: Coding Style

Guidelines derived from [PEP 8](#) suggest:

- Use 4 spaces indent, no tabs.
  - Wrap lines at 79 characters.
  - Use blank lines to separate functions/classes and blocks.
  - Prefer comments on own lines.
  - Use docstrings.
  - Space around operators and after commas: `a = f(1, 2) + g(3, 4)`
  - Class names `UpperCamelCase`; function/method names `lowercase_with_underscores`.
  - Always use `self` as the first method argument.
  - Use ASCII; avoid non-ASCII in identifiers.
- 

## Navigation

- [Index](#)
- [Modules](#)
- [Next: Data Structures](#)
- [Previous: An Informal Introduction to Python](#)

## Additional Links

- [Report a bug](#)
- [Show source](#)
- [Python official website](#)

© [Python Software Foundation](#), 2001-2025

Page licensed under PSF License Version 2.

Examples and recipes are also licensed under the Zero Clause BSD License.

[More info](#)

Please donate: [DS\\_PSF Donations](#)

Last updated on Jul 05, 2025 (07:10 UTC)

Created using [Sphinx](#) 8.2.3.

## 4. More Control Flow Tools¶

### 4.1. `if` Statements¶

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
... x = 0
... print('Negative changed to zero')
... elif x == 0:
... print('Zero')
... elif x == 1:
... print('Single')
... else:
... print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `'elif'` is short for 'else if', and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the `switch` or `case` statements found in other languages.

If you're comparing the same value to several constants, or checking for specific types or attributes, you may also find the `match` statement useful. For more details see [match Statements](#).

### 4.2. `for` Statements¶

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Instead of iterating over a sequence of numbers or defining steps explicitly, Python's `for` loops iterate over the items of any sequence, in order:

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
... print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Modifying a collection while iterating over it can be tricky. Instead, iterate over a copy or create a new collection:

```
Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

Strategy: Iterate over a copy
for user, status in users.copy().items():
 if status == 'inactive':
 del users[user]

Strategy: Create a new collection
```

```

active_users = {}
for user, status in users.items():
 if status == 'active':
 active_users[user] = status

```

## 4.3. The `range()` Function ¶

Use `range()` to generate sequences of numbers:

```

>>> for i in range(5):
... print(i)
...
0
1
2
3
4

```

The end point isn't included. You can specify start, end, and step:

```

>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(-10, -100, -30))
[-10, -40, -70]

```

To iterate over sequence indices:

```

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
... print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

A more convenient method is to use `enumerate()`.

The object returned by `range` behaves as if it is a list, but it's more space-efficient as it is an iterable.

```

>>> sum(range(4)) # 0 + 1 + 2 + 3
6

```

## 4.4. `break` and `continue` Statements ¶

- The `break` statement exits the innermost loop.
- The `continue` statement skips to the next iteration.

Example of `break`:

```

>>> for n in range(2, 10):
... for x in range(2, n):
... if n % x == 0:
... print(f"{n} equals {x} * {n//x}")
... break

```

```

... else:
... # loop fell through without finding a factor
... print(n, "is a prime number")
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3

```

Example of continue:

```

>>> for num in range(2, 10):
... if num % 2 == 0:
... print(f"Found an even number {num}")
... continue
... print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9

```

## 4.5. `else` Clauses on Loops

An optional `else` can be attached to `for` or `while` loops. The `else` executes only if the loop wasn't terminated by a `break`:

- In a `for` loop, after completing all iterations without `break`.
- In a `while` loop, after the condition becomes false and no `break` occurred.

Example searching for primes, with `else`:

```

for n in range(2, 10):
 for x in range(2, n):
 if n % x == 0:
 print(n, 'equals', x, '*', n//x)
 break
 else:
 # loop fell through without finding a factor
 print(n, 'is a prime number')

```

The `else` runs only if no `break` was triggered (no factors found).

## 4.6. `pass` Statements

The `<code>pass</code>` statement does nothing but acts as a placeholder:

```

>>> while True:
... pass # Busy-wait for keyboard interrupt (Ctrl+C)

```



```
...
```

Useful for minimal classes:

```
>>> class MyEmptyClass:
... pass
...
```

And for empty functions or condition bodies:

```
>>> def initlog(*args):
... pass # Remember to implement this!
...
```

## 4.7. match Statements

Pattern matching: compare an expression to patterns (like switch-case in other languages). Executes the first matching pattern.

Example:

```
def http_error(status):
 match status:
 case 400:
 return "Bad request"
 case 404:
 return "Not found"
 case 418:
 return "I'm a teapot"
 case _:
 return "Something's wrong with the internet"
```

The wildcard `_` matches anything if no previous case matched.

Combining patterns with `|`:

```
case 401 | 403 | 404:
 return "Not allowed"
```

Pattern with unpacking:

```
match point:
 case (0, 0):
 print("Origin")
 case (0, y):
 print(f"Y={y}")
 case (x, 0):
 print(f"X={x}")
 case (x, y):
 print(f"X={x}, Y={y}")
 case _:
 raise ValueError("Not a point")
```

Patterns can involve classes:

```
class Point:
 def __init__(self, x, y):
 self.x = x
```

```

 self.y = y

def where_is(point):
 match point:
 case Point(x=0, y=0):
 print("Origin")
 case Point(x=0, y=y):
 print(f"Y={y}")
 case Point(x=x, y=0):
 print(f"X={x}")
 case Point(x=x, y=y):
 print(f"X={x}, Y={y}")
 case _:
 print("Not a point")

```

You can use `__match_args__` for positional class patterns.

## 4.8. Defining Functions ¶

Example: Fibonacci generator up to n:

```

def fib(n):
 """Print a Fibonacci series less than n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

```

Function components:

- `def` declares a function.
- String immediately after is a docstring.
- Variables in function are local.
- Arguments are passed by object-reference.

Function objects can be assigned to other names:

```

fib
f = fib
f(100)

```

Functions without an explicit return return `None`.

Example returning list:

```

def fib2(n):
 """Return a list containing the Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result

```

Call:

```
f100 = fib2(100)
f100 = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

## 4.9. More on Defining Functions ¶

### 4.9.1. Default Argument Values ¶

Define functions with optional arguments:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
 while True:
 reply = input(prompt)
 if reply in {'y', 'ye', 'yes'}:
 return True
 if reply in {'n', 'no', 'nop', 'nope'}:
 return False
 retries -= 1
 if retries < 0:
 raise ValueError('invalid user response')
 print(reminder)
```

Call examples:

- `ask_ok('Do you really want to quit?')`
- `ask_ok('OK to overwrite the file?', 2)`
- `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Note: default values are evaluated at function definition time; mutable defaults retain changes across calls, which can cause issues:

```
def f(a, L=[]):
 L.append(a)
 return L
```

Calling `f(1)`, `f(2)` results in `[1]`, `[1,2]` etc. To avoid this, use:

```
def f(a, L=None):
 if L is None:
 L = []
 L.append(a)
 return L
```

### 4.9.2. Keyword Arguments ¶

Functions accept arguments by position or keyword:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
 print("-- This parrot wouldn't", action, end=' ')
 print("if you put", voltage, "volts through it.")
 print("-- Lovely plumage, the", type)
 print("-- It's", state, "!")
```

Call examples:

- `parrot(1000)`
- `parrot(voltage=1000)`
- `parrot(voltage=1000000, action='V00000M')`

- `parrot('a million', 'bereft of life', 'jump')`
- `parrot('a thousand', state='pushing up the daisies')`

Invalid calls:

- Missing required argument: `parrot()`
- Unknown keyword argument: `parrot(actor='John Cleese')`
- Duplicate argument: `parrot(110, voltage=220)`

Argument order:

```
def function(a, **kwds):
 return 'name' in kwds
```

But this only works with non-overlapping names unless using positional-only parameters:

```
def foo(name, /, **kwds):
 return 'name' in kwds
```

### 4.9.3. Special parameters ¶

- `/` (positional-only)
- `*` (keyword-only)

Examples:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
 pass
```

- `pos1`, `pos2` are positional-only
- `pos_or_kwd` can be positional or keyword
- `kwd1`, `kwd2` are keyword-only

### 4.9.4. Arbitrary Argument Lists ¶

Use `*args` for variable positional arguments:

```
def write_multiple_items(file, separator, *args):
 file.write(separator.join(args))
```

Call with:

```
def concat(*args, sep="/"):
 return sep.join(args)
```

```
concat("earth", "mars", "venus") # 'earth/mars/venus'
concat("earth", "mars", "venus", sep=".")
```

### 4.9.5. Unpacking Argument Lists ¶

Unpack lists or tuples into function arguments:

```
args = [3, 6]
list(range(*args))
```

Unpack dictionaries into keyword arguments:

```
d = {"voltage": "four million", "state": "bleedin' demised"}
parrot(**d)
```

## 4.9.6. Lambda Expressions ¶

Create anonymous functions:

```
add = lambda a, b: a + b
print(add(2, 3)) # 5
```

Use in functions like `list.sort()`:

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
```

## 4.9.7. Documentation Strings ¶

Conventions:

- First line: concise summary, start uppercase, end with period.
- Followed by blank line and detailed description.
- Indentation is kept unless stripped manually.
- Example:

```
def my_function():
 """Do nothing, but document it."""
 pass
```

## 4.9.8. Function Annotations ¶

Optional metadata stored in `__annotations__`. Parameters can be annotated with types:

```
def f(ham: str, eggs: str = 'eggs') -> str:
 """Show annotations."""
 print("Annotations:", f.__annotations__)
 print("Arguments:", ham, eggs)
 return ham + ' and ' + eggs

f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

## 4.10. Intermezzo: Coding Style ¶

Most code should follow [PEP 8](#), which promotes readability via:

- 4 spaces indentation, no tabs.
- Lines no longer than 79 characters.
- Use blank lines for separation.
- Comments on own lines when possible.
- Use docstrings.

- Spaces around operators and after commas.
- Class names: `UpperCamelCase`.
- Function/method names: `lower_case_with_underscores`.
- Always use `self` as first argument in methods.
- Use UTF-8 or ASCII.
- Avoid non-ASCII characters in identifiers.

*Footnote:*

[1]: Actually, *call by object reference* would be a better description, since if a mutable object is passed, changes are visible to the caller (e.g., items inserted into a list).

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)`  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)`  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`  
Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.
- `list.pop([i])`  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.
- `list.clear()`  
Remove all items from the list. Similar to `del a[:]`.
- `list.index(x[, start[, end]])`  
Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- `list.count(x)`  
Return the number of times `x` appears in the list.
- `list.sort(*, key=None, reverse=False)`  
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).
- `list.reverse()`  
Reverse the elements of the list in place.
- `list.copy()`  
Return a shallow copy of the list. Similar to `a[:]`.

## Example Usage of List Methods

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.count('apple') # 2
```

```
fruits.count('tangerine') # 0
fruits.index('banana') # 3
fruits.index('banana', 4) # 6 (start searching from position 4)
fruits.reverse()
fruits.append('grape')
fruits.sort()
fruits.pop() # 'pear'
```

Note that methods like `insert`, `remove`, or `sort` that only modify the list have no return value printed—they return the default `None`. This design principle applies to all mutable data structures in Python.

---

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack is now [3, 4, 5, 6, 7]
stack.pop() # 7
stack # [3, 4, 5, 6]
stack.pop() # 6
stack.pop() # 5
stack is now [3, 4]
```

---

### 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first retrieved ("first-in, first-out"); however, this is inefficient because inserting or popping from the beginning of the list is slow.

To implement a queue efficiently, use `collections.deque`, designed for fast appends and pops from both ends:

```
from collections import deque

queue = deque(['Eric', 'John', 'Michael'])
queue.append('Terry') # Terry arrives
queue.append('Graham') # Graham arrives
queue.popleft() # 'Eric' departs
queue.popleft() # 'John' departs
Remaining queue:
deque(['Michael', 'Terry', 'Graham'])
```

---

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications include creating new lists where each element is the result of some operation applied to each member of another sequence, or selecting a subsequence based on a condition.

#### Examples:



- Creating a list of squares:

```
squares = [x**2 for x in range(10)]
or using map:
squares = list(map(lambda x: x**2, range(10)))
```

- Filtering to exclude negative numbers:

```
vec = [-4, -2, 0, 2, 4]
non_negative = [x for x in vec if x >= 0]
```

- Applying `abs()` to all elements:

```
abs_vec = [abs(x) for x in vec]
```

- Calling a method on each element:

```
freshfruit = [' banana', ' loganberry ', 'passion fruit ']
stripped = [fruit.strip() for fruit in freshfruit]
```

- Creating tuples of number and its square:

```
pairs = [(x, x**2) for x in range(6)]
Note: tuple parentheses are optional but often necessary to avoid syntax errors
```

- Flattening a list of lists:

```
vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for elem in vec for num in elem]
```

- Complex expressions with functions and nested functions:

```
from math import pi
pi_strs = [str(round(pi, i)) for i in range(1, 6)]
```

## 5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can itself be a list comprehension, enabling the creation of complex structures like matrix transpositions.

Example: Transposing a 3x4 matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]

transposed = [[row[i] for row in matrix] for i in range(4)]
or using zip
transposed = list(zip(*matrix))
```

The `zip()` function is the preferred and more efficient way to transpose matrices.

## 5.2. The `del` Statement

The `del` statement removes an item from a list given its index rather than its value, and can also delete slices of lists or entire variables.

Examples:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0] # remove first element
del a[2:4] # remove elements at indexes 2 and 3
del a[:] # clear entire list
del a # delete variable a
```

Referencing the variable `a` after deletion raises an error.

---

## 5.3. Tuples and Sequences

Lists and strings share properties like indexing and slicing, making them sequence types. Python also has the immutable `tuple` type.

Examples:

```
t = 12345, 54321, 'hello!'
t[0] # 12345
t = (12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
t[0] = 88888 # error: 'tuple' object does not support item assignment
v = ([1, 2, 3], [3, 2, 1]) # mutable objects inside tuples
```

Tuples can be nested, and creating a tuple with a single element requires a trailing comma:

```
empty = ()
singleton = 'hello', # Note the comma
len(empty) # 0
len(singleton) # 1
```

Sequence unpacking:

```
t = 1, 2, 3
x, y, z = t
```

---

## 5.4. Sets

Python includes a set data type: an unordered collection of unique elements supporting operations like union, intersection, difference, and symmetric difference.

Create sets with `{}` or `set()`:

```
basket = {'apple', 'orange', 'pear', 'banana'}
print(basket) # duplicates removed
print('orange' in basket) # membership test
a = set('abracadabra')
b = set('alacazam')
a - b # letters in a but not in b
a | b # letters in a or b or both
a & b # letters in both a and b
a ^ b # letters in a or b but not both
```

Set comprehensions:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
```

---

## 5.5. Dictionaries

Dictionaries are collections of key-value pairs, where keys are unique and immutable. Common operations:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
del tel['sape']
tel['irv'] = 4127
list(tel) # keys
sorted(tel) # sorted keys
'guido' in tel # key check
```

Create dictionaries with curly braces, the `dict()` constructor, or dictionary comprehensions:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{x: x**2 for x in (2, 4, 6)}
```

---

## 5.6. Looping Techniques

Using `.items()` to iterate over key-value pairs:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
```

Using `enumerate()` over a sequence:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
```

Zippping multiple sequences:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print(f'What is your {q}? It is {a}.')
```

Reversing sequences:

```
for i in reversed(range(1, 10, 2)):
 print(i)
```

Sorting:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
```

Looping over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
```

---

### 5.6.1. Filtering Data

Filtering data with a loop:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
```

---

## 5.7. More on Conditions

Conditions can contain any operators, including membership (`in`, `not in`), identity (`is`, `is not`), and logical operators (`and`, `or`, `not`). Multiple comparisons can be chained.

Boolean expressions can be assigned to variables:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3
```

Note: in Python, assignment inside expressions uses the walrus operator `:=`.

---

## 5.8. Comparing Sequences and Other Types

Sequence objects are compared lexicographically, similar to dictionary order: compare items one by one until mismatch, shorter sequences are smaller if all compared elements are equal.

Examples:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
(1, 2, 3, 4) < (1, 2)
(1, 2) == (1.0, 2.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Comparing objects of different types may raise `TypeError`, except for compatible numeric types.

---

*Footnote:*

1. Other languages may return the mutated object, allowing method chaining, e.g., `d->insert("a")->remove("b")->sort();`

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

- `list.extend(iterable)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a)] = iterable`.

- `list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

- `list.remove(x)`

Remove the first item from the list whose value is equal to `x`. It raises a [ValueError](#) if there is no such item.

- `list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item. It raises an [IndexError](#) if the list is empty or index is out of range.

- `list.clear()`

Remove all items from the list. Similar to `del a[:]`.

- `list.index(x[, start[, end]])`

Return zero-based index in the list of the first item whose value is equal to `x`. Raises a [ValueError](#) if no such item. The optional `start` and `end` limit the search to a subsequence.

- `list.count(x)`

Return the number of times `x` appears in the list.

- `list.sort(*, key=None, reverse=False)`

Sort the items of the list in place.

- `list.reverse()`

Reverse the elements of the list in place.

- `list.copy()`

Return a shallow copy of the list.

An example using most list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple')) # 2
print(fruits.count('tangerine')) # 0
print(fruits.index('banana')) # 3
print(fruits.index('banana', 4)) # 6
fruits.reverse()
print(fruits) # ['banana', 'apple', 'kiwi', 'banana', 'pear', 'a
fruits.append('grape')
print(fruits) # ['banana', 'apple', 'kiwi', 'banana', 'pear', 'a
fruits.sort()
print(fruits) # ['apple', 'apple', 'banana', 'banana', 'grape',
print(fruits.pop()) # 'pear'
```

Methods like `insert`, `remove`, or `sort` only modify the list and do not return a new list; they return `None`. This is a design principle for all mutable data structures in Python.

Not all data can be sorted or compared, e.g., `[None, 'hello', 10]` or `3+4j < 5+7j` are invalid comparisons.

### 5.1.1. Using Lists as Stacks

List methods make it easy to use a list as a stack with last-in, first-out behavior. Use `append()` to push, and `pop()` to retrieve the last item:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack) # [3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack) # [3, 4, 5, 6]
print(stack.pop()) # 6
print(stack.pop()) # 5
```

### 5.1.2. Using Lists as Queues

Lists can be used as queues (first-in, first-out), but they are inefficient for this purpose. Append/pop from the end are fast; insert/pop from the beginning are slow.

To implement a queue efficiently, use `collections.deque`:

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # The first to arrive now leaves ('Eric')
print(queue.popleft()) # The second to arrive now leaves ('John')
print(queue) # Remaining in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists:

```
squares = []
for x in range(10):
 squares.append(x**2)
print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Or, more concisely:

```
squares = list(map(lambda x: x**2, range(10)))
or
squares = [x**2 for x in range(10)]
```

They can contain complex expressions and nested functions, e.g.:

```
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

List comprehensions can include `if` conditions and multiple `for` clauses:

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

## 5.1.4. Nested List Comprehensions

They can contain other list comprehensions, e.g., to transpose a matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
Transpose rows and columns
transposed = [[row[i] for row in matrix] for i in range(4)]
Same as:
transposed = []
for i in range(4):
 transposed_row = []
 for row in matrix:
 transposed_row.append(row[i])
 transposed.append(transposed_row)
```

Better to use built-in functions like `zip()` for this:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

## 5.2. The `del` Statement

To remove an item by index:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a is now: [1, 66.25, 333, 333, 1234.5]
del a[2:4]
a is now: [1, 66.25, 1234.5]
del a[:]
a is now: []
```

You can delete entire variables:

```
del a
```

Referencing `a` after deletion causes an error.

## 5.3. Tuples and Sequences

Lists and strings are sequence types. Tuples are also sequences:

```
t = 12345, 54321, 'hello!'
print(t[0]) # 12345
u = (12345, 54321, 'hello!')
Tuples may be nested:
u = t, (1, 2, 3, 4, 5)
Tuples are immutable:
t[0] = 88888 # TypeError
But they can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
```

Empty tuples are `()`, a single item tuple is `('hello',)`. Example:

```
empty = ()
singleton = 'hello', # note trailing comma
len(empty) # 0
len(singleton) # 1
print(singleton) # ('hello',)
```

Sequence unpacking:

```
t = 1, 2, 3
x, y, z = t
```

Multiple assignment combines packing and unpacking.

## 5.4. Sets

Sets are unordered collections with no duplicates, useful for membership testing and eliminating duplicates:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
print('orange' in basket) # True
print('crabgrass' in basket) # False
a = set('abracadabra')
b = set('alacazam')
a - b # in a but not in b
a | b # in a or b
a & b # in both a and b
a ^ b # in a or b but not both
```

Set comprehensions:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
```

## 5.5. Dictionaries



Dictionaries are key-value stores with unique keys:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
del tel['sape']
tel['irv'] = 4127
list(tel) # list of keys: ['jack', 'guido', 'irv']
sorted(tel) # sorted keys
'guido' in tel # True
'jack' not in tel # False
```

Create dictionaries:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionary comprehensions:

```
{x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

With keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## 5.6. Looping Techniques

Looping through dictionaries:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
gallahad the pure
robin the brave
```

Looping through sequences with index:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
0 tic
1 tac
2 toe
```

Looping over multiple sequences:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
```

Reversing sequences:

```
for i in reversed(range(1, 10, 2)):
 print(i)
9
7
5
```

```
3
1
```

Sorting sequences:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
apple
apple
banana
orange
orange
pear
```

Loop over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
apple
banana
orange
pear
```

Filtering data with conditions:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
[56.2, 51.7, 55.3, 52.5, 47.8]
```

## 5.7. More on Conditions

Operators:

- `x in y` / `x not in y` for membership.
- `x is y` / `x is not y` for identity.
- Comparisons can be chained: `a < b == c`.

Logical operators:

- `and`, `or`, `not`, with short-circuit behavior.

Assign boolean expressions to variables:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3
'Trondheim'
```

Note: in Python, assignment inside expressions requires the `:=` operator (walrus operator).

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically, element by element:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2)
(1, 2) == (1.0, 2.0, 3.0) # False due to length mismatch
(1, 2, 3) < (1, 2, (-1,))
```

Comparing objects of different types is allowed if appropriate comparison methods exist; otherwise, a `TypeError` is raised.

---

*Footnotes and external links omitted for brevity.*

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)`  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)`  
Insert an item at a given position. The first argument is the index before which to insert, so `a.insert(0, x)` inserts at the front, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`  
Remove the first item from the list whose value is equal to `x`. Raises a `ValueError` if no such item exists.
- `list.pop([i])`  
Remove the item at the given position and return it. If no index is specified, `a.pop()` removes and returns the last item. Raises an `IndexError` if the list is empty or index is out of range.
- `list.clear()`  
Remove all items from the list. Similar to `del a[:]`.
- `list.index(x[, start[, end]])`  
Return the zero-based index of the first item whose value is equal to `x`. Raises a `ValueError` if no such item exists. Optional `start` and `end` limit the search.
- `list.count(x)`  
Return the number of times `x` appears in the list.
- `list.sort(*, key=None, reverse=False)`  
Sort the list in place using optional customization arguments.
- `list.reverse()`  
Reverse the list in place.
- `list.copy()`  
Return a shallow copy of the list.

### Example using most list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple')) # 2
print(fruits.count('tangerine')) # 0
print(fruits.index('banana')) # 3
print(fruits.index('banana', 4)) # 6
```

```

fruits.reverse()
print(fruits) # ['banana', 'apple', 'kiwi', 'banana', 'pear']
fruits.append('grape')
print(fruits) # ['banana', 'apple', 'kiwi', 'banana', 'pear']
fruits.sort()
print(fruits) # ['apple', 'apple', 'banana', 'banana', 'grap']
print(fruits.pop()) # 'pear'

```

Note: Methods like `insert`, `remove`, or `sort` modify the list in place and return `None`. This design principle applies to all mutable data structures in Python.

## Additional Notes:

- Not all objects can be sorted or compared; comparing incompatible types like `None` and strings raises `TypeError`.
  - Some types like complex numbers do not support comparison.
- 

### 5.1.1. Using Lists as Stacks

The list methods make it easy to use a list as a stack, where last added is first retrieved ( `LIFO` ). Use `append()` to add, and `pop()` to remove from the top. For example:

```

stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack) # [3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack) # [3, 4, 5, 6]
print(stack.pop()) # 6
print(stack) # [3, 4]

```

### 5.1.2. Using Lists as Queues

Lists can be used as queues ( `FIFO` ), but they are inefficient for this purpose because inserting or popping at the beginning is slow. Use `collections.deque` for efficient appends and pops from both ends:

```

from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # 'Eric'
print(queue.popleft()) # 'John'
print(queue) # deque(['Michael', 'Terry', 'Graham'])

```

### 5.1.3. List Comprehensions

List comprehensions create lists concisely:

```

squares = [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```

Without side effects:

```

squares = list(map(lambda x: x**2, range(10)))

```

Or:

```
squares = [x**2 for x in range(10)]
```

More complex examples:

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Nested list comprehensions:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
transposed = [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Using `zip()` to transpose:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

## 5.2. The `del` Statement

`del` removes an item from a list by index:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print(a) # [1, 66.25, 333, 333, 1234.5]
del a[2:4]
print(a) # [1, 66.25, 1234.5]
del a[:]
print(a) # []
```

`del` can also delete entire variables:

```
del a
```

---

## 5.3. Tuples and Sequences

Lists and strings share properties like indexing and slicing and are examples of **sequence** types. Tuples are another standard sequence type:

```
t = 12345, 54321, 'hello!'
print(t[0]) # 12345
print(t) # (12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
print(u) # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
Tuples are immutable:
t[0] = 88888 # Raises TypeError
v = ([1, 2, 3], [3, 2, 1])
Tuples may contain mutable objects:
print(v) # ([1, 2, 3], [3, 2, 1])
```

Tuple syntax quirks:

```

empty = () # Empty tuple
singleton = 'hello', # Tuple with one item (note comma)
len(empty) # 0
len(singleton) # 1
print(singleton) # ('hello',)

```

Tuple packing:

```

t = 12345, 54321, 'hello!'
x, y, z = t # Sequence unpacking

```

---

## 5.4. Sets

Sets are unordered collections with no duplicates:

```

basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
print('orange' in basket) # True
print('crabgrass' in basket) # False

Set operations:
a = set('abracadabra')
b = set('alacazam')

a # {'a', 'r', 'b', 'c', 'd'}
a - b # {'r', 'd', 'b'}
a | b # {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
a & b # {'a', 'c'}
a ^ b # {'r', 'd', 'b', 'm', 'z', 'l'}

```

Set comprehensions:

```

a = {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}

```

---

## 5.5. Dictionaries

A dictionary is a set of key-value pairs:

```

tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print(tel) # {'jack': 4098, 'sape': 4139, 'guido': 4127}
print(tel['jack']) # 4098
del tel['sape']
tel['irv'] = 4127
print(tel) # {'jack': 4098, 'guido': 4127, 'irv': 4127}
list(tel) # ['jack', 'guido', 'irv']
sorted(tel) # ['guido', 'irv', 'jack']
'guido' in tel # True
'jack' not in tel # False

```

Create from key-value pairs:

```

dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}

```

Dictionary comprehension:

```
{x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Using keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

---

## 5.6. Looping Techniques

Iterate over dictionary keys and values:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
gallahad the pure
robin the brave
```

Iterate over sequence with index:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
0 tic
1 tac
2 toe
```

Loop over multiple sequences:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}'.format(q, a))
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Loop in reverse:

```
for i in reversed(range(1, 10, 2)):
 print(i)
9
7
5
3
1
```

Sorted order:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
apple
apple
banana
orange
```



```
orange
pear
```

Loop over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
apple
banana
orange
pear
```

---

## 5.6.1 Filtering Data

Removing NaN values with `math.isnan`:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
[56.2, 51.7, 55.3, 52.5, 47.8]
```

---

## 5.7. More on Conditions

Operators:

- `in` / `not in` (membership tests)
- `is` / `is not` (identity tests)

Chained comparisons:

```
a < b == c
```

Boolean operators:

A and B or C

with short-circuit behavior, and negation with `not`.

Assigning boolean expressions:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3 # 'Trondheim'
```

Using the walrus operator:

```
if (n := len(some_list)) > 0:
 print(f"List has {n} elements")
```

## 5.8. Comparing Sequences and Other Types

Sequence objects can be compared lexicographically:

```
(1, 2, 3) < (1, 2, 4) # True
[1, 2, 3] < [1, 2, 4] # True
'ABC' < 'C' # True
'Pascal' < 'Python' # True
(1, 2, 3, 4) < (1, 2) # False
(1, 2) < (1, 2, -1) # True
(1, 2, 3) == (1.0, 2.0, 3.0) # True
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a')) # True
```

Comparisons between objects of different types are permissible if comparison methods exist. Otherwise, `TypeError` is raised.

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

- `list.extend(iterable)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a)] = iterable`.

- `list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

- `list.remove(x)`

Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.

- `list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.

- `list.clear()`

Remove all items from the list. Similar to `del a[:]`.

- `list.index(x, [start], [end])`

Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.

- `list.count(x)`

Return the number of times `x` appears in the list.

- `list.sort(*, key=None, reverse=False)`

Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

- `list.reverse()`

Reverse the elements of the list in place.

- `list.copy()`

Return a shallow copy of the list. Similar to `a[:]`.

## Example using list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.count('apple') # 2
fruits.count('tangerine') # 0
fruits.index('banana') # 3
fruits.index('banana', 4) # 6
fruits.reverse()
fruits is now ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
fruits.append('grape')
fruits is now ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
fruits.sort()
fruits is now ['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
fruits.pop() # 'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack is now [3, 4, 5, 6, 7]
stack.pop() # 7
stack is now [3, 4, 5, 6]
stack.pop() # 6
stack.pop() # 5
stack is now [3, 4]
```

### 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of the list are fast, doing inserts or pops from the beginning of a list is slow (because all the other elements have to be shifted by one).

To implement a queue, use `collections.deque`, which was designed to have fast appends and pops from both ends. For example:

```

from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
queue.popleft() # The first to arrive now leaves
'Eric'
queue.popleft() # The second to arrive now leaves
'John'
Remaining queue in order of arrival:
deque(['Michael', 'Terry', 'Graham'])

```

## 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, creating a list of squares:

```

squares = []
for x in range(10):
 squares.append(x**2)
print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, more concisely:

```
squares = [x**2 for x in range(10)]
```

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it.

## Examples:

- Combining elements of two lists if they are not equal:

```

[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

```

- Creating a filtered list and applying functions:

```

vec = [-4, -2, 0, 2, 4]
[x**2 for x in vec] # [-8, -4, 0, 4, 8]
[x for x in vec if x >= 0] # [0, 2, 4]
[abs(x) for x in vec] # [4, 2, 0, 2, 4]
freshfruit = [' banana', ' loganberry ', 'passion fruit ']
[weapon.strip() for weapon in freshfruit] # ['banana', 'loganberry', 'passion fruit']
List of tuples: number and its square
[(x, x**2) for x in range(6)] # [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
Error: parentheses are required around the tuple
[x, x**2 for x in range(6)] # SyntaxError
Flattening a 2D list

```

```
vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[num for elem in vec for num in elem] # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can also contain complex expressions and nested functions:

```
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## 5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider a 3x4 matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
]
```

Transposing rows and columns:

```
transposed = [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

This is equivalent to:

```
transposed = []
for i in range(4):
 transposed.append([row[i] for row in matrix])
```

which, in turn, can be written with nested loops:

```
transposed = []
for i in range(4):
 transposed_row = []
 for row in matrix:
 transposed_row.append(row[i])
 transposed.append(transposed_row)
```

Using the built-in `zip()` function is often preferred:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

## 5.2. The `del` Statement

The `del` statement can remove an item from a list given its index:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a is now [1, 66.25, 333, 333, 1234.5]
del a[2:4]
a is now [1, 66.25, 1234.5]
```

```
del a[:]
a is now []
```

`del` can also delete entire variables:

```
del a
```

After this, referencing `a` will raise an error until a new value is assigned.

## 5.3. Tuples and Sequences

Lists and strings have many properties in common, such as indexing and slicing. They are two examples of sequence data types (see [Sequence Types — list, tuple, range](#)).

A tuple consists of a number of values separated by commas:

```
t = 12345, 54321, 'hello!'
t[0] # 12345
t = (12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
u # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples are immutable:

```
t[0] = 88888 # Raises TypeError: 'tuple' object does not support item assignment
```

They can contain mutable objects:

```
v = ([1, 2, 3], [3, 2, 1])
v is ([1, 2, 3], [3, 2, 1])
```

Tuples are always enclosed in parentheses, especially when nested or in complex expressions. For a single-item tuple, a trailing comma is necessary:

```
empty = ()
singleton = 'hello', # Note trailing comma
len(empty) # 0
len(singleton) # 1
singleton # ('hello',)
```

Tuple packing:

```
t = 12345, 54321, 'hello!'
```

Sequence unpacking:

```
x, y, z = t
```

## 5.4. Sets

Python includes a data type for sets:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)
{'orange', 'banana', 'pear', 'apple'}
'orange' in basket # True
'crabgrass' in basket # False
```

Set operations:

```

a = set('abracadabra')
b = set('alacazam')
a # {'a', 'r', 'b', 'c', 'd'}
a - b # {'r', 'd', 'b'}
a | b # {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
a & b # {'a', 'c'}
a ^ b # {'r', 'd', 'b', 'm', 'z', 'l'}

```

Set comprehensions:

```

a = {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}

```

## 5.5. Dictionaries

Dictionaries are key-value mappings:

```

tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
del tel['sape']
tel['irv'] = 4127
list(tel) # ['jack', 'guido', 'irv']
sorted(tel) # ['guido', 'irv', 'jack']
'guido' in tel # True

```

Dictionary constructors:

```

dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
{ x: x**2 for x in (2, 4, 6) }
{2: 4, 4: 16, 6: 36}

```

Keyword arguments for dict:

```

dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}

```

## 5.6. Looping Techniques

Iterate over dictionary items:

```

knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
gallahad the pure
robin the brave

```

Enumerate over a sequence:

```

for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
0 tic
1 tac
2 toe

```

Zip multiple sequences:



```

questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.

```

Reverse a sequence:

```

for i in reversed(range(1, 10, 2)):
 print(i)
9
7
5
3
1

```

Sorted order:

```

basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
apple
apple
banana
orange
orange
pear

```

Loop over unique elements in sorted order:

```

for f in sorted(set(basket)):
 print(f)
apple
banana
orange
pear

```

Filtering data:

```

import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
filtered_data is [56.2, 51.7, 55.3, 52.5, 47.8]

```

## 5.7. More on Conditions

Operators:

- `in`, `not in` for membership testing.
- `is`, `is not` for identity comparison.
- Comparisons can be chained: `a < b == c`.
- Boolean logic: `and`, `or`, `not`.

- Short-circuit evaluation.

Example:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3
'Trondheim'
```

Note: In Python, assignment inside expressions uses the walrus operator `:=`.

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Comparisons between objects of incompatible types may raise `TypeError`, except for some common numeric types which are comparable across types.

[1] Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")->sort();`.

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)`  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)`  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`  
Remove the first item from the list whose value is equal to x. It raises a [ValueError](#) if there is no such item.
- `list.pop([i])`  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an [IndexError](#) if the list is empty or the index is outside the list range.
- `list.clear()`  
Remove all items from the list. Similar to `del a[:]`.
- `list.index(x[, start[, end]])`  
Return zero-based index in the list of the first item whose value is equal to x. Raises a [ValueError](#) if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- `list.count(x)`  
Return the number of times x appears in the list.
- `list.sort(*, key=None, reverse=False)`  
Sort the items of the list in place (the arguments can be used for sort customization, see [sorted\(\)](#) for their explanation).
- `list.reverse()`  
Reverse the elements of the list in place.
- `list.copy()`  
Return a shallow copy of the list. Similar to `a[:]`.

### Example of list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple')) # 2
```

```

print(fruits.count('tangerine')) # 0
print(fruits.index('banana')) # 3
print(fruits.index('banana', 4)) # 6
fruits.reverse()
print(fruits)
fruits.append('grape')
print(fruits)
fruits.sort()
print(fruits)
print(fruits.pop()) # 'pear'

```

Methods like `insert`, `remove`, or `sort` modify the list in place and return `None`, which is a design principle for mutable data structures in Python.

Note that not all data can be sorted or compared. For example, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings, and `None` cannot be compared to other types. Comparing complex objects like `3+4j` and `5+7j` with `<` or `>` is invalid.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top, use `pop()` without an explicit index. Example:

```

stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack) # [3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack) # [3, 4, 5, 6]
print(stack.pop()) # 6
print(stack) # [3, 4, 5]
print(stack.pop()) # 5
print(stack) # [3, 4]

```

### 5.1.2. Using Lists as Queues

Lists can also be used as queues, where the first element added is the first retrieved ("first-in, first-out"). However, this is inefficient because inserting or removing from the start of a list is slow.

For efficient queues, use `collections.deque`:

```

from collections import deque

queue = deque(['Eric', 'John', 'Michael'])

queue.append('Terry') # Terry arrives
queue.append('Graham') # Graham arrives
print(queue.popleft()) # The first to arrive now leaves ('Eric')
print(queue.popleft()) # The second to arrive now leaves ('John')
print(queue) # Remaining queue in order of arrival

```

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. They are useful for generating new lists by applying operations to each element of a sequence or filtering elements based on conditions.

Examples:

Create a list of squares:

```
squares = [x**2 for x in range(10)]
```

Create a list of doubled values:

```
vec = [-4, -2, 0, 2, 4]
doubled = [x * 2 for x in vec] # [-8, -4, 0, 4, 8]
```

Filter a list to exclude negative numbers:

```
non_neg = [x for x in vec if x >= 0] # [0, 2, 4]
```

Apply a function to all elements:

```
abs_values = [abs(x) for x in vec] # [4, 2, 0, 2, 4]
```

Call a method on each element:

```
freshfruit = [' banana', ' loganberry ', 'passion fruit ']
cleaned = [fruit.strip() for fruit in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

Create a list of tuples (number, square):

```
squares = [(x, x**2) for x in range(6)]
```

Note: Tuple parentheses are necessary when generating tuples in list comprehensions to avoid syntax errors:

```
[(x, x**2) for x in range(6)]
```

Parallel list comprehension to flatten a list of lists:

```
vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## 5.1.4. Nested List Comprehensions

List comprehensions can include other list comprehensions to process matrices or multi-dimensional data.

Transpose a matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]

transposed = [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

or, more Pythonically, using zip():

list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

## 5.2. The `del` statement

You can remove an item by its index using `del`:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a is now [1, 66.25, 333, 333, 1234.5]
del a[2:4]
a is now [1, 66.25, 1234.5]
del a[:]
a is now []
```

You can also delete entire variables:

```
del a
```

Reference to the variable after deletion will raise an error.

## 5.3. Tuples and Sequences

Lists and strings are sequence types, as are tuples. Tuples are immutable sequences of values separated by commas:

```
t = 12345, 54321, 'hello!'
print(t[0]) # 12345
print(t) # (12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
print(u) # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuples are immutable; attempting to modify elements raises a `TypeError`:

```
t[0] = 88888 # TypeError
```

Tuples can contain mutable objects like lists:

```
v = ([1, 2, 3], [3, 2, 1])
```

Empty tuples are created with `()`, a single-item tuple with a trailing comma:

```
empty = ()
singleton = 'hello!', # Note the trailing comma
len(empty) # 0
len(singleton) # 1
print(singleton) # ('hello!',)
```

Tuple packing and unpacking:

```
t = 12345, 54321, 'hello!'
x, y, z = t
```

## 5.4. Sets

Python includes a `set` type for unordered collections with no duplicates:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
```

```
print('orange' in basket) # True
print('crabgrass' in basket) # False
```

Set operations:

```
a = set('abracadabra')
b = set('alacazam')

a - b # elements in a but not in b
a | b # elements in either a or b or both
a & b # elements in both a and b
a ^ b # elements in either a or b but not both
```

Set comprehensions:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
```

## 5.5. Dictionaries

Dictionaries are key-value mappings, with keys being immutable types:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
del tel['sape']
tel['irv'] = 4127
print(tel) # {'jack': 4098, 'guido': 4127, 'irv': 4127}
print(list(tel)) # ['jack', 'guido', 'irv']
print(sorted(tel)) # ['guido', 'irv', 'jack']
print('guido' in tel) # True
print('jack' not in tel) # False
```

Dictionary construction:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionary comprehension:

```
{x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Using keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## 5.6. Looping Techniques

Iterate over keys and values:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
```

Iterate with index over a sequence:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
```

Loop over multiple sequences in parallel with `zip()`:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
```

Reverse a sequence:

```
for i in reversed(range(1, 10, 2)):
 print(i)
```

Sort a sequence while leaving the source unchanged:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(basket):
 print(f)
```

Loop over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
```

Filtering data:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
```

## 5.7. More on Conditions

Conditions may include any operators, including `in`, `not in`, `is`, `is not`.

Chained comparisons are possible:

```
a < b == c # b == c and a < b
```

Boolean logic:

```
non_null = string1 or string2 or string3
```

Logical operators:

```
if a and b or c:
 pass
```

Boolean operators short-circuit evaluation.

Assignment within expressions uses the walrus operator `:=`:

```
if (n := len(some_list)) > 10:
 print('List is long:', n)
```



## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Comparisons between objects of incompatible types will raise a `TypeError`. Mixing numeric types like `0` and `0.0` is allowed, with `0 == 0.0` evaluating to `True`.

### Footnote:

Other languages may allow method chaining with returning mutated objects, e.g., `d->insert("a")->remove("b")->sort();`

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- **append(x)**  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- **extend(iterable)**  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- **insert(i, x)**  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- **remove(x)**  
Remove the first item from the list whose value is equal to `x`. It raises a [ValueError](#) if there is no such item.
- **pop([i])**  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an [IndexError](#) if the list is empty or the index is outside the list range.
- **clear()**  
Remove all items from the list. Similar to `del a[:]`.
- **index(x[, start[, end]])**  
Return zero-based index in the list of the first item whose value is equal to `x`. Raises a [ValueError](#) if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence.
- **count(x)**  
Return the number of times `x` appears in the list.
- **sort(\*, key=None, reverse=False)**  
Sort the items of the list in place (the arguments can be used for sort customization, see [sorted\(\)](#) for their explanation).
- **reverse()**  
Reverse the elements of the list in place.
- **copy()**  
Return a shallow copy of the list. Similar to `a[:]`.

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
```

```

0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4) # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'

```

You might have noticed that methods like **insert**, **remove**, or **sort** that only modify the list have no return value printed—they return the default `[None](../library/exceptions.html#ValueError)`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```

>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack
[3, 4]
>>> stack.pop()
5
>>> stack
[3, 4]

```

### 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of the list are fast, doing inserts or pops from the beginning is slow because all the other elements have to be shifted by one.

To implement a queue efficiently, use `collections.deque`, which was designed for fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry") # Terry arrives
>>> queue.append("Graham") # Graham arrives
>>> queue.popleft() # The first to arrive now leaves
'Eric'
>>> queue.popleft() # The second to arrive now leaves
'John'
>>> queue # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares:

```
>>> squares = []
>>> for x in range(10):
... squares.append(x ** 2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares more concisely and without side effects using:

```
squares = list(map(lambda x: x ** 2, range(10)))
```

or, more readably:

```
squares = [x ** 2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this list comp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1, 2, 3]:
... for y in [3, 1, 4]:
... if x != y:
... combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the `for` and `if` statements is the same in both snippets.

If the expression is a tuple (e.g. `(x, y)` in the previous example), it must be parenthesized.

Similarly, list comprehensions can contain complex expressions and nested functions:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## 5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>> matrix = [
... [1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12]
...]
```

The following list comprehension will transpose rows and columns:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw earlier, the inner list comprehension is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
transposed = []
for i in range(4):
 transposed.append([row[i] for row in matrix])
```

and in turn is the same as:

```
transposed = []
for i in range(4):
 transposed_row = []
 for row in matrix:
 transposed_row.append(row[i])
 transposed.append(transposed_row)
```

In practice, you should prefer built-in functions to complex flow statements. The `zip()` function is perfect for this use case:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See [Unpacking Argument Lists](#) for details on the asterisk in that line.

## 5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (as earlier by assigning an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```
>>> del a
```

After this, referencing `a` is an error (until another value is assigned to it). We will see other uses for `del` later.

## 5.3. Tuples and Sequences

We saw that lists and strings have many properties in common, such as indexing and slicing operations. They are two examples of *sequence* data types (see [Sequence Types — list, tuple, range](#)). Since Python is evolving, other sequence types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t = (12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Output tuples are always enclosed in parentheses, which helps interpret nested tuples correctly; they may be input with or without parentheses, although parentheses are often necessary anyway. It is impossible to assign to individual items of a tuple, but tuples can contain mutable objects such as lists.

Although tuples may seem similar to lists, they are used differently. Tuples are *immutable*, often contain heterogeneous elements, and are accessed via unpacking or indexing. Lists are *mutable*, generally contain homogeneous elements, and are accessed via iteration.

Special syntax is required for constructing tuples with zero or one items: empty tuples use `()`, and a single-item tuple is created by following a value with a comma (not just parentheses):

```
>>> empty = ()
>>> singleton = 'hello', # note trailing comma
>>> len(empty)
0
```

```
>>> len singleton
1
>>> singleton
('hello',)
```

Tuple packing: the statement `t = 12345, 54321, 'hello!'` packs values into a tuple. The reverse — unpacking — works like this:

```
>>> x, y, z = t
```

Sequences must have the same number of variables on the left as there are items in the sequence.

## 5.4. Sets

Python includes a data type for *sets*, which are unordered collections with no duplicate elements. Use sets for membership testing and eliminating duplicates. Sets support mathematical operations like union (`|`), intersection (`&`), difference (`-`), and symmetric difference (`^`).

Create sets with curly braces or the `set()` function. To create an empty set, use `set()`, since `{}` creates an empty dictionary:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket) # duplicates removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket # membership testing
True
>>> 'crabgrass' in basket
False
```

Set operations example:

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b # in a but not in b
{'r', 'd', 'b'}
>>> a | b # in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b # in both a and b
{'a', 'c'}
>>> a ^ b # in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

## 5.5. Dictionaries

Python's *dictionary* (see [Mapping Types — dict](#)) is an associative array. Keys can be any immutable type, like strings or numbers. Tuples can be keys if they contain only immutable objects; mutable objects cannot.

Dictionaries are written in braces with `key: value` pairs:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Create dictionaries from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dictionary comprehensions:

```
>>> {x: x ** 2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Use keyword arguments for simple key-value pairs:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## 5.6. Looping Techniques

When looping through dictionaries, access key-value pairs with `.items()`:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
... print(k, v)
...
gallahad the pure
robin the brave
```

Loop over sequences with indices using `enumerate()`:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
... print(i, v)
...
0 tic
1 tac
2 toe
```

Loop over multiple sequences simultaneously with `zip()`:

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
```



```
>>> for q, a in zip(questions, answers):
... print('What is your {0}? It is {1}.'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Loop in reverse with `reversed()`:

```
>>> for i in reversed(range(1, 10, 2)):
... print(i)
...
9
7
5
3
1
```

Loop over sorted elements:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
... print(i)
...
apple
apple
banana
orange
orange
pear
```

Loop over unique sorted elements:

```
>>> for f in sorted(set(basket)):
... print(f)
...
apple
banana
orange
pear
```

## Removing entries during iteration

To filter data, create a new list instead of modifying a list while iterating over it. Example:

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
... if not math.isnan(value):
... filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

## 5.7. More on Conditions

Conditions can contain any operators, including `in`, `not in`, `is`, `is not`. Comparisons can be chained:

```
>>> a = 1
>>> b = 2
>>> c = 2
>>> a < b == c
False
```

Logical operators:

- `and`, `or`, `not`

Operators have specific precedence. Parentheses can be used to specify desired evaluation order.

Assignment inside expressions is possible via the *walrus operator* (`:=`):

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

## Boolean expressions as variables

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note: in Python, assignment within expressions uses the walrus operator:

```
if (m := len(s)) > 5:
 print(f"Long string of length {m}")
```

## 5.8. Comparing Sequences and Other Types

Sequences compare lexicographically: compare elements in order until a difference is found, or sequences are exhausted. Examples:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Comparisons between objects of different types are valid if the objects support comparison; otherwise, a `TypeError` is raised.

---

[1] Method `insert`, `remove`, or `sort` that modify the list return `None`; this is standard for mutable data structures in Python.

# Data Structures — Python 3.13.5 documentation

This chapter describes some things you’ve learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

- `list.extend(iterable)`

Extend the list by appending all the items from the iterable. Similar to `a[len(a)] = iterable`.

- `list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

- `list.remove(x)`

Remove the first item from the list whose value is equal to `x`. It raises a [ValueError](#) if there is no such item.

- `list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an [IndexError](#) if the list is empty or the index is outside the list range.

- `list.clear()`

Remove all items from the list. Similar to `del a[:]`.

- `list.index(x[, start[, end]])`

Return zero-based index in the list of the first item whose value is equal to `x`. Raises a [ValueError](#) if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.

- `list.count(x)`

Return the number of times `x` appears in the list.

- `list.sort(*, key=None, reverse=False)`

Sort the items of the list in place (the arguments can be used for sort customization, see [sorted\(\)](#) for their explanation).

- `list.reverse()`

Reverse the elements of the list in place.

- `list.copy()`

Return a shallow copy of the list. Similar to `a[:]`.

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4) # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
```

```
>>> stack
[3, 4]
```

### 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque`, which was designed to have fast appends and pops from both ends. For example:

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # The first to arrive now leaves
print(queue.popleft()) # The second to arrive now leaves
print(queue) # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares:

```
>>> squares = []
>>> for x in range(10):
... squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this list comp combines elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1, 2, 3]:
... for y in [3, 1, 4]:
... if x != y:
... combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the `for` and `if` statements is the same in both snippets.

If the expression is a tuple (e.g. `(x, y)` in the previous example), it must be parenthesized.

```
>>> vec = [-4, -2, 0, 2, 4]
create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1
 [x, x**2 for x in range(6)]
 ^^^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
```

To flatten a list using a list comprehension with two `for`:

```
>>> vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## 5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>> matrix = [
... [1, 2, 3, 4],
```

```
... [5, 6, 7, 8],
... [9, 10, 11, 12],
...]
```

The following list comprehension will transpose rows and columns:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the inner list comprehension is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
>>> transposed = []
>>> for i in range(4):
... transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
>>> transposed = []
>>> for i in range(4):
... # the following 3 lines implement the nested listcomp
... transposed_row = []
... for row in matrix:
... transposed_row.append(row[i])
... transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job for this use case:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See [Unpacking Argument Lists](#) for details on the asterisk in this line.

## 5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

[2] `del` can also be used to delete entire variables:

```
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We'll find other uses for `del` later.

## 5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types ([Sequence Types — list, tuple, range](#)). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t = (12345, 54321, 'hello!')
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> t[0] = 88888
TypeError: 'tuple' object does not support item assignment
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however, it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are [immutable](#), and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of [named tuples](#)). Lists are [mutable](#), and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello', # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```



This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

## 5.4. Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. To create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket) # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b # letters in both a and b
{'a', 'c'}
>>> a ^ b # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to [list comprehensions](#), set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
```

## 5.5. Dictionaries

Another useful data type built into Python is the *dictionary* ([Mapping Types — dict](#)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of

key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## 5.6. Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
... print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
... print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function:

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
... print('What is your {0}? It is {1}'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function:

```
>>> for i in reversed(range(1, 10, 2)):
... print(i)
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
... print(i)
...
apple
apple
banana
orange
orange
pear
```

Using `set()` on a sequence eliminates duplicate elements. The use of `sorted()` in combination with `set()` over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
... print(f)
...
apple
banana
orange
pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
... if not math.isnan(value):
... filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

## 5.7. More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The operators `in` and `not in` are membership tests to determine whether a value is in (or not in) a container. The operators `is` and `is not` compare whether two objects are the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. Parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the [walrus operator](#) `:=`. This avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

## 5.8. Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ, this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial subsequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters.

Some examples of comparisons between sequences of the same type:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with `<` or `>` is legal provided that the objects have appropriate comparison methods. Otherwise, the interpreter will raise a `TypeError` exception.

---

[1] Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")->sort();`.

[2] `del` can also be used to delete entire variables.

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)` : Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)` : Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)` : Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)` : Remove the first item from the list whose value is equal to x. It raises a [ValueError](#) if there is no such item.
- `list.pop([i])` : Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an [IndexError](#) if the list is empty or the index is outside the list range.
- `list.clear()` : Remove all items from the list. Similar to `del a[:]`.
- `list.index(x, [start[, end]])` : Return zero-based index in the list of the first item whose value is equal to x. Raises [ValueError](#) if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- `list.count(x)` : Return the number of times x appears in the list.
- `list.sort(*, key=None, reverse=False)` : Sort the items of the list in place (the arguments can be used for sort customization, see [sorted\(\)](#) for their explanation).
- `list.reverse()` : Reverse the elements of the list in place.
- `list.copy()` : Return a shallow copy of the list. Similar to `a[:]`.

### Example of List Methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple')) # 2
print(fruits.count('tangerine')) # 0
print(fruits.index('banana')) # 3
print(fruits.index('banana', 4)) # 6 # Find next banana starting at position 4
fruits.reverse()
print(fruits)
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
fruits.append('grape')
print(fruits)
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
fruits.sort()
print(fruits)
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
print(fruits.pop()) # 'pear'
```

You might have noticed that methods like `insert`, `remove`, or `sort` that only modify the list have no return value printed – they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings and `None` can't be compared to other types. Also, there are some types that don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't a valid comparison.

## Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack)
[3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack)
[3, 4, 5, 6]
print(stack.pop()) # 6
print(stack.pop()) # 5
print(stack)
[3, 4]
```

## Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow because all of the other elements have to be shifted by one.

To implement a queue, use `collections.deque`, which was designed to have fast appends and pops from both ends. For example:

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # The first to arrive now leaves
Eric
print(queue.popleft()) # The second to arrive now leaves
John
print(queue)
deque(['Michael', 'Terry', 'Graham'])
```

## List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, creating a list of squares:

```
squares = []
for x in range(10):
 squares.append(x ** 2)
```

```
print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This can be written more concisely as:

```
squares = list(map(lambda x: x ** 2, range(10)))
or
squares = [x ** 2 for x in range(10)]
```

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result is a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it.

Examples:

- Combine elements of two lists if they are not equal:

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- Generate tuples with their squares:

```
[(x, x ** 2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

- More complex expressions and nested functions:

```
from math import pi
[Str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Example: Transposing a matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
transposed = [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Alternatively, using `zip()`:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

## 5.2. The `del` Statement

The `del` statement provides a way to remove an item from a list given its index:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print(a) # [1, 66.25, 333, 333, 1234.5]
del a[2:4]
```



```
print(a) # [1, 66.25, 1234.5]
del a[:]
print(a) # []
```

`del` can also be used to delete entire variables:

```
del a
```

Referencing `a` afterwards will raise an error unless it is assigned again.

## 5.3. Tuples and Sequences

Lists and strings are sequence data types. The `<tuple>` is another standard sequence type.

Tuples are immutable sequences:

```
t = 12345, 54321, 'hello!'
print(t[0]) # 12345
print(t) # (12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
print(u) # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
Tuples are immutable:
t[0] = 88888 # raises TypeError
Tuples can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
print(v) # ([1, 2, 3], [3, 2, 1])
```

To create an empty tuple: `()`

To create a singleton tuple: `'hello',` (note trailing comma)

Tuple packing:

```
t = 12345, 54321, 'hello!'
x, y, z = t # sequence unpacking
```

## 5.4. Sets

Python supports sets:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
print('orange' in basket) # True
print('crabgrass' in basket) # False
```

```
a = set('abracadabra')
b = set('alacazam')
print(a) # {'a', 'r', 'b', 'c', 'd'}
print(a - b) # {'r', 'd', 'b'}
print(a | b) # {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
print(a & b) # {'a', 'c'}
print(a ^ b) # {'r', 'd', 'b', 'm', 'z', 'l'}
```

Set comprehensions:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
print(a) # {'r', 'd'}
```

## 5.5. Dictionaries

Dictionaries are mapping types:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print(tel) # {'jack': 4098, 'sape': 4139, 'guido': 4127}
print(tel['jack']) # 4098
del tel['sape']
tel['irv'] = 4127
print(tel) # {'jack': 4098, 'guido': 4127, 'irv': 4127}
print(list(tel)) # ['jack', 'guido', 'irv']
print(sorted(tel)) # ['guido', 'irv', 'jack']
print('guido' in tel) # True
print('jack' not in tel) # False
```

Creating dictionaries:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dict comprehensions:

```
{x: x ** 2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Using keyword arguments:

```
dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

## 5.6. Looping Techniques

- To retrieve key and value:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
gallahad the pure
robin the brave
```

- To retrieve index and value in a sequence:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
0 tic
1 tac
2 toe
```

- To iterate over multiple sequences simultaneously:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

- To iterate in reverse order:

```
for i in reversed(range(1, 10, 2)):
 print(i)
9
7
5
3
1
```

- To iterate over sorted elements:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
apple
apple
banana
orange
orange
pear
```

- To iterate over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
apple
banana
orange
pear
```

- Filtering data:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
print(filtered_data) # [56.2, 51.7, 55.3, 52.5, 47.8]
```

## 5.7. More on Conditions

Comparison operators support chaining:

```
a < b == c
```

Boolean operators:

```
a and b or c
not a
```

Operations like:

```
string1 = '', string2 = 'Trondheim', string3 = 'Hammer Dance'
non_null = string1 or string2 or string3
```

Note: in Python, assignment inside expressions uses the walrus operator `:`.`=`.

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Comparisons between objects of different types with `<` or `>` are only valid if they have appropriate comparison methods; otherwise, a [TypeError](#) is raised.

## 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

### 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- **append(x)**: Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- **extend(iterable)**: Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- **insert(i, x)**: Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- **remove(x)**: Remove the first item from the list whose value is equal to `x`. It raises a [ValueError](#) if there is no such item.
- **pop([i])**: Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an [IndexError](#) if the list is empty or the index is outside the list range.
- **clear()**: Remove all items from the list. Similar to `del a[:]`.
- **index(x[, start[, end]])**: Return zero-based index in the list of the first item whose value is equal to `x`. Raises a [ValueError](#) if there is no such item. The optional `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- **count(x)**: Return the number of times `x` appears in the list.
- **sort(\*, key=None, reverse=False)**: Sort the items of the list in place (the arguments can be used for sort customization, see [sorted\(\)](#) for their explanation).
- **reverse()**: Reverse the elements of the list in place.
- **copy()**: Return a shallow copy of the list. Similar to `a[:]`.

#### Example Usage

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple')) # 2
print(fruits.index('banana')) # 3
print(fruits.index('banana', 4)) # 6
fruits.reverse()
fruits.append('grape')
fruits.sort()
last_item = fruits.pop()
```

Methods like `insert`, `remove`, or `sort` that only modify the list have no return value printed—they return `None`. This is a design principle for all mutable data structures in Python.

Note that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings, and `None` can't be compared to other types. Also, comparisons like `3+4j < 5+7j` aren't valid.

#### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top, use `pop()` without an explicit index.

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack) # [3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack) # [3, 4, 5, 6]
```

## 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first retrieved (“first-in, first-out”); however, this is inefficient. While appends and pops from the end are fast, inserting or popping from the beginning is slow because all other elements must be shifted.

For efficiency, use `collections.deque`, which supports fast appends and pops from both ends:

```
from collections import deque

queue = deque(['Eric', 'John', 'Michael'])
queue.append('Terry') # Terry arrives
queue.append('Graham') # Graham arrives
print(queue.popleft()) # The first to arrive now leaves ('Eric')
print(queue.popleft()) # The second to arrive now leaves ('John')
print(queue) # Remaining in order of arrival
```

## 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications include creating new lists from operations on existing sequences or creating subsequences based on conditions.

### Example 1: Creating Squares

```
squares = [x**2 for x in range(10)]
Equivalent to:
squares = list(map(lambda x: x**2, range(10)))
```

### Example 2: Complex List Comprehensions

```
Combining elements from two lists if they are not equal
combs = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]

Creating a list of doubled values, filtering, and applying functions
vec = [-4, -2, 0, 2, 4]
doubled = [x*2 for x in vec]
non_neg = [x for x in vec if x >= 0]
abs_values = [abs(x) for x in vec]
stripped = [weapon.strip() for weapon in [' banana', ' loganberry ', 'passion fruit']]
tuples = [(x, x**2) for x in range(6)]

Creating a list of lists (flattening)
```

```
vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for elem in vec for num in elem]
```

## Example: Using math functions

```
from math import pi
rounded_pi = [str(round(pi, i)) for i in range(1, 6)]
```

## 5.1.4. Nested List Comprehensions

The initial expression can be any arbitrary expression, including another list comprehension. For example, transpose a matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]

Transpose
transposed = [[row[i] for row in matrix] for i in range(4)]
Equivalent to:
transposed = []
for i in range(4):
 transposed.append([row[i] for row in matrix])
Or more explicitly:
transposed_rows = []
for i in range(4):
 transposed_row = []
 for row in matrix:
 transposed_row.append(row[i])
 transposed_rows.append(transposed_row)
```

A recommended way is to use `zip()`:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

This is more concise and efficient.

## 5.2. The `del` Statement

You can remove an item from a list by index using `del`:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0] # removes -1
del a[2:4] # removes elements at index 2 and 3
del a[:] # clears the list
```

You can also delete entire variables:

```
del a
```

Referencing `a` after this will result in an error.

## 5.3. Tuples and Sequences

Lists and strings share many properties such as indexing and slicing, but are mutable sequences. Tuples are similar but immutable.

Examples:

```
t = 12345, 54321, 'hello!'
print(t[0]) # 12345
u = t, (1, 2, 3, 4, 5) # Tuple of tuples
Tuples are immutable: t[0] = 88888 # raises TypeError
v = ([1, 2, 3], [3, 2, 1]) # Mutable objects inside tuples
```

Tuples can be nested, and empty or singleton tuples have special syntax:

```
empty = ()
singleton = 'hello', # note trailing comma
len(empty) # 0
len(singleton) # 1
```

Tuple packing and unpacking:

```
t = 12345, 54321, 'hello!'
x, y, z = t
```

## 5.4. Sets

Python includes a set data type: an unordered collection with no duplicates.

Examples:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
```

```
'orange' in basket # True
'crabgrass' in basket # False
```

```
a = set('abracadabra')
b = set('alacazam')
a - b # elements in a but not in b
a | b # union
a & b # intersection
a ^ b # symmetric difference
```

Set comprehensions are supported:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
```

## 5.5. Dictionaries

Dictionaries are key-value stores. Keys can be any immutable type: strings, numbers, tuples (with only immutable contents).

Examples:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
```



```
del tel['sape']
tel['irv'] = 4127
list(tel) # list of keys
sorted(tel) # sorted list of keys
'guido' in tel # True
```

Constructors and comprehensions:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{ x: x**2 for x in (2,4,6) }
dict(sape=4139, guido=4127, jack=4098)
```

## 5.6. Looping Techniques

Iterate over dictionary items:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
```

Iterate over sequences with index:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
```

Loop over multiple sequences:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
```

Reversed order:

```
for i in reversed(range(1, 10, 2)):
 print(i)
```

Sorted sequence:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
```

Loop over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
```

Filtering data:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
```

## 5.7. More on Conditions

Operators in `while` and `if` conditions can include `in`, `not in`, `is`, `is not`. Comparisons can be chained:

```
a < b == c # equivalent to a < b and b == c
```

Boolean expressions can be combined with `and`, `or`, and negated with `not`. They are short-circuit operators:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3 # results in 'Trondheim'
```

Assignment within expressions requires the walrus operator `:=`:

```
if (n := len(some_list)) > 10:
 print(f"List is too long ({n} elements)")
```

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2)
(1, 2) == (1.0, 2.0, 3.0) # false, different lengths or types
```

Comparison between objects of different types may raise `TypeError` unless explicitly supported by the classes.

---

*Page includes navigation links and licensing info.*

# Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)`  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)`  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`  
Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.
- `list.pop([i])`  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.
- `list.clear()`  
Remove all items from the list. Similar to `del a[:]`.
- `list.index(x[, start[, end]])`  
Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- `list.count(x)`  
Return the number of times `x` appears in the list.
- `list.sort(*, key=None, reverse=False)`  
Sort the items of the list in place (the arguments can be used for sort customization, see [sorted\(\)](#) for their explanation).
- `list.reverse()`  
Reverse the elements of the list in place.
- `list.copy()`  
Return a shallow copy of the list. Similar to `a[:]`.

### Example:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
print(fruits.count('apple')) # 2
```

```

print(fruits.index('banana')) # 3
print(fruits.index('banana', 4)) # 6; find next banana starting at position 4
fruits.reverse()
print(fruits)
fruits.append('grape')
print(fruits)
fruits.sort()
print(fruits)
print(fruits.pop()) # 'pear'

```

Methods like `insert`, `remove`, or `sort` that only modify the list have no return value printed—they return `None`. This is a design principle for all mutable data structures in Python.

Note: Not all objects can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings, and `None` can't be compared to other types. Also, comparing objects of different types with `<` or `>` may raise a `TypeError`. For example, `3+4j < 5+7j` isn't a valid comparison.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index.

#### Example:

```

stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack) # [3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack) # [3, 4, 5, 6]
print(stack.pop()) # 6
print(stack.pop()) # 5
print(stack) # [3, 4]

```

### 5.1.2. Using Lists as Queues

It is possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of the list are fast, doing inserts or pops from the beginning of a list is slow because all other elements have to be shifted.

To implement a queue efficiently, use [collections.deque](#), which is designed for fast appends and pops from both ends.

#### Example:

```

from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # The first to arrive now leaves: Eric
print(queue.popleft()) # The second to arrive now leaves: John
print(queue) # Remaining queue: deque(['Michael', 'Terry', 'Graham'])

```

## 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of elements satisfying a condition.

### Example:

```
squares = []
for x in range(10):
 squares.append(x ** 2)
print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can achieve this more concisely using:

```
squares = list(map(lambda x: x ** 2, range(10)))
or
squares = [x ** 2 for x in range(10)]
```

### List comprehensions with conditions:

```
vec = [-4, -2, 0, 2, 4]
doubled = [x * 2 for x in vec] # create a new list with values d
non_negative = [x for x in vec if x >= 0] # filter the list to exclude negati
abs_list = [abs(x) for x in vec] # apply a function to all elements
stripped = [weapon.strip() for weapon in freshfruit] # call a method on each element
tuples = [(x, x ** 2) for x in range(6)] # create list of tuples
Note: parenthesis are needed if the element is a tuple inside comprehension
pair_list = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```

### Nested list comprehensions:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12],
]
transposed = [[row[i] for row in matrix] for i in range(4)]
Alternatively, use zip():
transposed = list(zip(*matrix))
```

Python's built-in `zip()` function is often preferable for such tasks.

## 5.2. The `del` statement

The `del` statement removes an item from a list by index, or slices, or deletes entire variables.

### Examples:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print(a) # [1, 66.25, 333, 333, 1234.5]
del a[2:4]
print(a) # [1, 66.25, 1234.5]
```

```
del a[:]
print(a) # []

del a # deletes the variable `a`
referencing `a` now raises NameError
```

## 5.3. Tuples and Sequences

Lists and strings have many properties in common, such as indexing and slicing. They are examples of sequence data types (see [Sequence Types — list, tuple, range](#)). Other sequence types can be added; one such type is the `tuple`.

A tuple is an ordered collection of values separated by commas:

```
t = 12345, 54321, 'hello!'
print(t[0]) # 12345
print(t) # (12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
print(u) # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
Tuples are immutable:
t[0] = 88888 # TypeError: 'tuple' object does not support item assignment
But they can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1])
print(v) # ([1, 2, 3], [3, 2, 1])
```

Note:

- Tuples are enclosed in parentheses for representing nested tuples or when needed to avoid syntax errors.
- Single-item tuples require a trailing comma:

```
empty = ()
singleton = 'hello', # note the comma
len(empty) # 0
len(singleton) # 1
```

- Tuple packing: `t = 12345, 54321, 'hello!'` packs values into a tuple.
- Sequence unpacking: `x, y, z = t` unpacks tuple into variables.

## 5.4. Sets

Python supports a data type called `set`. A set is an unordered collection with no duplicate elements, supporting operations like union, intersection, difference, and symmetric difference.

Create sets with braces or `set()`:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
'orange' in basket # True
'crabgrass' in basket # False
a = set('abracadabra')
b = set('alacazam')
a - b # letters in a but not in b
a | b # letters in a or b
a & b # letters in both a and b
a ^ b # letters in a or b but not both
```

Set comprehensions:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
```

## 5.5. Dictionaries

Dictionaries store key-value pairs (see [Mapping Types — dict](#)). Keys must be immutable types; strings, numbers, or tuples of immutable objects.

Basic usage:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
del tel['sape']
tel['irv'] = 4127
print(tel) # {'jack': 4098, 'guido': 4127, 'irv': 4127}
list(tel) # ['jack', 'guido', 'irv']
sorted(tel) # ['guido', 'irv', 'jack']
'guido' in tel # True
'tim' not in tel # True
```

Creating dictionaries:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
or
{x: x ** 2 for x in (2, 4, 6)}
or keyword arguments:
dict(sape=4139, guido=4127, jack=4098)
```

## 5.6. Looping Techniques

- Using `items()` to get key-value pairs:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
```

- Using `enumerate()` for index-value pairs:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
```

- Using `zip()` to iterate over multiple sequences:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
```

- Reversing a sequence:

```
for i in reversed(range(1, 10, 2)):
 print(i)
```

- Sorted iteration:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
```

- Using `set()` and `sorted()` to iterate over unique sorted elements:

```
for f in sorted(set(basket)):
 print(f)
```

- Filtering data:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
```

## 5.7. More on Conditions

Logical operators and membership tests:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3
```

Comparison operators:

- `in`, `not in` for membership testing
- `is`, `is not` for object identity

Chained comparisons:

```
a < b == c
```

Logical combinations:

```
if a < b and b == c or c:
 ...
```

Boolean operators `and`, `or` are short-circuit:

```
x and y # evaluates y only if x is true
```

and their results are the last evaluated argument when used as expressions.

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Objects of different types can be compared if comparison operators are properly defined; otherwise, a `TypeError` is raised.

---

*This page is licensed under the Python Software Foundation License Version 2.  
Examples, recipes, and other code are also licensed under the Zero Clause BSD License.*



See [History and License](#) for more information.  
[Please donate](#).

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)`  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)`  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`  
Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.
- `list.pop([i])`  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.
- `list.clear()`  
Remove all items from the list. Similar to `del a[:]`.
- `list.index(x[, start[, end]])`  
Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- `list.count(x)`  
Return the number of times `x` appears in the list.
- `list.sort(*, key=None, reverse=False)`  
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).
- `list.reverse()`  
Reverse the elements of the list in place.
- `list.copy()`  
Return a shallow copy of the list. Similar to `a[:]`.

### Example:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.count('apple') # 2
```

```

fruits.reverse()
fruits.append('grape')
fruits.sort()
print(fruits.pop()) # 'pear'

```

Methods like `insert()`, `remove()`, or `sort()` modify the list in place and return `None`. This design principle applies to all mutable data structures in Python.

Note: Not all data can be sorted or compared. For example, `[None, 'hello', 10]` cannot be sorted because integers can't be compared to strings, and `None` can't be compared to other types. Comparing complex objects like `3+4j < 5+7j` is not valid.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack (last-in, first-out). Use `append()` to add an item and `pop()` to retrieve an item from the top.

```

stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack.pop()) # 7
print(stack.pop()) # 6
print(stack.pop()) # 5

```

### 5.1.2. Using Lists as Queues

Lists can be used as queues (first-in, first-out), but they are not efficient for that purpose. Appends and pops from the end are fast; inserts and pops from the beginning are slow because all other elements need to be shifted.

For efficient queues, use `collections.deque`:

```

from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # The first to arrive now leaves
print(queue.popleft()) # The second to arrive now leaves
print(queue) # Remaining queue in order of arrival

```

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists.

**Example:**

Create a list of squares:

```

squares = []
for x in range(10):
 squares.append(x**2)

```

Can be rewritten as:

```

squares = list(map(lambda x: x**2, range(10)))
or more concisely:

```

```
squares = [x**2 for x in range(10)]
```

List comprehensions consist of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses.

**Example:**

Combine elements of two lists if they are not equal:

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
Results: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

They can also contain complex expressions and nested functions, such as:

```
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
Results: ['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

## 5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any expression, including another list comprehension.

**Example:**

Transpose a matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
transposed = [[row[i] for row in matrix] for i in range(4)]
Result: [[1,5,9], [2,6,10], [3,7,11], [4,8,12]]
```

Or using `zip()`:

```
list(zip(*matrix))
Result: [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

## 5.2. The `del` Statement

The `del` statement removes an item from a list by index:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
del a[2:4]
del a[:] # clears entire list
You can also delete entire variables:
del a
```

## 5.3. Tuples and Sequences

Tuples are immutable sequences of values. They are created with comma-separated values, optionally enclosed in parentheses:

```
t = 12345, 54321, 'hello!'
u = t, (1, 2, 3, 4, 5)
```

Tuples are immutable, so item assignment raises `TypeError`, but they can contain mutable objects like lists:

```
v = ([1, 2, 3], [3, 2, 1])
```

Special syntax for single-item tuples:

```
empty = ()
singleton = 'hello', # Note trailing comma
len(empty) # 0
len(singleton) # 1
```

Packing and unpacking:

```
t = 12345, 54321, 'hello!'
x, y, z = t
```

## 5.4. Sets

Sets are unordered collections of unique elements:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
'orange' in basket # True
'a' = set('abracadabra')
'b' = set('alacazam')
```

```
a - b # letters in a but not in b
a | b # letters in a or b or both
a & b # letters in both a and b
a ^ b # letters in a or b but not both
```

Set comprehensions:

```
a = {x for x in 'abracadabra' if x not in 'abc'}
```

## 5.5. Dictionaries

Dictionaries are key-value stores:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
del tel['sape']
tel['irv'] = 4127
list(tel) # ['jack', 'guido', 'irv']
sorted(tel) # ['guido', 'irv', 'jack']
'guido' in tel # True
```

Creating dictionaries:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{x: x**2 for x in (2, 4, 6)}
dict(sape=4139, guido=4127, jack=4098)
```

## 5.6. Looping Techniques

Loop through dictionary items:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
 print(k, v)
```

Loop through sequence with index:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
```

Loop over multiple sequences:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}'.format(q, a))
```

Reverse a sequence:

```
for i in reversed(range(1, 10, 2)):
 print(i)
```

Sort and unique elements:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
 print(f)
```

Filtering data:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
```

## 5.7. More on Conditions

Operators:

- `in`, `not in` for membership tests
- `is`, `is not` for object identity
- Comparisons can be chained: `a < b == c`

Boolean expressions:

```
non_null = string1 or string2 or string3
```

Logical operators `and`, `or`, `not` support short-circuit evaluation.

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Comparisons between different types may raise `TypeError` depending on the objects.

---

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- `list.append(x)`  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- `list.extend(iterable)`  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- `list.insert(i, x)`  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`  
Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.
- `list.pop([i])`  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.
- `list.clear()`  
Remove all items from the list. Similar to `del a[:]`.
- `list.index(x[, start[, end]])`  
Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.
- `list.count(x)`  
Return the number of times `x` appears in the list.
- `list.sort(*, key=None, reverse=False)`  
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).
- `list.reverse()`  
Reverse the elements of the list in place.
- `list.copy()`  
Return a shallow copy of the list. Similar to `a[:]`.

**Example using most list methods:**

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
```



```

2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4) # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'

```

*Note:* Methods like `insert`, `remove`, or `sort` that only modify the list have no return value; they return `None`. This design principle applies to all mutable data structures in Python.

*Note:* Not all data can be sorted or compared. For example, `[None, 'hello', 10]` doesn't sort because integers cannot be compared to strings, and `None` cannot be compared to other types. Also, types like `complex` numbers ( `3+4j` ) do not support ordering comparisons.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. Example:

```

>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack
[3, 4]

```

### 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first retrieved ("first-in, first-out"). However, lists are not efficient for this purpose:

- Append and pop from the end are fast.
- Insert or pop from the beginning are slow because all other elements need to be shifted.

To implement a queue efficiently, use `collections.deque`, which is designed for fast appends and pops from both ends.

Example:

```
>>> from collections import deque
>>> queue = deque(['Eric', 'John', 'Michael'])
>>> queue.append('Terry') # Terry arrives
>>> queue.append('Graham') # Graham arrives
>>> queue.popleft() # The first to arrive now leaves
'Eric'
>>> queue.popleft() # The second to arrive leaves
'John'
>>> queue # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

---

## 5.1.3. List Comprehensions

List comprehensions offer a concise way to create lists. They are useful for creating new lists by applying operations to each element of an existing sequence or iterable, or filtering elements based on a condition.

Example: create a list of squares:

```
>>> squares = []
>>> for x in range(10):
... squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Using `map()` and `lambda`:

```
squares = list(map(lambda x: x**2, range(10)))
```

Or, more concisely with list comprehension:

```
squares = [x**2 for x in range(10)]
```

List comprehensions are composed of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result is a list evaluated from these components.

Additional examples:

- Combining elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- Creating a list of doubles or filtered values:

```
vec = [-4, -2, 0, 2, 4]
[x*2 for x in vec] # doubles
[x for x in vec if x >= 0] # non-negative
[abs(x) for x in vec] # absolute values
[weapon.strip() for weapon in freshfruit] # calling a method on each element
[(x, x**2) for x in range(6)] # list of tuples
```

*Note:* For tuples, parentheses are required; otherwise, a syntax error occurs:

```
[x, x**2 for x in range(6)] # SyntaxError
```

# Correct:

```
[(x, x**2) for x in range(6)]
```

*Example of flattening a nested list:*

```
vec = [[1,2,3], [4,5,6], [7,8,9]]
[result for elem in vec for result in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

---

## 5.1.4. Nested List Comprehensions

List comprehensions can be nested, with the initial expression including multiple `for` clauses. For example, to transpose a 3x4 matrix:

```
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
transposed = [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Equivalent to:

```
transposed = []
for i in range(4):
 transposed.append([row[i] for row in matrix])
```

The use of the built-in `zip()` function is often preferred:

```
list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

---

## 5.2. The `del` Statement

The `del` statement removes an item from a list by index, or slices of a list, or clears an entire list. It can also delete entire variables.

Examples:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
>>> del a
```

*Note:* After deleting a variable, referencing it again causes an error.

---

## 5.3. Tuples and Sequences

Lists and strings have many common properties such as indexing and slicing. They are sequences. Python also includes the `tuple` data type, which is an immutable sequence.

Examples:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t = (12345, 54321, 'hello!')
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> t[0] = 88888
TypeError: 'tuple' object does not support item assignment
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Tuples are enclosed in parentheses, especially for nested or multiple element tuples. Single-element tuples require a trailing comma:

```
empty = ()
singleton = 'hello', # Note trailing comma
len(empty) # 0
len(singleton) # 1
```

*Tuple packing and unpacking example:*

```
t = 1, 2, 3
x, y, z = t
```

---

## 5.4. Sets

Python includes a `set` data type for unordered collections of unique elements. They support mathematical operations such as union, intersection, difference, and symmetric difference.

Examples:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False

Set operations:
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b # letters in a but not in b
```

```
{'r', 'd', 'b'}
>>> a | b # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b # letters in both a and b
{'a', 'c'}
>>> a ^ b # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
{'r', 'd'}
```

---

## 5.5. Dictionaries

Dictionaries are built-in mappings of key-value pairs.

- Keys can be any immutable type, such as strings, numbers, or tuples containing only immutable objects.
- Values can be any object.

Examples:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Dictionary constructor:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dict comprehensions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Keyword arguments for small dictionaries:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

---

## 5.6. Looping Techniques

- To retrieve keys and values during dictionary iteration:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
... print(k, v)
...
gallahad the pure
robin the brave
```

- To iterate over sequence indices and values:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
... print(i, v)
...
0 tic
1 tac
2 toe
```

- To iterate over multiple sequences simultaneously:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}.'.format(q, a))
```

- To iterate in reverse order:

```
for i in reversed(range(1, 10, 2)):
 print(i)
```

- To iterate over a sorted sequence:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for i in sorted(basket):
 print(i)
```

- To process unique elements:

```
for f in sorted(set(basket)):
 print(f)
```

- To filter data:

```
import math
raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
filtered_data = []
for value in raw_data:
 if not math.isnan(value):
 filtered_data.append(value)
```

---

## 5.7. More on Conditions

- Membership tests and identity tests:

```
>>> 'a' in 'banana'
True
>>> 'a' not in 'banana'
False
>>> a = b = c = 'python'
```

```
>>> a is b
True
>>> a is not c
False
```

- Chained comparisons:

```
>>> a = 5
>>> b = 10
>>> c = 15
>>> a < b == c - 10
True
```

- Logical operators:

```
>>> a = True
>>> b = False
>>> a and b
False
>>> a or b
True
>>> not a
False
```

- Combining conditions with `and`, `or`, and `not`:

```
if (a > 0) and (b > 0):
 print('Both positive')
```

- Assigning Boolean expressions:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3
```

*Note:* The "walrus operator" `:=` allows assignment inside expressions:

```
if (n := len(s)) > 10:
 print(f"Long string of length {n}")
```

---

## 5.8. Comparing Sequences and Other Types

Sequence objects are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'))
```

Comparisons between objects of different types are valid if the types support comparison; otherwise, a `TypeError` is raised.

---

*Footnotes:*

1. Other languages may allow method chaining with method returns, such as `d->insert("a")->remove("b")->sort();`.
- 

*End of content.*



# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

- **append(x)**  
Add an item to the end of the list. Similar to `a[len(a):] = [x]`.
- **extend(iterable)**  
Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.
- **insert(i, x)**  
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- **remove(x)**  
Remove the first item from the list whose value is equal to x. It raises a `ValueError` if there is no such item.
- **pop([i])**  
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.
- **clear()**  
Remove all items from the list. Similar to `del a[:]`.
- **index(x, [start], [end])**  
Return zero-based index in the list of the first item whose value is equal to x. Raises a `ValueError` if there is no such item. Optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.
- **count(x)**  
Return the number of times x appears in the list.
- **sort(\*, key=None, reverse=False)**  
Sort the items of the list in place (arguments can be used for customization, see [sorted\(\)](#)).
- **reverse()**  
Reverse the elements of the list in place.
- **copy()**  
Return a shallow copy of the list. Similar to `a[:]`.

An example that uses most of the list methods:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.count('apple') # 2
fruits.count('tangerine') # 0
fruits.index('banana') # 3
```

```

fruits.index('banana', 4) # 6 (find next banana starting at position 4)
fruits.reverse()
fruits is now ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
fruits.append('grape')
fruits.sort()
fruits is now ['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
fruits.pop() # 'pear'

```

You might have noticed that methods like `insert`, `remove`, or `sort` that only modify the list have no return value printed—they return the default `None`. [1] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn't sort because integers can't be compared to strings, and `None` can't be compared to other types. Also, some types don't have a defined ordering relation. For example, `3+4j < 5+7j` isn't valid.

### 5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top, use `pop()` without an index. Example:

```

stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack is [3, 4, 5, 6, 7]
print(stack.pop()) # 7
print(stack) # [3, 4, 5, 6]
print(stack.pop()) # 6
print(stack.pop()) # 5
stack is now [3, 4]

```

### 5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. Appends and pops from the end are fast, but insertions or pops from the start are slow because all other elements need to be shifted.

To implement a queue efficiently, use `[collections.deque]` (`./library/collections.html#collections.deque`) which supports fast appends and pops from both ends. Example:

```

from collections import deque

queue = deque(["Eric", "John", "Michael"])
queue.append("Terry") # Terry arrives
queue.append("Graham") # Graham arrives
print(queue.popleft()) # The first to arrive now leaves ('Eric')
print(queue.popleft()) # The second to arrive now leaves ('John')
print(queue) # Remaining queue in order of arrival (deque(['Michael', 'Terry'])

```

### 5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications include making new lists where each element is the result of some operation applied to each member of another sequence or iterable, or to create a subsequence of those elements satisfying a condition.

For example, create a list of squares:

```
squares = []
for x in range(10):
 squares.append(x**2)
squares is [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Can be rewritten as:

```
squares = list(map(lambda x: x**2, range(10)))
or, more concisely:
squares = [x**2 for x in range(10)]
```

List comprehensions have the form `[expression for item in iterable if condition]`.

Examples:

```
Create list of doubled values
[x * 2 for x in vec]

Filter list to exclude negative numbers
[x for x in vec if x >= 0]

Apply abs() to all elements
[abs(x) for x in vec]

Call a method on each element
[f.strip() for f in freshfruit]

List of 2-tuples (number, square)
[(x, x**2) for x in range(6)]

Nested list comprehension for matrix transpose
matrix = [
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
transposed = [[row[i] for row in matrix] for i in range(4)]
or using zip
list(zip(*matrix))
```

Note: For a tuple (e.g., `(x, y)`), parentheses are required; otherwise, syntax errors occur.

**Common list comprehension pitfalls:**

```
Incorrect: missing parentheses for tuple
[x, x**2 for x in range(6)] # SyntaxError
Correct:
[(x, x**2) for x in range(6)]
```

## 5.1.4. Nested List Comprehensions

List comprehensions can contain arbitrary expressions, including other list comprehensions. For example, to create a 3x4 matrix:

```
matrix = [
 [1, 2, 3, 4],
```

```

 [5, 6, 7, 8],
 [9, 10, 11, 12]
]
Transpose rows and columns
transposed = [[row[i] for row in matrix] for i in range(4)]

```

Equivalent to:

```

transposed = []
for i in range(4):
 transposed_row = []
 for row in matrix:
 transposed_row.append(row[i])
 transposed.append(transposed_row)

```

The recommended approach uses the built-in function `zip()`:

```

list(zip(*matrix))
[('1', '5', '9'), ('2', '6', '10'), ('3', '7', '11'), ('4', '8', '12')]

```

## 5.2. The `del` Statement

You can remove an item by index using `del`. It differs from `pop()` in that it does not return the value. It can also delete slices or entire variables:

```

a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a is now [1, 66.25, 333, 333, 1234.5]
del a[2:4]
a is now [1, 66.25, 1234.5]
del a[:]
a is now []
del a # deletes variable 'a' entirely

```

Referencing `a` afterwards causes an error unless another value is assigned. Use `del` later as needed.

## 5.3. Tuples and Sequences

Lists and strings share properties like indexing and slicing. Tuples are another sequence type, created by comma-separated values:

```

t = 12345, 54321, 'hello!'
print(t[0]) # 12345
t = (12345, 54321, 'hello!')
Tuples can be nested:
u = t, (1, 2, 3, 4, 5)
print(u) # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
Tuples are immutable:
t[0] = 88888 # TypeError
They can contain mutable objects:
v = ([1, 2, 3], [3, 2, 1]) # tuple of lists

```

Tuples are encased in parentheses for nested or multiple items; for singletons, a trailing comma is required:

```

empty = ()
singleton = 'hello', # note the trailing comma

```

```
len(empty) # 0
len singleton # 1
print(singleton) # ('hello',)
```

Sequence unpacking:

```
x, y, z = t
```

## 5.4. Sets

Python has a set type, an unordered collection with no duplicates:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # {'orange', 'banana', 'pear', 'apple'}
print('orange' in basket) # True
print('crabgrass' in basket) # False
Set operations
a = set('abracadabra')
b = set('alacazam')
a - b # elements in a but not in b
a | b # union
a & b # intersection
a ^ b # symmetric difference
Set comprehension
a = {x for x in 'abracadabra' if x not in 'abc'}
print(a) # {'r', 'd'}
```

## 5.5. Dictionaries

Dictionaries store key-value pairs, with keys being immutable types:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print(tel) # {'jack': 4098, 'sape': 4139, 'guido': 4127}
del tel['sape']
tel['irv'] = 4127
print(tel) # {'jack': 4098, 'guido': 4127, 'irv': 4127}
list(tel) # ['jack', 'guido', 'irv']
sorted(tel) # ['guido', 'irv', 'jack']
'guido' in tel # True
'jack' not in tel # False
```

Constructors:

```
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
Output: {'sape': 4139, 'guido': 4127, 'jack': 4098}
{x: x**2 for x in (2, 4, 6)} # Dict comprehension
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Keyed argument syntax:

```
dict(sape=4139, guido=4127, jack=4098)
```

## 5.6. Looping Techniques

Iterate through dictionary items:

```
knight = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knight.items():
 print(k, v)
Outputs:
gallahad the pure
robin the brave
```

Iterate through sequences with indices:

```
for i, v in enumerate(['tic', 'tac', 'toe']):
 print(i, v)
Outputs:
0 tic
1 tac
2 toe
```

Pair sequences with `zip()`:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
 print('What is your {0}? It is {1}'.format(q, a))
```

Reverse iteration:

```
for i in reversed(range(1, 10, 2)):
 print(i)
Outputs 9, 7, 5, 3, 1
```

Sorted iteration over unique elements:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
 print(f)
Outputs:
apple
banana
orange
pear
```

## 5.7. More on Conditions

Operators `in` / `not in` test membership, `is` / `is not` compare object identity. Comparisons can be chained:

```
a < b == c # same as a < b and b == c
```

Operators `and`, `or`, and `not` combine conditions:

```
string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
non_null = string1 or string2 or string3 # 'Trondheim'
```

Note: Python uses the walrus operator `:=` for inline assignments within expressions.

## 5.8. Comparing Sequences and Other Types

Sequences are compared lexicographically:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C'
(1, 2, 3, 4) < (1, 2)
(1, 2) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4) # compares nested tuples
```

Comparing objects of different types with `<` or `>` is only valid if they have appropriate comparison methods; otherwise, a `TypeError` is raised.

---

## Footer

© [Python Software Foundation](#), 2001–2025

This page is licensed under the Python Software Foundation License Version 2.

Examples and code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit. [Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#)); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:



```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name `fibo` from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `importlib.reload()`, e.g.,

```
import importlib
importlib.reload(module_name)
```

### 6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

### 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the `sys.path` module search path](#).

**Note:** On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#) for more information.

### 6.1.3. “Compiled” Python files

---

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3, the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This process is automatic. Moreover, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances: first, it always recompiles and does not store the result for the module loaded directly from the command line; second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements; the `-OO` switch removes both assert statements and `__doc__` strings.
- A program doesn't run any faster when read from a `.pyc` file than from a `.py` file; the only speed advantage is in loading.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- More detail about this process, including a flow chart of the decisions, can be found in [PEP 3147](#).

## 6.2. Standard Modules

Python includes a library of standard modules, described in the [Python Library Reference](#). Some modules are built into the interpreter; these provide access to operations that are not part of the language core but are built in for efficiency or to access operating system primitives like system calls. The set of such modules is platform-dependent (e.g., `winreg` only on Windows).

A key module is `sys`, which is built into every Python interpreter. It defines variables such as:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
>>> print('Yuck!')
Yuck!
C>
```

These variables are only defined in interactive mode.

The variable `sys.path` determines where Python searches for modules. You can modify it using list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` finds out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', ...]
```

Without arguments, `dir()` lists the names currently defined:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note: `dir()` lists all types of names: variables, modules, functions, etc.

It does not list the names of built-in functions and variables. To get these, import the `builtins` module:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ...]
```

## 6.4. Packages

Packages organize modules into hierarchies by using dotted module names. For instance, `A.B` designates a submodule `B` in package `A`. This structure helps avoid name conflicts among modules and large collections of modules (like NumPy or Pillow).

Suppose you want to create a collection for handling sound files and data, with formats like `wav`, `aiff`, and operations like effects and filters. Your directory structure might look like this:

|              |                             |
|--------------|-----------------------------|
| sound/       | Top-level package           |
| __init__.py  | Initialize sound package    |
| formats/     | Subpackage for file formats |
| __init__.py  |                             |
| wavread.py   |                             |
| wavwrite.py  |                             |
| aiffread.py  |                             |
| aiffwrite.py |                             |
| auread.py    |                             |
| auwrite.py   |                             |
| effects/     | Subpackage for effects      |
| __init__.py  |                             |
| echo.py      |                             |
| surround.py  |                             |
| reverse.py   |                             |
| filters/     | Subpackage for filters      |
| __init__.py  |                             |
| equalizer.py |                             |
| vocoder.py   |                             |
| karaoke.py   |                             |

Python searches the directories in `sys.path` when importing packages. The presence of `__init__.py` files indicates directories are packages (unless using namespace packages).

You can import submodules as follows:

```
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Or import specific submodules:

```
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)
```

Or import specific functions:

```
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)
```

### 6.4.1. Importing \* From a Package

When using `from package import *`, the package's `__init__.py` must define a list `__all__`, containing the module names to import:

```
__all__ = ["echo", "surround", "reverse"]
```

Without `__all__`, `import *` from a package only imports the package itself, not submodules. To control what gets imported, maintain this list carefully. For example, if `__init__.py` contains an `__all__` as above, then:

```
from sound.effects import *
```

imports the specified submodules. If a `reverse` function is added directly in `__init__.py`, it shadows the submodule:

```
__all__ = ["echo", "surround", "reverse"]
def reverse(msg: str): # shadows 'reverse' submodule
 return msg[::-1]
```

Importing `*` does not import submodules if `__all__` is not defined, only the names in `__all__`. This includes all names explicitly loaded by earlier import statements.

### 6.4.2. Intra-package References

Within packages, absolute imports explicitly specify the package path:

```
from sound.effects import echo
```

Relative imports use leading dots:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Modules intended to be run as the main script should always use absolute imports because `__name__` is `"__main__"` in that case.

### 6.4.3. Packages in Multiple Directories

The `__path__` attribute supports multiple directories for a package, initialized as a sequence of strings indicating where the package is located. It can be modified to include additional directories, extending the set of modules found.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`).

In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as fb
>>> fb.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `fb`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

**Note:**

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `importlib.reload()`, e.g., `import importlib; importlib.reload(modulename)`.

---

### 6.1.1. Executing modules as scripts

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

---

### 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the `sys.path` module search path](#).

**Note:**

On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#).

---



### 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and `__doc__` strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. “Optimized” modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` files is the speed with which they are loaded.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in [PEP 3147](#).

---

## 6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

---

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fibo.fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
```

```
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

---

## 6.4. Packages

Packages are a way of structuring Python's module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

|                           |                                        |
|---------------------------|----------------------------------------|
| <code>sound/</code>       | Top-level package                      |
| <code>__init__.py</code>  | Initialize the sound package           |
| <code>formats/</code>     | Subpackage for file format conversions |
| <code>__init__.py</code>  |                                        |
| <code>wavread.py</code>   |                                        |
| <code>wavwrite.py</code>  |                                        |
| <code>aiffread.py</code>  |                                        |
| <code>aiffwrite.py</code> |                                        |
| <code>auread.py</code>    |                                        |
| <code>auwrite.py</code>   |                                        |
| <code>...</code>          |                                        |
| <code>effects/</code>     | Subpackage for sound effects           |
| <code>__init__.py</code>  |                                        |
| <code>echo.py</code>      |                                        |
| <code>surround.py</code>  |                                        |
| <code>reverse.py</code>   |                                        |

```

...
filters/ Subpackage for filters
 __init__.py
 equalizer.py
 vocoder.py
 karaoke.py
 ...

```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a [namespace package](#), a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from <package> import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

## 6.4.1. Importing \* From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named

`__all__`, it is taken to be the list of module names that should be imported when `from <package> import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing `*` from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound.effects` package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a `reverse` function to the `sound/effects/__init__.py` file, the `from sound.effects import *` would only import the two submodules `echo` and `surround`, but **not** the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
 "echo", # refers to the 'echo.py' file
 "surround", # refers to the 'surround.py' file
 "reverse", # !!! refers to the 'reverse' function now !!!
]
```

```
def reverse(msg: str): # <-- this name shadows the 'reverse.py' submodule
 return msg[::-1] # in the case of a 'from sound.effects import *'
```

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from <package> import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#)); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. <sup>1</sup> (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module:

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter — or, if it's just one module you want to test interactively, use `importlib.reload()`, e.g.

```
import importlib; importlib.reload(modulename)
```

## 6.1.1. Executing modules as scripts

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

## 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`.

`sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the sys.path module search path](#).

**Note** On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended.



### 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3, the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This process is automatic. The compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances:

- The module is loaded directly from the command line.
- There is no source module (i.e., only a compiled `.pyc` exists).

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and `__doc__` strings.
- A program doesn't run any faster when read from a `.pyc` file than from a `.py` file; it is just loaded faster.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- More details are at [PEP 3147](#).

## 6.2. Standard Modules

Python comes with a library of standard modules, described in the [Python Library Reference](#). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nonetheless built in, either for efficiency or to provide access to operating system primitives such as system calls. For example, the `sys` module is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized from the environment variable `PYTHONPATH` or a default. You can modify it with standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', ...]
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. To do that, import the `builtins` module and call `dir()` on it:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ...]
```

## 6.4. Packages

Packages are a way of structuring Python's module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule `B` in a package named `A`. Using dotted names saves authors of multi-module packages like NumPy or Pillow from worrying about module name conflicts.

Suppose you want to design a collection of modules for sound processing, organized hierarchically as:

|                               |                                        |
|-------------------------------|----------------------------------------|
| <code>sound/</code>           | Top-level package                      |
| <code>  __init__.py</code>    | Initialize the sound package           |
| <code>  formats/</code>       | Subpackage for file format conversions |
| <code>    __init__.py</code>  |                                        |
| <code>    wavread.py</code>   |                                        |
| <code>    wavwrite.py</code>  |                                        |
| <code>    aiffread.py</code>  |                                        |
| <code>    aiffwrite.py</code> |                                        |
| <code>    auread.py</code>    |                                        |
| <code>    auwrite.py</code>   |                                        |
| <code>  effects/</code>       | Subpackage for sound effects           |
| <code>    __init__.py</code>  |                                        |
| <code>    echo.py</code>      |                                        |
| <code>    surround.py</code>  |                                        |
| <code>    reverse.py</code>   |                                        |
| <code>  filters/</code>       | Subpackage for filters                 |
| <code>    __init__.py</code>  |                                        |
| <code>    equalizer.py</code> |                                        |
| <code>    vocoder.py</code>   |                                        |
| <code>    karaoke.py</code>   |                                        |

When importing the package, Python searches through directories in `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages, unless using *namespace packages*. They prevent directories with common names (like `string`) from hiding valid modules later in the search path. An empty `__init__.py` is sufficient, but it can also execute initialization code and set the `__all__` variable.

Users can import individual modules directly, e.g.,:

```
import sound.effects.echo
```

or:

```
from sound.effects import echo
```

and call functions as:

```
sound.effects.echo.echofilter(input, output, delay=0.7,atten=4)
```

or:

```
from sound.effects import echo
echo.echofilter(input, output, delay=0.7,atten=4)
```

or:

```
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7,atten=4)
```

When using `from package import *`, the `__init__.py` files should define `__all__` as a list of module names to import. If `__all__` is not defined, only the package initialization code runs, and names explicitly loaded are imported.

Relative imports can be used within packages, e.g.:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that modules used as the main application module must use absolute imports, since `__name__` is `"__main__"`.

Packages can support multiple directories by modifying `__path__`, a sequence of strings with directory names where the package's `__init__.py` is. This can allow a package to extend its module search path dynamically.

---

**1:** Function definitions are statements that are executed at the module level, thereby adding the function name to the module's global namespace.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current [namespace](#) (see [Python Scopes and Namespaces](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `importlib.reload()`, e.g., `import importlib; importlib.reload(modulename)`.

## 6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

## 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in [sys.builtin\\_module\\_names](#). If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the [site](#) module).

More details are at [The initialization of the sys.path module search path](#).

### Note:

On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#) for more information.

## 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and `__doc__` strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. "Optimized" modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` files is the speed with which they are loaded.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in [PEP 3147](#).

## 6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fibo.fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',
```



```
'PermissionError', 'ProcessLookupError', 'ReferenceError', 'ResourceWarning',
'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError',
'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '_', '__build_class__', '__debug__', '__doc__',
'__import__', '__name__', '__package__', 'abs', 'all', 'any', 'ascii', 'bin',
'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open',
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

## 6.4. Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/ # Top-level package
 __init__.py # Initialize the sound package
 formats/ # Subpackage for file format conversions
 __init__.py
 wavread.py
 wavwrite.py
 aiffread.py
 aiffwrite.py
 auread.py
 auwrite.py
 effects/ # Subpackage for sound effects
 __init__.py
 echo.py
 surround.py
 reverse.py
 filters/ # Subpackage for filters
 __init__.py
 equalizer.py
 vocoder.py
 karaoke.py
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a [namespace package](#), a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an [ImportError](#) exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

### 6.4.1. Importing \* From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing \* from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound.effects` package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a `reverse` function to the `sound/effects/__init__.py` file, the `from sound.effects import *` would only import the two submodules `echo` and `surround`, but *not* the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
 "echo", # refers to the 'echo.py' file
 "surround", # refers to the 'surround.py' file
 "reverse", # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str): # ← this name shadows the 'reverse.py' submodule
 return msg[::-1] # in the case of a 'from sound.effects import *'
```

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.[1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all [import statements](#) at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the import statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `importlib.reload()`, e.g.:

```
import importlib; importlib.reload(modulename)
```

### 6.1.1. Executing modules as scripts

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

### 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in [sys.builtin\\_module\\_names](#). If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the [site](#) module).

More details are at [The initialization of the sys.path module search path](#).

*Note:* On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is **not** added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#) for more information.

### 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and `__doc__` strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. "Optimized" modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` files is the speed with which they are loaded.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in [PEP 3147](#).

## 6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1='C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Without arguments, `dir()` lists the names you have currently defined:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
```



```
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

## 6.4. Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

|                           |                                        |
|---------------------------|----------------------------------------|
| <code>sound/</code>       | Top-level package                      |
| <code>__init__.py</code>  | Initialize the sound package           |
| <code>formats/</code>     | Subpackage for file format conversions |
| <code>__init__.py</code>  |                                        |
| <code>wavread.py</code>   |                                        |
| <code>wavwrite.py</code>  |                                        |
| <code>aiffread.py</code>  |                                        |
| <code>aiffwrite.py</code> |                                        |
| <code>auread.py</code>    |                                        |
| <code>auwrite.py</code>   |                                        |
| <code>...</code>          |                                        |
| <code>effects/</code>     | Subpackage for sound effects           |
| <code>__init__.py</code>  |                                        |
| <code>echo.py</code>      |                                        |
| <code>surround.py</code>  |                                        |
| <code>reverse.py</code>   |                                        |
| <code>...</code>          |                                        |
| <code>filters/</code>     | Subpackage for filters                 |
| <code>__init__.py</code>  |                                        |
| <code>equalizer.py</code> |                                        |
| <code>vocoder.py</code>   |                                        |

```
karaoke.py
...
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a [namespace package](#), a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

This also loads the submodule `echo`, but this time directly makes its function `echofilter()` available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `<package> import <item>`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an [ImportError](#) exception is raised.

Contrarily, when using syntax like `<import> <item.subitem.subsubitem>`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

## 6.4.1. Importing \* From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing \* from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound.effects` package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a `reverse` function to the `sound/effects/__init__.py` file, the `from sound.effects import *` would only import the two submodules `echo` and `surround`, but **not** the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
 "echo", # refers to the 'echo.py' file
 "surround", # refers to the 'surround.py' file
 "reverse", # !!! refers to the 'reverse' function now !!!
]
```

```
def reverse(msg: str) # <-- this name shadows the 'reverse.py' submodule
 return msg[::-1] # in the case of a 'from sound.effects import *'
```

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from <package> import <specific_submodule>`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. ([1]) (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all [import statements](#) at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the import statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken into the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module:

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilizing `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

**Note:** For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `importlib.reload()`, e.g.

```
import importlib; importlib.reload(modulename)
```

## 6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This pattern is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

## 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the sys.path module search path](#).

**Note:** On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See [Standard Modules](#) for more information.

## 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3, the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This process is automatic. The compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances:

- It always recompiles and does not store the result when the module is loaded directly from the command line.
- It does not check the cache if there is no source module. To support a non-source (compiled-only) distribution, the compiled module must be in the source directory, and no source module should exist.

**Tips for experts:**

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module.
- A program doesn't run faster from a `.pyc` than from a `.py`; the load speed is faster.
- The `compileall` module can create `.pyc` files for all modules in a directory.
- More details are in [PEP 3147](#).

## 6.2. Standard Modules

Python provides a library of standard modules, described in the [Python Library Reference](#) (hereafter “Library Reference”). Some modules are built into the interpreter, providing access to operations not part of the core of the language but built in for efficiency or to access operating system primitives, like system calls. This set depends on the platform. For example, the `winreg` module exists only on Windows.

One module deserves special attention: `sys`, built into every Python interpreter. It defines:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These variables are only defined if the interpreter is in interactive mode. The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized from the environment variable `PYTHONPATH`, or from a default if `PYTHONPATH` is not set. You can modify it with list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', ...]
```

Without arguments, `dir()` lists the names you currently have defined:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. For those, use the `builtins` module:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', ..., 'zip']
```

## 6.4. Packages

Packages are a way of structuring Python's module namespace via dotted module names, e.g., `A.B` designates a submodule `B` in package `A`. Using packages prevents namespace clashes in multi-module packages like NumPy or Pillow.

Suppose you design a collection of modules for handling sound files. The directory structure might look like:

```
sound/ # Top-level package
__init__.py
formats/ # Subpackage for formats
__init__.py
wavread.py
wavwrite.py
aiffread.py
aiffwrite.py
auread.py
auwrite.py
effects/ # Subpackage for effects
__init__.py
echo.py
surround.py
reverse.py
filters/ # Subpackage for filters
__init__.py
equalizer.py
vocoder.py
karaoke.py
```

When importing, Python searches through `sys.path`. The `__init__.py` files are required to make Python treat directories as packages, unless using a namespace package. This prevents common names from hiding valid modules.

Examples of imports:



```
import sound.effects.echo
from sound.effects import echo
from sound.effects.echo import echofilter
```

You can import submodules directly or via `__init__.py`, and control what gets imported with `__all__`. Be cautious of shadowing submodule names with local definitions.

#### Submodule import placeholder

```
__all__ = ['echo', 'surround', 'reverse']
```

Be aware that `from package import *` only imports names listed in `__all__`, or all names if `__all__` is not defined, excluding those starting with `_`. It's generally discouraged in production code.

---

[1] Function definitions are also statements that are executed; executing a module-level function definition adds the function name to that module's global namespace.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#)) for more details; it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. (1) (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all [import statements](#) at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the [import](#) statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module:

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilizing `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `importlib.reload()`, e.g.:

```
>>> import importlib; importlib.reload(modulename)
```

## 6.1.1. Executing modules as scripts

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

## 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the sys.path module search path](#).

*Note:* On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See [Standard Modules](#).

### 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3, the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and `__doc__` strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. "Optimized" modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` files is the speed with which they are loaded.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in [PEP 3147](#).

---

## 6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
>>> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

---

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', ...]
```

Without arguments, `dir()` lists the names you have currently defined:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module [builtins](#):

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ...]
```

---

## 6.4. Packages

Packages are a way of structuring Python's module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

|             |                                        |
|-------------|----------------------------------------|
| sound/      | Top-level package                      |
| __init__.py | Initialize the sound package           |
| formats/    | Subpackage for file format conversions |
| __init__.py |                                        |

```

 wavread.py
 wavwrite.py
 aiffread.py
 aiffwrite.py
 auread.py
 auwrite.py
 ...
effects/ Subpackage for sound effects
 __init__.py
 echo.py
 surround.py
 reverse.py
 ...
filters/ Subpackage for filters
 __init__.py
 equalizer.py
 vocoder.py
 karaoke.py
 ...

```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a [namespace package](#), a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Alternatively, you can import the submodule directly:

```
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)
```

Or import the desired function or variable directly:

```
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class, or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an [ImportError](#) exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

## 6.4.1. Importing \* From a Package

When you write `from sound.effects import *`, ideally, this should go out to the filesystem, find which submodules are present in the package, and import them all. However, importing submodules might have unwanted side-effects, and it could take a long time.

Therefore, package authors should provide an explicit index with a list named `__all__` in the `__init__.py` file. For example:

```
__all__ = ["echo", "surround", "reverse"]
```

This signifies that `from sound.effects import *` would import these three submodules of the `sound.effects` package.

Be aware that submodules might become shadowed by locally defined names. For example, if you add a `reverse` function to the `sound/effects/__init__.py` file, `from sound.effects import *` will only import the two submodules `echo` and `surround`, but **not** the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
 "echo", # refers to the 'echo.py' file
 "surround", # refers to the 'surround.py' file
 "reverse", # <--- this refers to the function, shadowing the submodule
]

def reverse(msg: str) -> str: # <--- this name shadows the 'reverse.py' submodule
 return msg[::-1]
```

If `__all__` is not defined, the statement `from sound.effects import *` does **not** import all submodules; it only ensures that the package `sound.effects` has been imported, and then it imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`, as well as any submodules explicitly loaded by previous `import` statements, for example:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from ... import *` statement is executed (also works when `__all__` is defined). However, using `import *` in production code is generally discouraged, as it reduces code readability. It remains acceptable in interactive sessions for convenience.

Although certain modules are designed to export only names that follow certain patterns, using `from package import *` is considered bad practice in production.

Remember, it's better to use explicit imports like:

```
from sound.effects import echo
```

unless the import is really needed.

## 6.4.2. Intra-package References

When packages are structured into subpackages, you can use absolute imports to refer to submodules in sibling packages. For example, if `sound.filters.vocoder` needs to use the `echo` module in `sound.effects`, it can use:

```
from sound.effects import echo
```

You can also write relative imports, using leading dots to indicate the current and parent packages involved:



```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

### 6.4.3. Packages in Multiple Directories

Packages support a special attribute, `__path__`, initialized as a sequence of strings containing the directory of the package before the code in `__init__.py` runs. It can be modified to affect future module searches, allowing the set of modules in a package to be extended.

This feature is seldom needed but can be used for advanced use cases, such as extending or modifying package contents dynamically.

---

*Footnotes:*

[1](#id1) In fact, function definitions are also ‘statements’ that are ‘executed’; executing a module-level function definition adds the function name to the module’s global namespace.

## 6. Modules ¶

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules¶

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname .
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module:

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

#### Note

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `importlib.reload()`, e.g.:

```
import importlib; importlib.reload(modulename)
```

## 6.1.1. Executing modules as scripts¶

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

## 6.1.2. The Module Search Path¶

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the sys.path module search path](#).

#### Note

On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See [Standard Modules](#).

## 6.1.3. “Compiled” Python files¶

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This process is automatic. The compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances:

- When the module is loaded directly from the command line.
- When no source module exists, only the compiled `.pyc` file.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements; the `-OO` switch removes both assert statements and `__doc__` strings.
- A program doesn't run faster when read from a `.pyc` file than from a `.py` file; loading `.pyc` only speeds up the startup.
- The `compileall` module can create `.pyc` files for all modules in a directory.
- More details are in [PEP 3147](#).

## 6.2. Standard Modules¶

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference”). Some modules are built into the interpreter and provide access to various runtime operations. The set of such modules depends on the platform.

One module deserves special note: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1='C> '
C> print('Yuck!')
Yuck!
C>
```

These variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the search path for modules. It is initialized from the environment variable `PYTHONPATH`, or from a default if not set:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function¶

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', ...]
```

Without arguments, `dir()` lists the names you have currently defined:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

It lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. To list those, import the `builtins` module:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ...]
```

## 6.4. Packages ¶

Packages structure Python's module namespace using dotted module names. For example, `A.B` indicates a submodule `B` in package `A`. This helps organize code and prevent name clashes in large projects.

Suppose you want to design a collection of modules for handling sound files, with formats like `.wav`, `.aiff`, `.au`, etc., and operations like mixing or applying effects. A possible directory structure:

```
sound/ # Top-level package
 __init__.py # Initialize the sound package
 formats/ # Subpackage for formats
 __init__.py
 wavread.py
 wavwrite.py
 aiffread.py
 aiffwrite.py
 auread.py
 auwrite.py
 effects/ # Subpackage for effects
 __init__.py
 echo.py
 surround.py
 reverse.py
 filters/ # Subpackage for filters
 __init__.py
 equalizer.py
 vocoder.py
 karaoke.py
```

Importing the package involves Python searching `sys.path` for the directory containing the package. The `__init__.py` files make directories into packages (unless using namespace packages). These files can be

empty or contain initialization code.

Examples:

```
import sound.effects.echo
from sound.effects import echo
from sound.effects.echo import echofilter
```

When using `from sound.effects import *`, a list named `__all__` defined in `__init__.py` guides what to import:

```
__all__ = ["echo", "surround", "reverse"]
```

Note that local names can shadow submodules. For example, redefining `reverse` as a function in `__init__.py`:

```
__all__ = ["echo", "surround", "reverse"]
def reverse(msg: str):
 return msg[::-1]
```

would mean `from sound.effects import *` does not import the `reverse` submodule, as the `reverse` function shadows it.

**Remark:** Although `import *` is sometimes used in interactive sessions, it's discouraged in production code for readability reasons.

---

[1] Function definitions are also “statements” that are executed; executing a function definition adds its name to the module's global namespace.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a **module**; definitions from a module can be *imported* into other modules or into the **main** module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current **namespace** (see [Python Scopes and Namespaces](#) for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:



```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname.
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name `fibo` from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module:

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

**Note:** For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `importlib.reload()`, e.g., `import importlib; importlib.reload(modulename)`.

### 6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

### 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the sys.path module search path](#).

**Note:** On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is **not** added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See [Standard Modules](#) for more information.

### 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.<version>.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This process is automatic. The compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances:

- It always recompiles and does not store the result for the module that's loaded directly from the command line.
- It does not check the cache if there is no source module.

To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

**Tips for experts:**

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes `assert` statements, the `-OO` switch removes both `assert` statements and `__doc__` strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. "Optimized" modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` files is the speed with which they are loaded.
- The `compileall` module can create `.pyc` files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in [PEP 3147](#).

## 6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are available for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', ...]
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ...]
```

## 6.4. Packages

Packages are a way of structuring Python's module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, e.g., `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for conversion between formats. There are also many operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer, creating an artificial stereo effect), so you'll be writing many modules for these tasks. Here's a possible structure for your package:

```
sound/ # Top-level package
 __init__.py # Initialize the sound package
 formats/ # Subpackage for file format conversions
 __init__.py
 wavread.py
 wavwrite.py
 aiffread.py
 aiffwrite.py
 auread.py
 auwrite.py
 effects/ # Subpackage for sound effects
```

```

__init__.py
echo.py
surround.py
reverse.py
filters/ # Subpackage for filters
__init__.py
equalizer.py
vocoder.py
karaoke.py

```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a **namespace package**, a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name. Alternatively, you can import the submodule as:

```
from sound.effects import echo
```

which also loads the submodule `echo`, making it available without its package prefix:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

which loads the submodule `echo` but makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note about `from package import item`: The item can be either a submodule (or subpackage) or some other name defined in the package, like a function, class, or variable. The `import` statement first tests whether the item is defined in the package; if not, it attempts to load it as a module. If it fails to find it, an `ImportError` is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but not a class, function, or variable defined in the previous item.

### 6.4.1. Importing `*` From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, this should go to the filesystem, find all submodules in the package, and import them all. However, this could take a long time, and importing submodules might have side-effects that are undesirable unless explicitly imported.

The solution is for the package author to provide an explicit list of what should be imported via `__all__`. If `__init__.py` defines a list named `__all__`, it is taken as the list of module names to import when `from package import *` is encountered. It is up to the package author to keep this list current.

For example, `sound/effects/__init__.py` could contain:

```
__all__ = ["echo", "surround", "reverse"]
```

This means that `from sound.effects import *` imports these three submodules.

**Note:** Submodules may become shadowed by locally defined names. For example, if you add a `reverse` function to `sound/effects/__init__.py`, `from sound.effects import *` would only import the other submodules and not the new `reverse` function if it shadows the submodule.

If `__all__` is not defined, the statement `from sound.effects import *` only ensures that the package `sound.effects` has been imported (executing its `__init__.py`), and then it imports names defined in the package, including any submodules explicitly loaded or imported earlier. For example:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

This imports modules `echo` and `surround` because they are defined in the package at import time.

**Important:** Although using `*` is possible, it's generally discouraged outside interactive sessions due to poor readability.

## 6.4.2. Intra-package References

When packages are nested, absolute imports can refer to sibling modules. For example, if `sound.filters.vocoder` needs `sound.effects.echo`, it can use:

```
from sound.effects import echo
```

or via relative imports with leading dots (for current and parent packages):

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports depend on the module's name; modules for the main program should always use absolute imports.

## 6.4.3. Packages in Multiple Directories

`__path__` is an attribute of packages, initialized as a sequence of directory names containing the package's `__init__.py`. It can be modified to extend where Python looks for submodules, beyond the initial directory.

**Footnotes:**

[1] In fact, function definitions are also *statements* that are *executed*; executing a module-level function definition adds the function name to the module's global namespace.

## 6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#)); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname
```

Modules can import other modules. It is customary but not required to place all [import statements](#) at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the import statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```



Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `importlib.reload()`, e.g.:

```
import importlib; importlib.reload(modulename)
```

## 6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

## 6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the `sys.path` module search path](#).

**Note:** On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is **not** added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See [Standard Modules](#) for more information.

## 6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3, the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Additionally, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances:

- When loaded directly from the command line (it always recompiles).
- When there is no source module.

To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes `assert` statements; `-OO` removes both `assert` statements and `__doc__` strings.
- A program doesn't run any faster when loaded from a `.pyc` file; it's mainly faster to load.
- The `compileall` module can create `.pyc` files for all modules in a directory.
- More details are in [PEP 3147](#).

## 6.2. Standard Modules

Python comes with a library of standard modules, described in the *Python Library Reference*. Some modules are built into the interpreter; they provide access to operations that are not part of the core language but are built-in for efficiency or system access.

For example, the `sys` module is built into every Python interpreter. Its variables include:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings which determines the interpreter's search path for modules. It is initialized from the environment variable `PYTHONPATH` or from a default. You can modify it at runtime:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

## 6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', ..., 'version', 'version_info']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

It lists all types of names: variables, modules, functions, etc.

To list the names of built-in functions and variables, import the `builtins` module:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', ..., 'zip']
```

## 6.4. Packages

Packages are a way of structuring Python's module namespace via dotted names. For example, `A.B` refers to a submodule `B` in package `A`. This helps organize larger collections of modules like NumPy or Pillow and avoid name conflicts.

Suppose you want to design a package for handling sound files with different formats and effects. The structure might look like:

```
sound/ # Top-level package
 __init__.py # Initialize the sound package
 formats/ # Subpackage for file format conversions
 __init__.py
 wavread.py
 wavwrite.py
 aiffread.py
 aiffwrite.py
 auread.py
 auwrite.py
 effects/ # Subpackage for sound effects
 __init__.py
 echo.py
 surround.py
 reverse.py
 filters/ # Subpackage for filters
 __init__.py
 equalizer.py
 vocoder.py
 karaoke.py
```

The `__init__.py` files are required to make Python interpret directories as packages (unless using namespace packages). They can be empty or contain initialization code and the `__all__` variable.

You can import modules with:

```
import sound.effects.echo
```

or:

```
from sound.effects import echo
```

which makes `echo` available as `sound.effects.echo` or just `echo`.

To control what `from sound.effects import *` imports, define `__all__` in `sound/effects/__init__.py`:

```
__all__ = ["echo", "surround", "reverse"]
```

Note that if `__all__` is not defined, `import *` will only import submodules that are loaded or explicitly listed, and names shadowed by local definitions will not be imported.

**Relative imports:** Modules can use relative imports, such as:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

But modules intended as the main program should always use absolute imports since the main module's name is `__main__`.

**Multiple directories:** The `__path__` attribute in a package defines additional locations for modules, allowing a package to span multiple directories.

[1] In fact, function definitions are also statements that are executed, thus adding functions to a module's namespace during import time.

## 6. Modules ¶

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
Fibonacci numbers module

def fib(n):
 """Write Fibonacci series up to n."""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a + b
 print()

def fib2(n):
 """Return Fibonacci series up to n."""
 result = []
 a, b = 0, 1
 while a < n:
 result.append(a)
 a, b = b, a + b
 return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current *namespace* (see [Python Scopes and Namespaces](#)); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 6.1. More on Modules¶

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [1] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions,

```
modname.itemname.
```

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the `import` statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module:

```
>>> import fibo as f
>>> f.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `f`.

It can also be used when utilising `from` with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use `importlib.reload()`, e.g. `import importlib; importlib.reload(modulename)`.

### 6.1.1. Executing modules as scripts¶

When you run a Python module with:

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
 import sys
 fibo.fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

### 6.1.2. The Module Search Path¶

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at [The initialization of the sys.path module search path](#).

**Note:**

On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words, the directory containing the symlink is *not* added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See [Standard Modules](#) for more information.

### 6.1.3. “Compiled” Python files¶

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.version.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of `spam.py` would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances:

First, it always recompiles and does not store the result for the module that's loaded directly from the command line.

Second, it does not check the cache if there is no source module. To support a non-source (compiled-only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. `-O` removes assert statements, `-OO` removes both assert statements and `__doc__` strings. Since some programs may rely on having these available, use this only if you know what you're doing. "Optimized" modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing faster is the loading speed.
- The `compileall` module can create `.pyc` files for all modules in a directory.
- For more details, see [PEP 3147](#).

## 6.2. Standard Modules¶

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built-in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```



## 6.3. The `dir()` Function ¶

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', ...]
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fibo.fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ...]
```

## 6.4. Packages ¶

Packages are a way of structuring Python's module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/ # Top-level package
 __init__.py # Initialize the sound package
 formats/ # Subpackage for file format conversions
 __init__.py
 wavread.py
 wavwrite.py
 aiffread.py
 aiffwrite.py
 auread.py
 auwrite.py
 ...
 effects/ # Subpackage for sound effects
```

```

__init__.py
echo.py
surround.py
reverse.py
...
filters/ # Subpackage for filters
__init__.py
equalizer.py
vocoder.py
karaoke.py
...

```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a *namespace package*, a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name. Alternatively, you can do:

```
from sound.effects import echo
```

which also loads the submodule `echo`, making it available without its package prefix:

```
>>> echo.echofilter(input, output, delay=0.7, atten=4)
```

Another way is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
>>> echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

## 6.4.1. Importing \* From a Package ¶

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of

the package is released. Package authors may also decide not to support it, if they don't see a use for importing `*` from their package. For example, `sound/effects/__init__.py` could contain:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound.effects` package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a `reverse` function to `sound/effects/__init__.py`, the `from sound.effects import *` would only import the two submodules `echo` and `surround`, but *not* the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
 "echo", # refers to the 'echo.py' file
 "surround", # refers to the 'surround.py' file
 "reverse" # !!! refers to the 'reverse' function now !!!
]
```

```
def reverse(msg: str): # <-- this name shadows the 'reverse.py' submodule
 return msg[::-1] # in the case of a 'from sound.effects import *'
```

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from ... import` statement is executed (this also works when `__all__` is defined).

Although certain modules are designed to export only names that follow certain patterns, it is still considered bad practice in production code to use `from package import *`. Remember, there is nothing wrong with using `from package import specific_submodule`—this is the recommended way unless you need to use submodules with the same name from different packages.

## 6.4.2. Intra-package References¶

When packages are structured into subpackages (like with the `sound` package), you can use absolute imports to refer to submodules of sibling packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can do:

```
from sound.effects import echo
```

You can also write relative imports (using leading dots) to refer to sibling or parent packages, e.g., from the `surround` module:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use

absolute imports.

### 6.4.3. Packages in Multiple Directories¶

Packages support an attribute `__path__` which is initialized to a sequence of strings containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code, there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects that don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
```

```

'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10*3.25
>>> y = 200*200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, (\'spam\', \'eggs\'))'

```

The `string` module contains a `Template` class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals¶

**Formatted string literals** (also called `f-strings` for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds `pi` to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted. `!a` applies `ascii()`, `!s` applies `str()`, and `!r` applies `repr()`:

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```

>>> bugs = 'roaches'
>>> count = 13

```

```
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String `format()` Method ¶

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the method.

```
>>> print('{} and {}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument:

```
>>> print('This {food} is {adjective}'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be combined arbitrarily:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

If you have a long format string that you don't want to split up, it's possible to reference variables to be formatted by name instead of by position. This can be done by passing a dictionary and using square brackets `[]` to access the keys:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Alternatively, you can pass the dictionary as keyword arguments with the `**` notation:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k + '};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

```
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting¶

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

(Note that the space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. Similar methods include `str.ljust()` and `str.center()`. They do not write anything, only return a new string. If the input string is too long, they don't truncate it but return it unchanged.

There is also `str.zfill()`, which pads a numeric string on the left with zeros, understanding plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

### 7.1.4. Old string formatting¶

The `%` operator (modulo) can be used for string formatting. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
```



The value of `pi` is approximately 3.142.

More information can be found in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files¶

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument:

```
f = open('workfile', 'w', encoding='utf-8')
```

The first argument is a string containing the filename. The second argument is a string describing how the file will be used: `'r'` for read-only, `'w'` for write (existing file will be erased), `'a'` for append, and `'r+'` for read/write. The mode argument is optional; `'r'` is the default.

Normally, files are opened in text mode, meaning strings are read/written encoded in a specific encoding (default platform-dependent). Using `'b'` in mode opens the file in binary mode which reads/writes bytes objects. No encoding specification is allowed in binary mode.

In text mode, reading converts platform-specific line endings to `\n`, and writing converts `\n` back to platform-specific line endings. Binary mode works with bytes objects; be cautious when handling binary files like images or executables.

It is good practice to use the `with` statement when dealing with file objects:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
File is automatically closed after the block
```

Otherwise, you should call `f.close()` to close the file explicitly.

**Warning:** Calling `f.write()` without using `with` or `f.close()` might result in data not being fully written if the program exits prematurely.

Once a file is closed, using it again will cause errors:

```
f.close()
f.read() # Error: I/O operation on closed file
```

### 7.2.1. Methods of File Objects¶

Assuming a file object `f`:

- `f.read(size)` reads and returns up to `size` characters/bytes (or the entire file if omitted). Returns empty string/bytes at EOF.
- `f.readline()` reads one line at a time, ending with a newline `\n` (or empty string at EOF).
- Looping over `f` reads line by line efficiently:

```
for line in f:
 print(line, end='')
```

- `f.readlines()` returns a list of all remaining lines.
- `f.write(string)` writes a string to the file, returning number of characters written.

```
f.write('This is a test\n')
```

- Objects not strings need conversion:

```
value = ('the answer', 42)
s = str(value)
f.write(s)
```

- `f.tell()` returns the current position in the file, expressed as bytes in binary mode or an opaque number in text mode.
- `f.seek(offset, whence)` sets the file position, with `whence` values: `0` (start), `1` (current), `2` (end).

```
f.seek(5)
f.read(1) # Read 1 byte/character from position 5
f.seek(-3, 2) # Move 3 bytes before end
f.read(1)
```

In text files, seeking is limited: only relative to start or end is reliable.

## 7.2.2. Saving structured data with JSON¶

Strings can be easily saved. Numbers need serialization — converting to strings like `'123'` and parsing back with `int()`. For complex data like lists/dicts, use JSON.

```
import json
x = [1, 'simple', 'list']
json_str = json.dumps(x) # Serialize to JSON string
Output: '[1, "simple", "list"]'
```

Write to a file:

```
with open('data.json', 'w', encoding='utf-8') as f:
 json.dump(x, f)
```

Read back:

```
with open('data.json', 'r', encoding='utf-8') as f:
 x = json.load(f)
```

**Note:** JSON files must be encoded in UTF-8.

For serializing more complex objects, see the `json` module documentation. Alternatively, `pickle` can serialize arbitrarily complex Python objects but is insecure for untrusted sources.

[See also](#) - the pickle module.

---

*This completes the chapter on input and output in Python.*

# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`.

## Examples

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
```

```
>>> repr(s)
'Hello, world.'
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, (\'spam\', \'eggs\'))'
```

The [string](#) module contains a [Template](#) class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

## 7.1.1. Formatted String Literals

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```
import math
print(f'The value of pi is approximately {math.pi:.3f}.')
Output: The value of pi is approximately 3.142.
```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

Other modifiers can be used to convert the value before it is formatted:

- `!a` applies `ascii()`
- `!s` applies `str()`
- `!r` applies `repr()`

For example:

```
animals = 'eels'
print(f'My hovercraft is full of {animals}.') # Output: My hovercraft is f
print(f'My hovercraft is full of {animals!r}.') # Output: My hovercraft is f
```

The `=` specifier can be used to expand an expression to the text `expression`, then the `=` sign, then the representation of the evaluated expression:

```
bugs = 'roaches'
count = 13
area = 'living room'
print(f'Debugging {bugs=}, {count=}, {area=}')
Output: Debugging bugs='roaches', count=13, area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see [Format String Syntax](#).

## 7.1.2. The String `format()` Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
Output: We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the method.

Examples:

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
Output: spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
Output: eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument:

```
>>> print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely horr
Output: This spam is absolutely horrible
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
Output: The story of Bill, Manfred, and Georg.
```

You can reference variables to be formatted by name instead of by position by passing the dictionary directly and using square brackets `[]`:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(table))
Output: Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Alternatively, pass the dictionary as keyword arguments with `**`:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Output: Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is especially useful with `vars()`, which returns a dictionary of the current local symbol table:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = ' '.join([f'{k}: ' + '{' + f'{k}' + '}'; for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

And for nicely formatted tables:

```
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
Output:
1 1 1
2 4 8
3 9 27
```

```

4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

## 7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```

>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
Output:
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.

String methods like `str.rjust()`, `str.ljust()`, and `str.center()` can be used to pad strings to a specific width.

Another method, `str.zfill()`, pads a numeric string on the left with zeros, and handles plus and minus signs:

```

>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'

```

## 7.1.4. Old string formatting

The `%` operator (modulo) can be used for string formatting, a technique known as string interpolation:

```

>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
Output: The value of pi is approximately 3.142.

```

More information is available in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files

`open()` returns a file object, most commonly used with two positional arguments and one keyword argument:

```
f = open('workfile', 'w', encoding='utf-8')
```

- The first argument is the filename.
- The second argument is the mode ( `'r'`, `'w'`, `'a'`, `'r+'`, etc.).
- The optional `encoding` parameter specifies the text encoding.

Files are typically opened in text mode, which reads and writes strings encoded in a specific encoding. Binary mode ( `'b'` ) reads and writes bytes objects and does not support `encoding` .

It's best practice to use the `with` statement, which ensures automatic closure:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
File is automatically closed here
```

## Common file methods:

- `f.read(size)` reads data, returning a string or bytes.
- `f.readline()` reads a single line.
- Looping over the file object ( `for line in f:` ) reads line by line efficiently.
- `f.readlines()` returns all lines as a list.
- `f.write(string)` writes data to the file.

## Modifying file positions:

- `f.tell()` returns the current position.
- `f.seek(offset, whence)` changes position; `whence` defaults to 0.

## Example:

```
f = open('workfile', 'rb+')
f.write(b'0123456789abcdef')
f.seek(5)
print(f.read(1))
f.seek(-3, 2)
print(f.read(1))
```

## Saving structured data with `json`

Strings, numbers, lists, and dictionaries can be written/read easily. For complex data types, Python's `json` module (see [json](#)) serializes data hierarchies to strings (serialization) and reconstructs them back (deserialization).

```
import json
x = [1, 'simple', 'list']
json_str = json.dumps(x) # Convert to string

with open('file.json', 'w', encoding='utf-8') as f:
 json.dump(x, f)

Reading
with open('file.json', 'r', encoding='utf-8') as f:
 x_loaded = json.load(f)
```

Note: JSON files must be encoded in UTF-8. Serialization handles lists and dictionaries; for custom classes, additional effort is needed.

---

*For further details, consult the sections referenced above.*



# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
```

```

'Hello, world.'
>>> repr(s)
'"Hello, world."'
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, (\'spam\', \'eggs\'))'

```

The [string](#) module contains a [Template](#) class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals¶

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted:

- `!a` applies `ascii()`
- `!s` applies `str()`
- `!r` applies `repr()`

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=}, {count=}, {area=}')
Debugging bugs='roaches', count=13, area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String `format()` Method ¶

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say " {} !"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the method. A number in the brackets can be used to refer to the position of the object passed into the method:

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the method, their values are referred to by using the name of the argument:

```
>>> print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f"{{{k}}}: " + "{" + k + "};" for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting¶

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
... # Note use of 'end' on previous line
... print(repr(x**3).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything; they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column layout but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

### 7.1.4. Old string formatting¶

The `%` operator (modulo) can also be used for string formatting. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files¶

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument:

```
f = open('workfile', 'w', encoding='utf-8')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. `mode` can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), `'a'` for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The `mode` argument is optional; `'r'` is assumed if it's omitted.

Normally, files are opened in *text mode*, which means you read and write strings encoded in a specific *encoding*. If no `encoding` is specified, the default is platform dependent (see `open()`). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you need a different encoding. Appending `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as [bytes](#) objects. You cannot specify *encoding* in binary mode.

In text mode, reading converts platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert `\n` back to platform-specific line endings. This behind-the-scenes modification is fine for text files, but will corrupt binary data like in JPEG or EXE files. Use binary mode when handling such files.

It is good practice to use the [with](#) statement when working with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised. Using `with` is also much shorter than writing equivalent [try-finally](#) blocks:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
```

If you're not using `with`, you should call `f.close()` to close the file and immediately free system resources.

**Warning:** Calling `f.write()` without using `with` or calling `f.close()` might result in the arguments of `f.write()` not being completely written to disk, even if the program exits successfully.

Once a file object is closed, attempts to use it will fail:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

### 7.2.1. Methods of File Objects¶

The rest of this section assumes a file object `f` has been created.

- To read the file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

- `f.readline()` reads a single line from the file; a newline character `\n` is left at the end of the string, omitted only on the last line if the file doesn't end with newline. If it returns an empty string, the end of the file has been reached.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

- Loop over the file object for reading lines efficiently:

```
>>> for line in f:
... print(line, end='')
...
This is the first line of the file.
Second line of the file
```

- To read all lines into a list, use `list(f)` or `f.readlines()`.
- `f.write(string)` writes the contents of `string` to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

- Other objects need to be converted to string or bytes before writing:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

- `f.tell()` returns an integer indicating the current position in the file (number of bytes in binary mode) or an opaque number in text mode.
- To change position, use `f.seek(offset, whence)`, with `whence`:
  - 0 (from start of file)
  - 1 (from current position)
  - 2 (from end of file)

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # go to 6th byte
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 3rd byte before the end
13
```

```
>>> f.read(1)
b'd'
```

In text files, seeking is only allowed relative to the start, except to the very end ( `seek(0, 2)` ). Offsets must come from positions returned by `f.tell()` or zero. Other values may cause undefined behavior.

File objects have additional methods such as `isatty()` and `truncate()`, which are less frequently used; see the Library Reference.

## 7.2.2. Saving structured data with `json`

Strings can be easily written to and read from files. Numbers require extra effort, as the `read()` method returns strings, which must be converted using functions like `int()` .

For complex data types like nested lists and dictionaries, the `json` module simplifies serialization/deserialization. It converts Python data hierarchies into string representations ( `serialization` ) and reconstructs data from these strings ( `deserialization` ).

Example of serializing an object:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

To write JSON data to a file:

```
>>> json.dump(x, f)
```

To read and decode JSON data from a file:

```
>>> x = json.load(f)
```

**Note:** JSON files must be encoded in UTF-8. Use `encoding="utf-8"` when opening files.

Serializing lists and dictionaries is straightforward, but serializing arbitrary objects requires extra steps. The reference for the `json` module explains this.

See also [pickle](#), which allows serialization of complex Python objects, but is Python-specific and insecure if used with untrusted data.

# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use *formatted string literals*, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- You can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same as `repr()`.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
```



```

'''Hello, world.'''
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, ('spam', 'eggs'))'

```

The [string module](#) contains a [Template](#) class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted: `!a` applies [ascii\(\)](#), `!s` applies [str\(\)](#), and `!r` applies [repr\(\)](#):

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```

>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'

```

```
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` dot specifier. For a reference on these format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String format() Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the method:

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used, their values are referred to by using the name of the argument:

```
>>> print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be combined arbitrarily:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

To reference variables by name directly, pass a dictionary and use square brackets `[]`:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This can also be done by passing the `table` dictionary as keyword arguments with the `**` notation:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is useful with `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k + '};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

For example, here's code to produce a neatly aligned table of integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
```

```

6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

For a full overview, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting

Here's the same table formatted manually:

```

>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
...
' 1' ' 1' ' 1'
' 2' ' 4' ' 8'
' 3' ' 9' '27'
' 4' '16' '64'
' 5' '25' '125'
' 6' '36' '216'
' 7' '49' '343'
' 8' '64' '512'
' 9' '81' '729'
'10' '100' '1000'

```

Note: the remaining spaces are added automatically by the `print()` function, which inserts spaces between arguments.

String methods like `str.rjust()`, `str.ljust()`, and `str.center()` can be used to pad string objects to a specific width. For example:

```

>>> '3'.rjust(3)
' 3'

```

The `str.zfill()` method pads a numeric string on the left with zeros, understanding plus and minus signs:

```

>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'

```

### 7.1.4. Old string formatting

The `%` operator (modulo) can also be used for string formatting. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is known as string interpolation:

```

>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.

```

See [printf-style String Formatting](#) for more.

## 7.2. Reading and Writing Files

`open()` returns a [file object](#), most commonly used with two positional arguments and one keyword argument:

```
f = open('workfile', 'w', encoding='utf-8')
```

- The first argument is the filename as a string.
- The second argument is a mode: `'r'` (read), `'w'` (write, erasing existing files), `'a'` (append), or `'r+'` (read and write).
- The `encoding` parameter is optional; defaults depend on platform, but `'utf-8'` is recommended.

Files are normally opened in **text mode**, which reads/writes encoded strings. Binary mode ( `mode` ending with `'b'` ) reads/writes bytes objects; in binary mode, `encoding` cannot be specified.

In text mode, line endings are translated between platform-specific (e.g., `\n` in Unix, `\r\n` in Windows) and `\n`.

It's best practice to use the `with` statement when handling files, as it ensures files are closed properly:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
File is automatically closed here
```

If not using `with`, call `f.close()` to close the file.

### Warning:

Calling `f.write()` without `with` or `f.close()` might result in data not being saved until the buffer is flushed or the program exits. After closing, attempts to use the file produce errors:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

### 7.2.1. Methods of File Objects

Assuming a file object `f`, common methods include:

- `f.read(size)` – reads up to `size` characters or bytes; if omitted or negative, reads entire file.
- `f.readline()` – reads a single line; includes newline character unless EOF.
- Loop over `f`, e.g.:

```
for line in f:
 print(line, end='')
```

- `list(f)` or `f.readlines()` – reads all lines into a list.
- `f.write(string)` – writes string to file; returns number of characters written.
- `f.tell()` – current position as number of bytes (binary mode) or an opaque number (text mode).
- `f.seek(offset, whence)` – move position: `whence` 0 (start), 1 (current), 2 (end):

```
f.seek(5) # to 6th byte in file
f.seek(-3, 2) # 3 bytes before end
```

- `f.flush()` – flush buffer to disk.

- `f.truncate(size=None)` – truncate file to `size` bytes; if `size` is omitted, truncates at current position.

Additional methods like `isatty()` and `truncate()` are available; consult the Library Reference for details.

## 7.2.2. Saving structured data with `json`

You can write strings easily, but numbers and complex data types require serialization.

Python's `json` module allows serialization ("dump") and deserialization ("load") of data hierarchies, producing JSON-formatted strings or files.

Example:

```
import json
x = [1, 'simple', 'list']
json.dumps(x) # produces: '[1, "simple", "list"]'
```

Save to a file:

```
with open('file.json', 'w', encoding='utf-8') as f:
 json.dump(x, f)
```

Load from a file:

```
with open('file.json', 'r', encoding='utf-8') as f:
 x = json.load(f)
```

Note: JSON should be encoded in UTF-8. The `json` module handles lists and dictionaries well, but serializing arbitrary class instances requires custom handling.

See [JSON](#) for details.

**Also:** Consider [pickle](#) — suitable for serializing complex Python objects, but less secure and interoperable with other languages.

# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code, there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same as `repr()`.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
```

```

'Hello, world.'
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))
'(32.5, 40000, (\'spam\', \'eggs\'))'

```

The `string` module contains a `Template` class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals

**Formatted string literals** (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `<code>{expression}</code>`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies `ascii()`, `'!s'` applies `str()`, and `'!r'` applies `repr()`:

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```

>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'

```

```
>>> print(f'Debugging {bugs=}, {count=}, {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the reference guide for the [Format Specification Mini-Language](#).

## 7.1.2. The String `format()` Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say " {} !"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation.

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k + '};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print(f'{x:2d} {x*x:3d} {x*x*x:4d}')
1 1 1
2 4 8
3 9 27
4 16 64
5 25125
6 36216
7 49343
8 64512
9 81729
101001000
```



```

2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```

>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column layout but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```

>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'

```

### 7.1.4. Old string formatting

The `%` operator (modulo) can also be used for string formatting. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```

>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)

```

The value of `pi` is approximately 3.142.

More information can be found in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument:

```
open(filename, mode, encoding=None)
```

- The first argument is a string containing the filename.
- The second argument is another string containing a few characters describing the way in which the file will be used.
  - `'r'` when the file will only be read
  - `'w'` for only writing (an existing file with the same name will be erased)
  - `'a'` opens the file for appending; data written is added to the end
  - `'r+'` opens the file for both reading and writing

The `mode` argument is optional; `'r'` will be assumed if omitted.

Normally, files are opened in *text mode*, which means you read and write strings, encoded in a specific *encoding*. If *encoding* is not specified, the default is platform dependent (see `open()`). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless a different encoding is needed.

Appending `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as [bytes](#) objects. You cannot specify *encoding* in binary mode.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing, `\n` is converted back to platform-specific line endings. This conversion keeps text files consistent but can corrupt binary data like JPEG or EXE files. Use binary mode for such files.

It is good practice to use the [with](#) keyword when working with file objects. This ensures proper closing of the file after its suite finishes, even if exceptions occur. For example:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
```

```
Check if file is closed
print(f.closed)
Output: True
```

If not using `with`, call `f.close()` to free resources:

```
f.close()
```

## Methods of File Objects

Assuming a file object `f`:

- `f.read(size)` reads some bytes or characters from the file; with no argument, it reads the entire file.

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

- `f.readline()` reads a single line from the file.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

- Looping over the file object reads lines efficiently:

```
for line in f:
 print(line, end='')
```

- `f.readlines()` reads all lines into a list.
- `f.write(string)` writes a string to the file and returns the number of characters written.

```
>>> f.write('This is a test\n')
15
```

- To write other types, convert them to string first:

```
value = ('the answer', 42)
s = str(value)
f.write(s)
```

- `f.tell()` returns the current position in the file (byte offset in binary mode).
- `f.seek(offset, whence)` moves the file pointer:

```
f.seek(5) # Go to the 6th byte
f.read(1) # Read one byte
f.seek(-3, 2) # Go to the 3rd byte before end
f.read(1)
```

- `f.close()` closes the file explicitly.

Warning: writing after closing the file results in errors:

```
f.close()
f.read()
ValueError: I/O operation on closed file.
```

## Saving structured data with JSON

Strings can be written/read easily. Numbers require parsing or serializing. For complex data types like nested lists and dictionaries, use JSON:

```
import json

x = [1, 'simple', 'list']
json_str = json.dumps(x) # Serialize to string
Output: '[1, "simple", "list"]'

Saving to a file
with open('file.json', 'w', encoding='utf-8') as f:
 json.dump(x, f)
```

```
Loading from a file
with open('file.json', 'r', encoding='utf-8') as f:
 x_loaded = json.load(f)
```

Note: JSON files must be UTF-8 encoded.

For more complex objects, serialization with `json` can be extended or use `pickle`. The pickle module supports serializing arbitrarily complex Python objects but is Python-specific and insecure if loading data from untrusted sources.

---

# Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |

[Python](#) »

[3.13.5 Documentation](#)

[The Python Tutorial](#) »

## 7. Input and Output

---

Copyright: *Python Software Foundation, 2001-2025.*

This page is licensed under the [Python Software Foundation License Version 2](#). Examples and code are licensed under the Zero Clause BSD License.

[Donate](#)

*Last updated on Jul 05, 2025 (07:10 UTC).*

[Report a bug](#)

[Show source](#)

Created using [Sphinx](#) 8.2.3.

# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use **formatted string literals**, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code, there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {2:.2%}'.format(yes_votes, total_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign.

- You can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`.

### Examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
```

```
"'Hello, world.'"
```

```
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
```

The [string](#) module contains a [Template](#) class that offers yet another way to substitute values into strings, using placeholders like `$x`, but offers much less control of the formatting.

## 7.1.1. Formatted String Literals

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide, useful for making columns line up.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678
```

Other modifiers can be used to convert the value before it is formatted:

- `!a` applies `ascii()`
- `!s` applies `str()`
- `!r` applies `repr()`

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String format() Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets `{}` inside the string are format fields, which are replaced with the objects passed to `format()`. You can refer to objects by position with numbers, or by keyword.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Keyword arguments can also be used:

```
>>> print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

You can reference variables by name from a dictionary using square brackets:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Or, pass the dictionary as keyword arguments with `**`:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

You can access local variables directly with `vars()`:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f"{{{k}}}: " + "{" + k + "}" for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

And produce aligned columns:

```
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

For a complete overview, see [Format String Syntax](#).

## 7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

The `str.rjust()` method right-justifies a string in a field of a specified width. Similar methods include `ljust()` and `center()`. They do not truncate strings, only pad them.

The `str.zfill()` method pads a numeric string on the left with zeros, and understands plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

## 7.1.4. Old string formatting

The `%` operator can also be used for string formatting.

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

For more details, see [printf-style String Formatting](#).

## 7.2. Reading and Writing Files

`open()` returns a file object, commonly used with two positional arguments and one keyword argument:

```
f = open('workfile', 'w', encoding='utf-8')
```

- The first argument is the filename.
- The second argument is the mode:

- `'r'` for reading
- `'w'` for writing (erasing the file if it exists)
- `'a'` for appending
- `'r+'` for reading and writing



- `encoding` specifies the text encoding; recommended is `"utf-8"`.

Files are normally opened in text mode, converting platform-specific line endings (`\n`, `\r\n`) internally. Binary mode (`'b'`) reads/writes bytes directly and does not handle encoding.

Using the `with` statement ensures proper resource management:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
File is automatically closed after the block
```

Calling `f.close()` is needed if not using `with`.

Warning: writing without `with` or `close()` might prevent data from being written completely.

## Methods of File Objects

Assuming `f` is a file object:

- `f.read(size)` reads up to `size` characters or bytes, returning a string or bytes.
- `f.readline()` reads a single line, with a trailing `\n` unless EOF.
- Looping over `f` yields lines efficiently.
- `f.readlines()` reads all lines into a list.
- `f.write(string)` writes data; length of data returned.
- `f.tell()` returns the current position.
- `f.seek(offset, whence)` changes position based on `whence`:
  - 0: start of file
  - 1: current position
  - 2: end of file

## Saving structured data with json

Use the `json` module to serialize Python objects:

```
import json
x = [1, 'simple', 'list']
json_str = json.dumps(x)
Output: '[1, "simple", "list"]'
```

```
with open('data.json', 'w', encoding='utf-8') as f:
 json.dump(x, f)
```

# To load data:

```
with open('data.json', 'r', encoding='utf-8') as f:
 data = json.load(f)
```

Note: JSON files should be UTF-8 encoded. Serialization supports lists and dictionaries. For more complex objects, additional effort is needed.

See also: [pickle](#) for serializing arbitrary Python objects, but it is insecure for untrusted data.

# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {2:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
```

```

'Hello, world.'
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, (\'spam\', \'eggs\'))'

```

The [string](#) module contains a [Template](#) class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies [ascii\(\)](#), `'!s'` applies [str\(\)](#), and `'!r'` applies [repr\(\)](#):

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```

>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'

```

```
>>> print(f'Debugging {bugs=}, {count=}, and {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String `format()` Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument:

```
>>> print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = ' '.join([f'{k}: ' + '{' + k + '};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print(f'{x:2d} {x*x:3d} {x*x*x:4d}')
...
```

|    |     |      |
|----|-----|------|
| 1  | 1   | 1    |
| 2  | 4   | 8    |
| 3  | 9   | 27   |
| 4  | 16  | 64   |
| 5  | 25  | 125  |
| 6  | 36  | 216  |
| 7  | 49  | 343  |
| 8  | 64  | 512  |
| 9  | 81  | 729  |
| 10 | 100 | 1000 |

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
... # Note use of 'end' on previous line
... print(repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

### 7.1.4. Old string formatting

The `%` operator (modulo) can also be used for string formatting. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding='utf-8')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. `mode` can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The `mode` argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, meaning you read and write strings from and to the file, which are encoded in a specific *encoding*. If `encoding` is not specified, the default is platform dependent. Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know you need a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as [bytes](#). You cannot specify `encoding` when opening a file in binary mode.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the [with](#) keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised. Using `with` is also much shorter than writing equivalent [try-finally](#) blocks:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
```

You can check that the file has been automatically closed:

```
>>> f.closed
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Warning: Calling `f.write()` without using the `with` keyword or calling `f.close()` *might* result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

## Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). `size` is an optional numeric argument. When `size` is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most `size` characters (in text mode) or `size` bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
... print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file into a list, you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of `string` to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted—either to a string (in text mode) or a bytes object (in binary mode)—before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file, represented as the number of bytes from the beginning of the file in binary mode, or an opaque number in text mode.

To change the file object's position, use `f.seek(offset, whence)`. The position is computed by adding `offset` to a reference point; the reference point is selected by the `whence` argument. A `whence` value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. `whence` can be omitted, defaulting to 0.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (opened without `'b'` mode), only seeks relative to the beginning of the file are allowed (except seeking to the very end with `seek(0, 2)`), and the only valid offset values are those returned from `f.tell()` or zero. Any other offset produces undefined behavior.

File objects have additional methods such as `isatty()` and `truncate()`, which are less frequently used; consult the Library Reference for a complete guide.

## 7.2.2. Saving structured data with `json`

Strings can easily be written to and read from a file. Numbers take a bit more effort since the `read()` method only returns strings, which will have to be passed to functions like `int()`, which take a string like `'123'` and return its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly write and debug code to save complicated data types, Python offers the `json` module, which uses the popular JSON (JavaScript Object Notation) format. The standard module can convert Python data hierarchies to string representations (serialization) and reconstruct the data from the string (deserialization). The string can be stored in a file or transmitted over a network connection.

Note: JSON files must be encoded in UTF-8. Use `encoding="utf-8"` when opening JSON files for both reading and writing.

Example:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

To serialize to a file:

```
>>> json.dump(x, f)
```

To decode the object again:

```
>>> x = json.load(f)
```

For more details, see [the json module documentation](#).



# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Examples:

```
>>> s = 'Hello, world.'
>>> str(s)
```

```

'Hello, world.'
>>> repr(s)
'"Hello, world."'
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, (\'spam\', \'eggs\'))'

```

The `string` module contains a `Template` class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals

**Formatted string literals** (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted. `!a` applies `ascii()`, `!s` applies `str()`, and `!r` applies `repr()`:

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=}, {count=}, {area=}')
Debugging bugs='roaches', count=13, area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String `format()` Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}"!'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `[]` to access the keys:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}'.format(table))
Jack: 4098; Sjoerd: 4127
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}'.format(**table))
Jack: 4098; Sjoerd: 4127
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f"{{{k}}}: {" for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print(f'{x:2d} {x*x:3d} {x*x*x:4d}')
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

### 7.1.4. Old string formatting

The `%` operator (modulo) can also be used for string formatting. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument:

```
>>> f = open('workfile', 'w', encoding='utf-8')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. `mode` can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The `mode` argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific *encoding*. If *encoding* is not specified, the default is platform dependent (see [open\(\)](#)). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know that you need to use a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as [bytes](#) objects. You can not specify *encoding* when opening file in binary mode.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the [with](#) keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent [try-finally](#) blocks:

```
>>> with open('workfile', encoding='utf-8') as f:
... read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Warning: Calling `f.write()` without using the `with` keyword or calling `f.close()` *might* result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
```

```
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

## 7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* characters (in text mode) or *size* bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
... print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *whence* argument. A *whence* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *whence* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very end with `seek(0, 2)`) and the only valid *offset* values are those returned from `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

File objects have some additional methods, such as `isatty()` and `truncate()`, which are less frequently used; consult the Library Reference for a complete guide to file objects.

## 7.2.2. Saving structured data with json

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called **JSON (JavaScript Object Notation)**. The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

### Note:

The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

Another variant of the `json.dumps()` function, called `json.dump()`, simply serializes the object to a **text file**. So if `f` is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if `f` is a **binary file** or **text file** object which has been opened for reading:

```
x = json.load(f)
```

**Note:**

JSON files must be encoded in UTF-8. Use `encoding="utf-8"` when opening JSON file as a [text file](#) for both reading and writing.

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the [json](#) module contains an explanation of this.

**See also:**

[pickle](#) - the pickle module

Contrary to [JSON](#), [pickle](#) is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.



# 7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

## 7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code there are two examples of how to format variables:

```
>>> yes_votes = 42572654
>>> total_votes = 85705149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a [SyntaxError](#) if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
```

```

>>> repr(s)
'"Hello, world."'
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs'))
'"(32.5, 40000, ('spam', 'eggs'))"

```

The [string](#) module contains a [Template](#) class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

### 7.1.1. Formatted String Literals

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```

>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.

```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
... print(f'{name:10} ==> {phone:10d}')
...
Sjoerd ==> 4127
Jack ==> 4098
Dcab ==> 7678

```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies [ascii\(\)](#), `'!s'` applies [str\(\)](#), and `'!r'` applies [repr\(\)](#):

```

>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.

```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=}, {count=}, {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the [Format Specification Mini-Language](#).

## 7.1.2. The String `format()` Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}'.format(food='spam', adjective='absolutely horr
This spam is absolutely horrible.
```

Positional and keyword arguments can be combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg
The story of Bill, Manfred, and Georg.
```

For long format strings you can reference variables by name instead of position by passing the dictionary and using square brackets `[]`:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d};'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This can also be done by passing the dictionary as keyword arguments with `**`:

```
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Using `vars()`, which returns a dictionary of all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k + '};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

Here's an example to produce aligned columns with integers and their squares and cubes:

```
>>> for x in range(1, 11):
... print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1 1 1
```

```

2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

For a complete overview of string formatting with `str.format()`, see [Format String Syntax](#).

### 7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```

>>> for x in range(1, 11):
... print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
... # Note use of 'end' on previous line
... print(repr(x*x*x).rjust(4))
...
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method right-justifies a string in a field of a given width by padding it with spaces on the left. Similar methods include `str.ljust()` and `str.center()`. These methods do not write anything; they return a new string. If the input string is too long, they don't truncate it, but return it unchanged.

The `str.zfill()` method pads a numeric string on the left with zeros and understands about plus and minus signs:

```

>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'

```

### 7.1.4. Old string formatting

The `%` operator (modulo) can also be used for string formatting.

Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the [printf-style String Formatting](#) section.

## 7.2. Reading and Writing Files

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument:

```
open(filename, mode, encoding=None)
```

Example:

```
>>> f = open('workfile', 'w', encoding='utf-8')
```

The first argument is a string containing the filename. The second argument is another string describing how the file will be used:

- `'r'` — open for reading (default)
- `'w'` — open for writing, erasing the file if it exists
- `'a'` — open for appending; data written is added to the end
- `'r+'` — open for both reading and writing

The `mode` argument is optional; `'r'` is assumed if omitted.

Files are normally opened in **text mode**, which means strings are read and written, encoded in an `encoding`. If `encoding` is not specified, platform-specific defaults are used (see [open\(\)](#)). UTF-8 is the recommended encoding.

Appending `'b'` to the mode opens the file in **binary mode**. Data read and written in binary mode are `bytes` objects. In binary mode, `encoding` cannot be specified.

In text mode, line endings are automatically translated: `\n` on Unix, `\r\n` on Windows. When writing in text mode, `\n` is converted back to the platform-specific line ending. This behavior is fine for text files but should be avoided with binary data such as images or executables.

It is best practice to use the [with](#) statement when working with files, which ensures proper closing:

```
with open('workfile', encoding='utf-8') as f:
 read_data = f.read()
After this block, the file is automatically closed
```

If you don't use `with`, explicitly call `f.close()` to free system resources.

Warning: calling `f.write()` without `with` or `f.close()` might result in data not being completely written.

Read and write methods:

- `f.read(size)` — reads up to `size` characters/bytes (full content if omitted)
- `f.readline()` — reads one line, ending with `\n` (or empty string at EOF)
- looping over `f` yields lines efficiently
- `f.readlines()` — reads all lines into a list
- `f.write(string)` — writes string, returns number of characters/bytes

Other methods include:

- `f.tell()` — current position

- `f.seek(offset, whence)` — move position; `whence` =0 (start), 1 (current), 2 (end)

## 7.2.2. Saving structured data with json

Strings are easy to save/read, but numbers and complex data structures require serialization. Python's `json` module handles this by converting data to a string format.

Example:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

To save to a file:

```
json.dump(x, f)
```

To load from a file:

```
x = json.load(f)
```

Note: JSON files should be UTF-8 encoded. Use `encoding='utf-8'` when opening files.

JSON handles lists and dictionaries but not arbitrary class instances unless customized. The [json](#) documentation covers advanced support.

See also: [pickle](#) — serializes complex objects but is insecure and platform-dependent.

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>`) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 10 * (1/0)
 ^^^
ZeroDivisionError: division by zero
```

```
>>> 4 + spam * 3
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 4 + spam * 3
 ^^^^^
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 '2' + 2
 ^
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and

`TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
```

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try ... except` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, *TypeError, *NameError):
... pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
 pass

class C(B):
 pass

class D(C):
 pass
```



```

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
 except C:
 print("C")
 except B:
 print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, built-in exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit`, which is raised by `sys.exit()` and `KeyboardInterrupt`, which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:

```

```

 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise

```

The `try...except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```

import sys
for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()

```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try...except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```

>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero

```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 raise NameError('HiThere')
NameError: HiThere

```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise
```

## 8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")
```

→ Output:

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 open("database.sqlite")
~~~~~
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func():
    raise ConnectionError
...
try:
    func()
except ConnectionError as exc:
    raise RuntimeError('Failed to open database') from exc
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
    open('database.sqlite')
except OSError:
    raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue` or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue` or `return` statement, the `finally` clause will execute just prior to the `break`, `continue` or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

For example:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
  File "<stdin>", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       f()
|       ~^^
|   File "<stdin>", line 3, in f
|       raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
|   OSError: error 1
+----- 2 -----
|   SystemError: error 2
+-----
```

And catching it:

```
try:
    f()
except Exception as e:
    print(f'caught {type(e)}: {e}')
```

→ Output:

```
caught <class 'ExceptionGroup'>: there were problems (2 sub-exceptions)
```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be re-raised.

```
def f():
    raise ExceptionGroup(
        "group1",
        [
            OSError(1),
            SystemError(2),
            ExceptionGroup(
                "group2",
                [
                    OSError(3),
                    RecursionError(4)
                ]
            )
        ]
    )
```

```

        )
    ]
)

try:
    f()
except* OSError as e:
    print("There were OSError's")
except* SystemError as e:
    print("There were SystemErrors")

```

→ Output:

```

There were OSError's
There were SystemErrors

```

Note that the exceptions nested in an exception group must be instances, not types. This is because the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```

excs = []
for test in tests:
    try:
        test.run()
    except Exception as e:
        excs.append(e)

if excs:
    raise ExceptionGroup("Test Failures", excs)

```

## 8.10. Enriching Exceptions with Notes

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

```

>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information

```

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred.

```

def f():
    raise OSError('operation failed')
excs = []
for i in range(3):
    try:
        f()
    except Exception as e:
        e.add_note(f'Happened in Iteration {i+1}')
        excs.append(e)
raise ExceptionGroup('We have some problems', excs)
# + Exception Group Traceback (most recent call last):
# |   File "<stdin>", line 1, in <module>
# |       raise ExceptionGroup('We have some problems', excs)
# |       ~^^
# |   File "<stdin>", line 3, in f
# |       raise OSError('operation failed')
# | OSError: operation failed
# | Happened in Iteration 1
# +----- 2 -----
# | Traceback (most recent call last):
# |   File "<stdin>", line 3, in f
# |       raise OSError('operation failed')
# | OSError: operation failed
# | Happened in Iteration 2
# +----- 3 -----
# | Traceback (most recent call last):
# |   File "<stdin>", line 3, in f
# |       raise OSError('operation failed')
# | OSError: operation failed
# | Happened in Iteration 3
# +-----

```



## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
      File "<stdin>", line 1
        while True print('Hello world')
                                ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are [ZeroDivisionError](#), [NameError](#), and [TypeError](#). The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using **Control-C** or whatever the operating system supports); note that a user-generated interruption is signalled by raising the [KeyboardInterrupt](#) exception.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number. Try again...")
... 
```

The [try](#) statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
```

```

        print("D")
    except C:
        print("C")
    except B:
        print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__` to print all the arguments without explicitly accessing `args`.

```

>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst)) # the exception type
...     print(inst.args)  # arguments stored in .args
...     print(inst)       # __str__ allows args to be printed directly,
...                         # but may be overridden in exception subclasses
...     x, y = inst.args   # unpack args
...     print('x =', x)
...     print('y =', y)

```

The exception's `__str__` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise

```

The `try ... except` statement has an optional *else clause*, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```
import sys

for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise
```

## 8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
    open("database.sqlite")
except OSError:
    raise RuntimeError("unable to handle error")
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func():
    raise ConnectionError
...

try:
    func()
except ConnectionError as exc:
    raise RuntimeError('Failed to open database') from exc
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
    open('database.sqlite')
except OSError:
    raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue` or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the `break`, `continue`, or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

For example:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
```

```
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
def f():
    excs = [OSError('error 1'), SystemError('error 2')]
    raise ExceptionGroup('there were problems', excs)

try:
    f()
```

```
except Exception as e:
    print(f'caught {type(e)}: {e}')
```

Output:

```
caught <class 'ExceptionGroup'>: (2 sub-exceptions)
└─ ExceptionGroup: there were problems (2 sub-exceptions)
  └─ OSError: error 1
    └─ SystemError: error 2
```

Similarly, you can catch specific exception types within an `ExceptionGroup` using `except*`:

```
def f():
    raise ExceptionGroup(
        "group1",
        [
            OSError('error 1'),
            SystemError('error 2'),
            ExceptionGroup(
                "group2",
                [
                    OSError(3),
                    RecursionError(4)
                ]
            )
        ]
    )

try:
    f()
except* OSError as e:
    print("There were OSErrors")
except* SystemError as e:
    print("There were SystemErrors")
```

Output:

```
There were OSErrors
There were SystemErrors
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
excs = []
for test in tests:
    try:
        test.run()
    except Exception as e:
        excs.append(e)
if excs:
    raise ExceptionGroup("Test Failures", excs)
```

## 8.10. Enriching Exceptions with Notes

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught.



For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred.

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       raise ExceptionGroup('We have some problems', excs)
|   +---+----- 1 -----
|       | Traceback (most recent call last):
|       |   File "<stdin>", line 3, in <module>
|       |       raise OSError('operation failed')
|       |   operation failed
|       |   Happend in Iteration 1
|       +----- 2 -----
|       | Traceback (most recent call last):
|       |   File "<stdin>", line 3, in <module>
|       |       raise OSError('operation failed')
|       |   operation failed
|       |   Happend in Iteration 2
|       +----- 3 -----
|       | Traceback (most recent call last):
|       |   File "<stdin>", line 3, in <module>
|       |       raise OSError('operation failed')
|       |   operation failed
|       |   Happend in Iteration 3
|   +-----
```

This feature allows programs to provide much richer diagnostics after catching exceptions, especially when collecting multiple errors.

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors¶

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
      File "<stdin>", line 1
        while True print('Hello world')
                                ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `[print()](../library/functions.html#print)`, since a colon ( `:` ) is missing just before it.

The file name ( `<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions¶

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
    ~^^
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
    ~~~~^~~
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 '2' + 2
 ~
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are [ZeroDivisionError](#), [NameError](#), and [TypeError](#). The string printed as the exception type is the name of the built-in exception that occurred. This is true for all

built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions¶

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using *Control-C* or whatever the operating system supports); note that a user-generated interruption is signalled by raising the [KeyboardInterrupt](#) exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The [try](#) statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most, one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
except (RuntimeError, TypeError, NameError):
 pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception): pass
class C(B): pass
class D(C): pass
```

```
for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
```

```

except C:
 print("C")
except B:
 print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")

```

```
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise
```

The `try ... except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()
```

The use of the *else* clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions¶

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise
```

## 8.5. Exception Chaining ¶

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")
```

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 open("database.sqlite")
FileNotFoundError: [Errno 2] No such file or directory:
'database.sqlite'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
... raise ConnectionError
...
>>> try:
... func()
... except ConnectionError as exc:
... raise RuntimeError('Failed to open database') from exc
...
```

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 func()
 File "<stdin>", line 2, in func
ConnectionError
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the `from` idiom:

```
try:
 open('database.sqlite')
except OSError:
 raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions¶

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions¶

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
 raise KeyboardInterrupt
finally:
 print('Goodbye, world!')
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue`, or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the `break`, `continue`, or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

For example:

```
>>> def bool_return():
... try:
... return True
... finally:
... return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
... try:
... result = x / y
... except ZeroDivisionError:
```

```

... print("division by zero!")
... else:
... print("result is", result)
... finally:
... print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 divide("2", "1")
 File "<stdin>", line 3, in divide
 result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions ¶

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```

for line in open("myfile.txt"):
 print(line, end="")

```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```

with open("myfile.txt") as f:
 for line in f:
 print(line, end="")

```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions ¶

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use



cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
... excs = [OSError('error 1'), SystemError('error 2')]
... raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| f()
| ~^^
| File "<stdin>", line 3, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----
```

You can catch an `ExceptionGroup` similarly:

```
try:
 f()
except Exception as e:
 print(f'caught {type(e)}: {e}')
```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be reraised:

```
def f():
 raise ExceptionGroup(
 "group1",
 [
 OSError("1"),
 SystemError("2"),
 ExceptionGroup(
 "group2",
 [
 OSError("3"),
 RecursionError("4")
]
)
]
)

try:
 f()
except* OSError as e:
 print("There were OSErrors")
except* SystemError as e:
 print("There were SystemErrors")
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
excs = []
for test in tests:
 try:
 test.run()
 except Exception as e:
 excs.append(e)
if excs:
 raise ExceptionGroup("Test Failures", excs)
```

## 8.10. Enriching Exceptions with Notes ¶

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

```
>>> try:
... raise TypeError('bad type')
... except Exception as e:
... e.add_note('Add some information')
... e.add_note('Add some more information')
... raise
...
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
```

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + spam
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are **ZeroDivisionError**, **NameError**, and **TypeError**. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the `try/except` block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
except (RuntimeError, TypeError, NameError):
 pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print `B`, `C`, `D` in that order:

```
class B(Exception):
 pass

class C(B):
 pass

class D(C):
 pass

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
```

```

 print("D")
 except C:
 print("C")
 except B:
 print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed `B`, `B`, `B` — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, built-in exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")

```

```
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise
```

The `try ... except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the *try* clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()
```

The use of the *else* clause is better than adding additional code to the *try* clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
... x = 1/0
>>>
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise
```

## 8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")

Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise ... from` clause can be used:

```
exc must be exception instance or None.
raise RuntimeError('Failed to open database') from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func():
 raise ConnectionError

try:
 func()
except ConnectionError as exc:
 raise RuntimeError('Failed to open database') from exc
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
 open('database.sqlite')
except OSError:
 raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
 raise KeyboardInterrupt
finally:
 print('Goodbye, world!')
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue` or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the `break`, `continue`, or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

For example:

```
def bool_return():
 try:
 return True
 finally:
 return False

print(bool_return()) # Outputs: False
```

A more complicated example:

```
def divide(x, y):
 try:
 result = x / y
 except ZeroDivisionError:
 print("division by zero!")
 else:
 print("result is", result)
 finally:
 print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
```



```

executing finally clause
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 divide("2", "1")
 File "<stdin>", line 3, in divide
 result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen:

```

for line in open("myfile.txt"):
 print(line, end="")

```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```

with open("myfile.txt") as f:
 for line in f:
 print(line, end="")

```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```

>>> def f():
... excs = [OSError('error 1'), SystemError('error 2')]
... raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| f()

```

```

| ~^^
| File "<stdin>", line 3, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----

```

```

>>> try:
... f()
... except Exception as e:
... print(f'caught {type(e)}: {e}')
...
caught <class 'ExceptionGroup'>: there were problems (2 sub-exceptions)

```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be re-raised:

```

>>> def f():
... raise ExceptionGroup(
... "group1",
... [
... OSError(1),
... SystemError(2),
... ExceptionGroup(
... "group2",
... [
... OSError(3),
... RecursionError(4)
...]
...)
...]
...)
...
>>> try:
... f()
... except* OSError as e:
... print("There were OSErrors")
... except* SystemError as e:
... print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 2, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise ExceptionGroup(
| ...<12 lines>...
|)
| ExceptionGroup: group1 (1 sub-exception)

```

```

+-+----- 1 -----
| ExceptionGroup: group2 (1 sub-exception)
+-+----- 1 -----
| OSError: 3
+----- 2 -----
| RecursionError: 4
+-----

```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```

excs = []
for test in tests:
 try:
 test.run()
 except Exception as e:
 excs.append(e)

if excs:
 raise ExceptionGroup('Test Failures', excs)

```

## 8.10. Enriching Exceptions with Notes

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred:

```

>>> def f():
... raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
... try:
... f()
... except Exception as e:
... e.add_note(f'Happened in Iteration {i+1}')
... excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~^^
| File "<stdin>", line 3, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~^^
| File "<stdin>", line 2, in f
| raise OSError('operation failed')
| Happened in Iteration 1

```

```

+----- 2 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~^^
| File "<stdin>", line 2, in f
| raise OSError('operation failed')
| Happened in Iteration 2
+----- 3 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~^^
| File "<stdin>", line 2, in f
| raise OSError('operation failed')
| Happened in Iteration 3

```

This concludes the overview of errors and exceptions in Python.

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most, one handler is executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
except (RuntimeError, TypeError, NameError):
 pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an `except` clause listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
 pass

class C(B):
 pass

class D(C):
 pass

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
 except C:
 print("C")
```

```
except B:
 print("B")
```

Note that if the `except` clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching `except` clause is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, built-in exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```
>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__() allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```
import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise
```

The `try ... except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise
```

## 8.5. Exception Chaining



If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
>>> try:
... open('database.sqlite')
... except OSError:
... raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError("unable to handle error")
...
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
... raise ConnectionError
...
>>> try:
... func()
... except ConnectionError as exc:
... raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 File "<stdin>", line 2, in func
ConnectionError
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
 open('database.sqlite')
except OSError:
 raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
... raise KeyboardInterrupt
... finally:
... print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. Points for more complex cases:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue`, or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the `break`, `continue`, or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

Example:

```
>>> def bool_return():
... try:
... return True
... finally:
... return False
...
>>> bool_return()
False
```

More complex example:

```
>>> def divide(x, y):
... try:
```

```

... result = x / y
... except ZeroDivisionError:
... print("division by zero!")
... else:
... print("result is", result)
... finally:
... print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

## 8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. For example, to print the contents of a file:

```

for line in open("myfile.txt"):
 print(line, end="")

```

The problem with this code is that it leaves the file open for an indeterminate amount of time. To ensure it's always closed promptly and correctly, use the `with` statement:

```

with open("myfile.txt") as f:
 for line in f:
 print(line, end="")

```

After the statement is executed, the file `f` is always closed, even if an error occurs while processing the lines. Objects which provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions

Sometimes it is necessary to report several exceptions that have occurred. This is common in concurrency frameworks, where multiple tasks may fail in parallel, or other scenarios where continuing execution and collecting multiple errors is desirable.

The built-in `ExceptionGroup` wraps a list of exception instances so they can be raised together. It is an exception itself, so it can be caught like any other exception.

```

>>> def f():
... excs = [OSError('error 1'), SystemError('error 2')]
... raise ExceptionGroup('there were problems', excs)
...
>>> f()

```

```

+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----

>>> try:
... f()
... except Exception as e:
... print(f'caught {type(e)}: {e}')
...
caught <class 'ExceptionGroup'>: (2 sub-exceptions)

```

By using `except*` instead of `except`, we can selectively handle **only** exceptions in the group that match certain types. For example, in a nested exception group, each `except*` clause extracts exceptions of a certain type for handling:

```

>>> def f():
... raise ExceptionGroup(
... "group1",
... [
... OSError(1),
... SystemError(2),
... ExceptionGroup(
... "group2",
... [
... OSError(3),
... RecursionError(4),
...]
...)
...]
...)
...
>>> try:
... f()
... except* OSError as e:
... print("There were OSErrors")
... except* SystemError as e:
... print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 2, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise ExceptionGroup(...)
| ExceptionGroup: group1 (3 sub-exceptions)
+----- 1 -----

```

```

 | OSError: error 1
+-+----- 2 -----
 | SystemError: error 2
+-+----- 3 -----
 | ExceptionGroup: group2 (2 sub-exceptions)
 | +-+----- 1 -----
 | | | OSError: error 3
 | +-+----- 2 -----
 | | | RecursionError: 4
 | +-+-----
+-+-----

```

Note that nested exceptions in an exception group must be instances, not types, because they are typically raised and caught, not declared as types.

For example:

```

excs = []
for test in tests:
 try:
 test.run()
 except Exception as e:
 excs.append(e)

if excs:
 raise ExceptionGroup("Test Failures", excs)

```

## 8.10. Enriching Exceptions with Notes

When an exception is created to be raised, it is usually initialized with information describing the error. Sometimes, it is useful to add more information after the exception was caught, using the `add_note(note)` method, which adds a string to the exception's notes list. The traceback includes all notes after the exception.

```

>>> try:
... raise TypeError('bad type')
... except Exception as e:
... e.add_note('Add some information')
... e.add_note('Add some more information')
... raise
...
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information

```

For example, when collecting exceptions into an exception group, you might want to add context information for individual errors:

```

>>> def f():
... raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
... try:

```

```

... f()
... except Exception as e:
... e.add_note(f'Happened in Iteration {i+1}')
... excs.append(e)
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~^^
| File "<stdin>", line 3, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~^^
| File "<stdin>", line 2, in f
| raise OSError('operation failed')
| OSError: operation failed
| Happened in Iteration 1
+----- 1 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| f()
| ~^^
| OSError: operation failed
| Happened in Iteration 1
+----- 2 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| f()
| ~^^
| OSError: operation failed
| Happened in Iteration 2
+----- 3 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| f()
| ~^^
| OSError: operation failed
| Happened in Iteration 3
+-----

```

---

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 10 * (1/0)
 ^^^
ZeroDivisionError: division by zero
```

```
>>> 4 + spam*3
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 4 + spam*3
 ^^^^^
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 '2' + 2
 ^^^
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and

`TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. It typically contains a list of source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
... pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print `B`, `C`, `D` in that order:

```
class B(Exception): pass
```

```
class C(B): pass
```

```
class D(C): pass
```

```
for cls in [B, C, D]:
 try:
```



```

 raise cls()
 except D:
 print("D")
 except C:
 print("C")
 except B:
 print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed `B` three times — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`:

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit`, which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:
 print("OS error:", err)

```

```
except ValueError:
 print("Could not convert data to an integer.")
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise
```

The `try ... except` statement has an optional *else clause*, which, when present, must follow all *except clauses*. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
... x = 1 / 0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
 raise NameError('HiThere')
except NameError:
```

```
print('An exception flew by!')
raise
```

This will print `'An exception flew by!'` and then re-raise the same exception, propagating it upward.

## 8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")
```

This produces:

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 open("database.sqlite")
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when transforming exceptions. For example:

```
def func():
 raise ConnectionError
...
try:
 func()
except ConnectionError as exc:
 raise RuntimeError('Failed to open database') from exc
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
 open('database.sqlite')
except OSError:
 raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or

indirectly.

Exception classes can do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in "Error", similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
 raise KeyboardInterrupt
finally:
 print('Goodbye, world!')
```

If a `finally` clause is present, it will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception.

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If not handled, the exception is re-raised after the `finally` clause.
- An exception could occur during execution of an `except` or `else` clause, and again, the exception is re-raised after the `finally`.
- If the `finally` clause executes a `break`, `continue`, or `return` statement, exceptions are not re-raised.

For example:

```
def bool_return():
 try:
 return True
 finally:
 return False

print(bool_return()) # Outputs: False

def divide(x, y):
 try:
 result = x / y
 except ZeroDivisionError:
 print("division by zero!")
 else:
 print("result is", result)
 finally:
 print("executing finally clause")
```

---

*(The rest of the document continues with more sections and examples.)*

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors ¶

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions ¶

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 10 * (1/0)
 ^^^
ZeroDivisionError: division by zero
```

```
>>> 4 + spam*3
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 4 + spam*3
 ^^^^^
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 '2' + 2
 ^^^
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and

`TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

`Built-in Exceptions` lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions ¶

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
except (RuntimeError, TypeError, NameError):
 pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an `except` clause listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
 pass

class C(B):
 pass

class D(C):
 pass
```

```

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
 except C:
 print("C")
 except B:
 print("B")

```

Note that if the `except` clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching `except` clause is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()

```

```

 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise

```

The `try ... except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the *try* clause does not raise an exception. For example:

```

for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()

```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```

>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero

```

## 8.4. Raising Exceptions ¶

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere

```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:



```
try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise
```

## 8.5. Exception Chaining ¶

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")

Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 open("database.sqlite")
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
... raise ConnectionError
...
>>> try:
... func()
... except ConnectionError as exc:
... raise RuntimeError('Failed to open database') from exc
...

Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 func()
 File "<stdin>", line 2, in func
ConnectionError
The above exception was the direct cause of the following exception:
```

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the `from` idiom:

```
try:
 open('database.sqlite')
except OSError:
 raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions ¶

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the [Exception](#) class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions ¶

The [try](#) statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
 raise KeyboardInterrupt
finally:
 print('Goodbye, world!')
```

If a [finally](#) clause is present, the [finally](#) clause will execute as the last task before the [try](#) statement completes. The [finally](#) clause runs whether or not the [try](#) statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the [try](#) clause, the exception may be handled by an [except](#) clause. If the exception is not handled by an [except](#) clause, the exception is re-raised after the [finally](#) clause has been executed.
- An exception could occur during execution of an [except](#) or [else](#) clause. Again, the exception is re-raised after the [finally](#) clause has been executed.
- If the [finally](#) clause executes a [break](#), [continue](#), or [return](#) statement, exceptions are not re-raised.
- If the [try](#) statement reaches a [break](#), [continue](#), or [return](#) statement, the [finally](#) clause will execute just prior to the [break](#), [continue](#), or [return](#) statement’s execution.
- If a [finally](#) clause includes a [return](#) statement, the returned value will be the one from the [finally](#) clause’s [return](#) statement, not the value from the [try](#) clause’s [return](#) statement.

For example:

```
>>> def bool_return():
... try:
... return True
... finally:
... return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
... try:
... result = x / y
... except ZeroDivisionError:
... print("division by zero!")
... else:
... print("result is", result)
... finally:
... print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 divide("2", "1")
 File "<stdin>", line 3, in divide
 result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real-world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions ¶

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen:

```
for line in open("myfile.txt"):
 print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
 for line in f:
 print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions ¶

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The built-in `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
... excs = [OSError('error 1'), SystemError('error 2')]
... raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| f()
| ~^^
| File "<stdin>", line 3, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----
```

```
>>> try:
... f()
... except Exception as e:
... print(f'caught {type(e)}: {e}')
...
caught <class 'ExceptionGroup'>: (2 sub-exceptions)
```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be re-raised:

```
>>> def f():
... raise ExceptionGroup(
... "group1",
... [
... OSError(1),
... SystemError(2),
... ExceptionGroup(
... "group2",
... [
... OSError(3),
... RecursionError(4)
...]
...)
...]
...)
```

```

...)
...
>>> try:
... f()
... except* OSError as e:
... print("There were OSErrors")
... except* SystemError as e:
... print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors

```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```

excs = []
for test in tests:
 try:
 test.run()
 except Exception as e:
 excs.append(e)

if excs:
 raise ExceptionGroup("Test Failures", excs)

```

## 8.10. Enriching Exceptions with Notes ¶

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred:

```

>>> def f():
... raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
... try:
... f()
... except Exception as e:
... e.add_note(f'Happened in Iteration {i+1}')
... excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~~~~~
| File "<stdin>", line 2, in <module>
| raise ExceptionGroup('We have some problems', excs)

```

```

| ~~~~~
| ExceptionGroup: We have some problems (3 sub-exceptions)
+----- 1 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 1
+----- 2 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 2
+----- 3 -----
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 3

```

*Note:* The above Markdown is a simplified representation derived from the provided HTML content, focusing on the main explanations and code examples for clarity.

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^^
```

SyntaxError: invalid syntax

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the `print()` function, since a colon ( `:` ) is missing just before it.

The file name ( `<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 10 * (1/0)
 ^^^
```

ZeroDivisionError: division by zero

```
>>> 4 + spam*3
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 4 + spam*3
 ^^^^^
```

NameError: name 'spam' is not defined

```
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 '2' + 2
 ^^^
```

TypeError: can only concatenate str (not "int") to str

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are [ZeroDivisionError](#), [NameError](#), and [TypeError](#).

The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
... pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an `except` clause listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
 pass

class C(B):
 pass

class D(C):
 pass
```



```

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
 except C:
 print("C")
 except B:
 print("B")

```

Note that if the *except* clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching *except* clause is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__() allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit`, which is raised by `sys.exit()` and `KeyboardInterrupt`, which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()

```

```

 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise

```

The `try ... except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```

for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()

```

The use of the *else* clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try* clause, but also those that occur inside functions that are called (even indirectly) in the *try* clause. For example:

```

>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero

```

---

## Table of Contents

- [8. Errors and Exceptions](#)
    - [8.1. Syntax Errors](#)
    - [8.2. Exceptions](#)
    - [8.3. Handling Exceptions](#)
    - [8.4. Raising Exceptions](#)
    - [8.5. Exception Chaining](#)
    - [8.6. User-defined Exceptions](#)
    - [8.7. Defining Clean-up Actions](#)
    - [8.8. Predefined Clean-up Actions](#)
    - [8.9. Raising and Handling Multiple Unrelated Exceptions](#)
    - [8.10. Enriching Exceptions with Notes](#)
- 

## Previous topic

[8. Errors and Exceptions — Python 3.13.5 documentation](#)

## Next topic

[9. Classes](#)

## This page

- [Report a bug](#)
- [Show source](#)

---

## Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |

[Python](#) »

---



[Copyright](#) 2001-2025, Python Software Foundation.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

Last updated on Jul 05, 2025 (07:10 UTC).

[Found a bug?](#)

Created using [Sphinx](#) 8.2.3.

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`"<stdin>"` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam * 3
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions¶

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
try:
 raise (RuntimeError, TypeError, NameError)
except (RuntimeError, TypeError, NameError):
 pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
 pass

class C(B):
 pass

class D(C):
 pass

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
 except C:
```

```

 print("C")
 except B:
 print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit`, which is raised by `sys.exit()` and `KeyboardInterrupt`, which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:
 print("Could not convert data to an integer.")
except Exception as err:

```

```
print(f"Unexpected {err=}, {type(err)=}")
raise
```

The `try ... except` statement has an optional *else clause*, which, when present, must follow all *except clauses*. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions¶

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise
```

## 8.5. Exception Chaining¶

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")
```

During handling of the above exception, the traceback shows the original cause, which can be linked via the `from` clause:

```
exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
... raise ConnectionError
...
>>> try:
... func()
... except ConnectionError as exc:
... raise RuntimeError('Failed to open database') from exc
...
```

```
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
 File "<stdin>", line 2, in func
ConnectionError
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
 File "<stdin>", line 4, in <module>
 File "<stdin>", line 2, in <module>
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
 open('database.sqlite')
except OSError:
 raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions¶

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.



Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions¶

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
 raise KeyboardInterrupt
finally:
 print('Goodbye, world!')
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue`, or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the `break`, `continue`, or `return`.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return`.

For example:

```
>>> def bool_return():
... try:
... return True
... finally:
... return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
... try:
... result = x / y
... except ZeroDivisionError:
... print("division by zero!")
... else:
... print("result is", result)
... finally:
... print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
```

```
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions¶

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
 print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly:

```
with open("myfile.txt") as f:
 for line in f:
 print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions¶

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
... excs = [OSError('error 1'), SystemError('error 2')]
... raise ExceptionGroup('there were problems', excs)
...
>>> f()
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| f()
| ~^^
```

```

| File "<stdin>", line 3, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----

>>> try:
... f()
... except Exception as e:
... print(f'caught {type(e)}: {e}')
...
caught <class 'ExceptionGroup'>:
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 2, in <module>
| f()
| ~^^
| File "<stdin>", line 3, in f
| raise ExceptionGroup('there were problems', excs)
| ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----

```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be reraised.

```

>>> def f():
... raise ExceptionGroup(
... "group1",
... [
... OSError(1),
... SystemError(2),
... ExceptionGroup(
... "group2",
... [
... OSError(3),
... RecursionError(4)
...]
...)
...]
...)
...
>>> try:
... f()
... except* OSError as e:
... print("There were OSErrors")
... except* SystemError as e:
... print("There were SystemErrors")
...
There were OSErrors

```

```

There were SystemErrors
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 2, in <module>
| f()
| ~^^
| File "<stdin>", line 2, in f
| raise ExceptionGroup(
| ~~~~~~
| File "<stdin>", line 9, in <module>
| raise ExceptionGroup('group1', [
| ~~~
| File "<stdin>", line 9, in f
| raise ExceptionGroup(
| ~~~~~~
| File "<stdin>", line 5, in f
| raise ExceptionGroup(
| ~~~~~~
| File "<stdin>", line 19, in <module>
| raise ExceptionGroup('group2', [
| ~~~
| File "<stdin>", line 11, in f
| raise OSError(3)
| OSError: [Errno 3] Operation not permitted
| Happend in object 3
| File "<stdin>", line 12, in f
| raise RecursionError(4)
| RecursionError: Maximum recursion depth exceeded

```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```

excs = []
for test in tests:
 try:
 test.run()
 except Exception as e:
 excs.append(e)

if excs:
 raise ExceptionGroup("Test Failures", excs)

```

## 8.10. Enriching Exceptions with Notes ¶

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

```

>>> try:
... raise TypeError('bad type')
... except Exception as e:
... e.add_note('Add some information')
... e.add_note('Add some more information')
... raise

```

```
...
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information
```

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred:

```
>>> def f():
... raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
... try:
... f()
... except Exception as e:
... e.add_note(f'Happened in Iteration {i+1}')
... excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| raise ExceptionGroup('We have some problems', excs)
| ~~~~~
| ExceptionGroup: We have some problems (3 sub-exceptions)
+----- 1 -----
| | |
| | | Traceback (most recent call last):
| | | | |
| | | | | File "<stdin>", line 3, in <module>
| | | | | |
| | | | | | f()
| | | | | | ~^^
| | | | | | File "<stdin>", line 2, in f
| | | | | | | |
| | | | | | | | raise OSError('operation failed')
| | | | | | | | OSError: operation failed
| | | | | | | | Happened in Iteration 1
+----- 2 -----
| | |
| | | Traceback (most recent call last):
| | | | |
| | | | | File "<stdin>", line 3, in <module>
| | | | | |
| | | | | | f()
| | | | | | ~^^
| | | | | | File "<stdin>", line 2, in f
| | | | | | | |
| | | | | | | | raise OSError('operation failed')
| | | | | | | | OSError: operation failed
| | | | | | | | Happened in Iteration 2
+----- 3 -----
| | |
| | | Traceback (most recent call last):
| | | | |
| | | | | File "<stdin>", line 3, in <module>
| | | | | |
| | | | | | f()
| | | | | | ~^^
| | | | | | File "<stdin>", line 2, in f
| | | | | | | |
| | | | | | | | raise OSError('operation failed')
| | | | | | | | OSError: operation failed
| | | | | | | | Happened in Iteration 3
```

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
 File "<stdin>", line 1
 while True print('Hello world')
 ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`:`) is missing just before it.

The file name (`<stdin>`) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages like:

```
>>> 10 * (1/0)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 10 * (1/0)
 ^^^
ZeroDivisionError: division by zero
```

```
>>> 4 + spam
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 4 + spam
 ^^^^
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 '2' + 2
 ^^^
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and

`TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
... try:
... x = int(input("Please enter a number: "))
... break
... except ValueError:
... print("Oops! That was no valid number. Try again...")
...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the *try clause*, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
try:
 raise (RuntimeError, TypeError, NameError)
except (RuntimeError, TypeError, NameError):
 pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
 pass

class C(B):
 pass
```

```

class D(C):
 pass

for cls in [B, C, D]:
 try:
 raise cls()
 except D:
 print("D")
 except C:
 print("C")
 except B:
 print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except* clause may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
... raise Exception('spam', 'eggs')
... except Exception as inst:
... print(type(inst)) # the exception type
... print(inst.args) # arguments stored in .args
... print(inst) # __str__ allows args to be printed directly,
... # but may be overridden in exception subclasses
... x, y = inst.args # unpack args
... print('x =', x)
... print('y =', y)

```

The exception's `__str__` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit`, which is raised by `sys.exit()`, and `KeyboardInterrupt`, which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
 f = open('myfile.txt')
 s = f.readline()
 i = int(s.strip())
except OSError as err:
 print("OS error:", err)
except ValueError:

```



```

 print("Could not convert data to an integer.")
except Exception as err:
 print(f"Unexpected {err=}, {type(err)=}")
 raise

```

The `try` ... `except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the *try* clause does not raise an exception.

```

for arg in sys.argv[1:]:
 try:
 f = open(arg, 'r')
 except OSError:
 print('cannot open', arg)
 else:
 print(arg, 'has', len(f.readlines()), 'lines')
 f.close()

```

The use of the *else* clause is better than adding additional code to the *try* clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` ... `except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try* clause, but also those that occur inside functions that are called (even indirectly) in the *try* clause. For example:

```

>>> def this_fails():
... x = 1/0
...
>>> try:
... this_fails()
... except ZeroDivisionError as err:
... print('Handling run-time error:', err)
...
Handling run-time error: division by zero

```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: HiThere

```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```

try:
 raise NameError('HiThere')
except NameError:
 print('An exception flew by!')
 raise

```

## 8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
 open("database.sqlite")
except OSError:
 raise RuntimeError("unable to handle error")
```

Traceback (most recent call last):

```
File "<stdin>", line 2, in <module>
 open("database.sqlite")
~~~~~
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise` statement allows an optional `from` clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func():
    raise ConnectionError

try:
    func()
except ConnectionError as exc:
    raise RuntimeError('Failed to open database') from exc
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
  File "<stdin>", line 2, in func
ConnectionError
```

It also allows disabling automatic exception chaining using the `from` `from None` idiom:

```
try:
    open('database.sqlite')
except OSError:
    raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2
    raise KeyboardInterrupt
KeyboardInterrupt
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes `break`, `continue`, or `return`, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the execution of that statement.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return`.

### Example:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

## More complex example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
  File "<stdin>", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

The `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. For example, opening and printing a file:

```
for line in open("myfile.txt"):
    print(line, end="")
```

However, this leaves the file open for an indeterminate amount of time. The `with` statement ensures the resource is always cleaned up properly:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

When the `with` block ends, the file `f` is always closed, even if errors occur during iteration.

## 8.9. Raising and Handling Multiple Unrelated Exceptions

Situations may require reporting multiple exceptions that occurred. The `ExceptionGroup` wraps a list of exception instances so they can be raised together, and can be caught like any other exception.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: {e}')
...
caught <class 'ExceptionGroup'>: ExceptionGroup: there were problems (2 sub-exceptions)
```

The exception group can be unpacked and handled selectively using `except*` syntax:

```
try:
    f()
except* OSError as e:
    print("There were OSErrors")
except* SystemError as e:
    print("There were SystemErrors")
```

Note: The nested exceptions in an exception group must be instances, not types.

## 8.10. Enriching Exceptions with Notes

After catching an exception, you can add notes for more context:

```
try:
    raise TypeError('bad type')
except Exception as e:
    e.add_note('Add some information')
    e.add_note('Add some more information')
    raise
```

When raised, the traceback will include all notes attached to the exception.

Similarly, when handling multiple exceptions, notes can be added to each:

```
def f():
    raise OSError('operation failed')

excs = []
for i in range(3):
    try:
        f()
    except Exception as e:
        e.add_note(f'Happened in Iteration {i+1}')
        excs.append(e)

raise ExceptionGroup('We have some problems', excs)
```

This concludes the overview of errors and exceptions in Python.

## 8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

### 8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
      File "<stdin>", line 1
        while True print('Hello world')
                                ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the `print()` function, since a colon ( `:` ) is missing just before it.

The file name ( `<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

### 8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError`, and `TypeError`.

The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general, it contains a traceback listing source lines; however, it will not display lines read from standard input.

[Built-in Exceptions](#) lists the built-in exceptions and their meanings.

## 8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control - C` or whatever the operating system supports); note that a user-generated interruption is signaled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number. Try again...")
... 
```

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its type matches the exception named after the `except` keyword, the *except clause* is executed, and then execution continues after the `try/except` block.
- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A `try` statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same `try` statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an `except` clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
```

```

        print("D")
    except C:
        print("C")
    except B:
        print("B")

```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception's *arguments*. The presence and types of the arguments depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, built-in exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```

>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst)) # the exception type
...     print(inst.args)  # arguments stored in .args
...     print(inst)       # __str__ allows args to be printed directly,
...                         # but may be overridden in exception subclasses
...     x, y = inst.args   # unpack args
...     print('x =', x)
...     print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```

import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")

```



```
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

The `try ... except` statement has an optional *else* clause, which, when present, must follow all *except* clauses. It is useful for code that must be executed if the `try` clause does not raise an exception.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try* clause, but also those that occur inside functions that are called (even indirectly) in the *try* clause. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

## 8.4. Raising Exceptions

The `raise` statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise
```

## 8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
try:
    open("database.sqlite")
except OSError:
    raise RuntimeError("unable to handle error")
```

Traceback output:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the `raise from` clause is used:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when transforming exceptions. For example:

```
def func():
    raise ConnectionError
...
try:
    func()
except ConnectionError as exc:
    raise RuntimeError('Failed to open database') from exc
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
try:
    open('database.sqlite')
except OSError:
    raise RuntimeError from None
```

For more information about chaining mechanics, see [Built-in Exceptions](#).

## 8.6. User-defined Exceptions

Programs may create their own exceptions by defining a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

## 8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2
    raise KeyboardInterrupt
KeyboardInterrupt
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, it may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `finally` clause executes a `break`, `continue`, or `return` statement, exceptions are not re-raised.
- If the `try` statement reaches a `break`, `continue`, or `return` statement, the `finally` clause will execute just prior to the execution of that statement.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

Example:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

More complex example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
... 
```

```
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "1")
  File "<stdin>", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

The `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and is re-raised after the `finally` clause has been executed. In real applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

## 8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen:

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 8.9. Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
... 
```

```
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...", line 3, in f
ExceptionGroup: there were problems (2 sub-exceptions)
+----- 1 -----
| OSError: error 1
+----- 2 -----
| SystemError: error 2
+-----
```

You can catch an `ExceptionGroup`:

```
try:
    f()
except Exception as e:
    print(f'caught {type(e)}: {e}')
```

which will print:

```
caught <class 'ExceptionGroup'>: ExceptionGroup: there were problems (2 sub-exceptions)
```

You can also handle only specific exceptions within an `ExceptionGroup` using `except*`:

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except *OSError as e:
...     print("There were OSErrors")
... except *SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
excs = []
for test in tests:
    try:
        test.run()
```

```

        except Exception as e:
            excs.append(e)
    ...
if excs:
    raise ExceptionGroup("Test Failures", excs)

```

## 8.10. Enriching Exceptions with Notes

When an exception is created to be raised, it is usually initialized with information that describes the error. There are cases where it is useful to add information after the exception was caught. Exceptions have a method `add_note(note)` that accepts a string and adds it to the exception's notes list. The standard traceback rendering includes all notes, in the order they were added, after the exception.

```

>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information

```

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following, each exception in the group has a note indicating when this error occurred.

```

>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
|       raise ExceptionGroup('We have some problems', excs)
|   ~^^
|   File "<stdin>", line 3, in <module>
|       raise ExceptionGroup('We have some problems', excs)
|   |   ...
|   |   File "<stdin>", line 2, in f
|   |   OSError: operation failed
|   |   Happened in Iteration 1
|   |   ...
|   |   File "<stdin>", line 2, in f
|   |   OSError: operation failed
|   |   Happened in Iteration 2

```

```

| | ...
| | File "<stdin>", line 2, in f
| | OSError: operation failed
| | Happened in Iteration 3
| +----- 1 -----
| | Traceback (most recent call last):
| |   File "<stdin>", line 3, in <module>
| |   ...
| | OSError: operation failed
| |   Happened in Iteration 1
| +----- 2 -----
| | Traceback (most recent call last):
| |   File "<stdin>", line 3, in <module>
| |   ...
| | OSError: operation failed
| |   Happened in Iteration 2
| +----- 3 -----
| | Traceback (most recent call last):
| |   File "<stdin>", line 3, in <module>
| |   ...
| | OSError: operation failed
| |   Happened in Iteration 3
| +-----

```

This concludes the section on [Errors and Exceptions](#).

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.



By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should

be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below *Private Variables*), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1](#)

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called [builtins](#).)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the [nonlocal](#) statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no [global](#) or [nonlocal](#) statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, [import](#) statements and function definitions bind the module or function name in the local scope.

The [global](#) statement can be used to indicate that particular variables live in the global scope and should be rebound there; the [nonlocal](#) statement indicates that particular variables live in an enclosing scope and should

be rebound there.

## 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A **namespace** is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.



By the way, I use the word **attribute** for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1](#)

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A **scope** is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should

be rebound there.

## 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

**Note:** how the *local* assignment (which is default) didn't change `scope_test`'s binding of `spam`. The `nonlocal` assignment changed `scope_test`'s binding of `spam`, and the `global` assignment changed the module-level binding.

You can also see that there was no previous binding for `spam` before the `global` assignment.

## 9.3. A First Look at Classes¶

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax¶

The simplest form of class definition looks like this:

```
class ClassName:
    <statement>-1
    .
    .
```



```
.  
<statement>-N
```

Class definitions, like function definitions ([def](#) statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an [if](#), or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the `end`), a **class object** is created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

## 9.3.2. Class Objects¶

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:  
    """A simple example class"""  
    i = 12345  
  
    def f(self):  
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `doc` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

*Instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):  
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example:

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i # (3.0, -4.5)
```

### 9.3.3. Instance Objects¶

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

will print the value `16`, without leaving a trace.

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object. Valid method names of an instance object depend on its class: all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a **method object**, not a function object.

### 9.3.4. Method Objects¶

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

This will continue to print `'hello world'` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

*Actually*, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of  $n$  arguments is equivalent to calling the

corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

### 9.3.5. Class and Instance Variables¶

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
d.kind          # 'canine'
e.kind          # 'canine'
d.name          # 'Fido'
e.name          # 'Buddy'
```

As discussed in [A Word About Names and Objects](#), shared data can have possibly surprising effects with involving [mutable](#) objects such as lists and dictionaries. For example, the `tricks` list in the following code should not be used as a class variable because just a single list would be shared by all `Dog` instances:

```
class Dog:
    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over', 'play dead'], unexpectedly shared by all dogs
```

Correction involves making `tricks` an instance variable:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
```

```
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over']
e.tricks # ['play dead']
```

*(End of snippet)*

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense, the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case, there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module’s global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and

should be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.



By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1](#)

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should

be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how [global](#) and [nonlocal](#) affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn't change *scope\_test's* binding of *spam*. The [nonlocal](#) assignment changed *scope\_test's* binding of *spam*, and the [global](#) assignment changed the module-level binding.

## 9.3. A First Look at Classes¶

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax¶

The simplest form of class definition looks like this:

```
class ClassName:
    <statement>-1
    .
    .
    .
    <statement>-N
```

Class definitions, like function definitions ([def](#) statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an [if](#) statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

### 9.3.2. Class Objects¶

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `doc` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example:

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i
# (3.0, -4.5)
```

### 9.3.3. Instance Objects¶

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

will print `16` without leaving a trace (except for the deletion).

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object. Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

### 9.3.4. Method Objects¶

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `'hello world'` until interrupted.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

*Actually*, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of  $n$  arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

### 9.3.5. Class and Instance Variables¶

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
d.kind # 'canine'
e.kind # 'canine'
d.name # 'Fido'
e.name # 'Buddy'
```

As discussed in [A Word About Names and Objects](#), shared data can have possibly surprising effects with involving [mutable](#) objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:
    tricks = [] # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over', 'play dead']
e.tricks # ['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = [] # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
```

```
e.add_trick('play dead')
d.tricks # ['roll over']
e.tricks # ['play dead']
```

---

© [Python Software Foundation](#), 2001-2025.

*Last updated on Jul 05, 2025 (07:10 UTC).*

*This page is licensed under the [Python Software Foundation License Version 2](#).*

*Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.*

[Please donate](#).

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should



be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

(Note how the local assignment (which is default) didn't change `scope_test`'s binding of `spam`. The nonlocal assignment changed `scope_test`'s binding of `spam`, and the global assignment changed the module-level binding.)

## 9.3. A First Look at Classes¶

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax¶

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    ...
    <statement-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

### 9.3.2. Class Objects¶

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class’s namespace when the class object was created. For example, if the class looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
class Complex:
    def __init__(self, realpart, imagpart):
```

```

        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i # (3.0, -4.5)

```

### 9.3.3. Instance Objects¶

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code:

```

x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter

```

will print `16`. (Note: `del` removes the attribute `counter` from `x`.)

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

### 9.3.4. Method Objects¶

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```

xf = x.f
while True:
    print(xf())

```

This will continue to print `'hello world'` until interrupted.

What exactly happens when a method is called? You may have noticed that `x.f()` was invoked without an explicit argument, even though the function `f()` requires one. The answer is that the *instance object* is passed as the first argument of the function. In our example, the call `x.f()` is equivalent to `MyClass.f(x)`.

In general, calling a method with `n` arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance’s class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

## 9.3.5. Class and Instance Variables¶

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
print(d.kind) # 'canine'
print(e.kind) # 'canine'
print(d.name) # 'Fido'
print(e.name) # 'Buddy'
```

Shared data can have possibly surprising effects involving [mutable](#) objects such as lists and dictionaries. For example, the `tricks` list in the following code:

```
class Dog:
    tricks = [] # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)
```

should not be used as a class variable, because only a single list would be shared by all `Dog` instances. Instead, use an instance variable:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = [] # creates a new empty list for each dog
    def add_trick(self, trick):
        self.tricks.append(trick)
```

This design ensures each dog has its own tricks list.

---

*(The document continues with additional sections about private variables, odds and ends, iterators, generators, and more.)*

# 9. Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

## 9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

## 9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense, the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case, there is a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in

particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

### 9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9.3. A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement>-1
    .
    .
    .
    <statement>-N
```

---

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the `end`), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

### 9.3.2. Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example:



```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i # (3.0, -4.5)
```

### 9.3.3. Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

### 9.3.4. Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `'hello world'` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of `n` arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

## 9.3.5. Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
d.kind # 'canine'
e.kind # 'canine'
d.name # 'Fido'
e.name # 'Buddy'
```

As discussed in [A Word About Names and Objects](#), shared data can have possibly surprising effects involving [mutable](#) objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all `Dog` instances:

```
class Dog:
    tricks = [] # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over', 'play dead']
e.tricks # ['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = [] # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
```

```
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over']
e.tricks # ['play dead']
```

## 9. Classes ¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects ¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces ¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called [builtins](#).)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the [nonlocal](#) statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no [global](#) or [nonlocal](#) statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, [import](#) statements and function definitions bind the module or function name in the local scope.

The [global](#) statement can be used to indicate that particular variables live in the global scope and should be rebound there; the [nonlocal](#) statement indicates that particular variables live in an enclosing scope and should

be rebound there.

### 9.2.1. Scopes and Namespaces Example ¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

# 9. Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

## 9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

## 9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should



be rebound there.

## 9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A **namespace** is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word **attribute** for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace![1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an unhandled exception. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A **scope** is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the *local* names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and

should be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

---

## 9.3. A First Look at Classes¶

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax¶

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a **class object** is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

### 9.3.2. Class Objects¶

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example:

```
class Complex:
    def __init__(self, realpart, imagpart):
```

```

        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i # (3.0, -4.5)

```

---

### 9.3.3. Instance Objects ¶

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code:

```

x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter

```

will print `16`, without leaving a trace.

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function. However, `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

---

### 9.3.4. Method Objects ¶

Usually, a method is called right after it is bound:

```
x.f()
```

In the *MyClass* example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object and can be stored away and called at a later time. For example:

```

xf = x.f
while True:
    print(xf())

```

will continue to print `'hello world'` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

---

### 9.3.5. Class and Instance Variables¶

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
d.kind # 'canine'
e.kind # 'canine'
d.name # 'Fido'
e.name # 'Buddy'
```

As discussed in [A Word About Names and Objects](#), shared data can have possibly surprising effects with involving [mutable](#) objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:
    tricks = [] # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over', 'play dead']
e.tricks # ['roll over', 'play dead']
```

Correct design of the class should use an *instance* variable instead:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = [] # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
```

```
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over']
e.tricks # ['play dead']
```

---

## 9.4. Random Remarks ¶

If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
class Warehouse:
    purpose = 'storage'
    region = 'west'

w1 = Warehouse()
print(w1.purpose, w1.region) # storage west

w2 = Warehouse()
w2.region = 'east'
print(w2.purpose, w2.region) # storage east
```

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also OK. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x + y)

class C:
    f = f1

    def g(self):
        return 'hello world'

h = g
```



Now `f`, `g`, and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

---

## 9.5. Inheritance ¶

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a namespace accessible from the scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the

same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively *virtual*.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

### 9.5.1. Multiple Inheritance¶

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, calls each parent only once, and is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see [The Python 2.3 Method Resolution Order](#).

---

## 9.6. Private Variables¶

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

See also the [private name mangling specifications](#) for details and special cases.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

The above example would work even if `MappingSubclass` were to introduce an `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()`, and `delattr()`, as well as when referencing `__dict__`.

---

## 9.7. Odds and Ends ¶

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct,” bundling together a few named data items. The idiomatic approach is to use [dataclasses](#) for this purpose:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
john = Employee('john', 'computer lab', 1000)
john.dept # 'computer lab'
john.salary # 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

## 9.8. Iterators¶

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)

for element in (1, 2, 3):
    print(element)

for key in {'one': 1, 'two': 2}:
    print(key)

for char in "123":
    print(char)

for line in open("myfile.txt"):
    print(line, end='')

```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; the following example shows how it all works:

```
s = 'abc'
it = iter(s)
it # <str_iterator object at 0x10c90e650>
next(it) # 'a'
next(it) # 'b'
next(it) # 'c'
next(it) # Traceback (most recent call last): StopIteration

```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

```

```

def __iter__(self):
    return self

def __next__(self):
    if self.index == 0:
        raise StopIteration
    self.index -= 1
    return self.data[self.index]

rev = Reverse('spam')
iter(rev)
for char in rev:
    print(char)

```

---

## 9.9. Generators¶

**Generators** are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```

def reverse(data):
    for index in range(len(data) - 1, -1, -1):
        yield data[index]

for char in reverse('golf'):
    print(char)

```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

---

## 9.10. Generator Expressions¶

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```

sum(i * i for i in range(10)) # sum of squares

xvec = [10, 20, 30]
yvec = [7, 5, 3]
sum(x * y for x, y in zip(xvec, yvec)) # dot product

```

```
unique_words = set(word for line in page for word in line.split())

valedictorian = max((student.gpa, student.name) for student in graduates)

data = 'golf'
list(data[i] for i in range(len(data) - 1, -1, -1))
# ['f', 'l', 'o', 'g']
```

Footnotes:

[1] Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

# Hello

## Hello

### Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

[Hello](#)

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

---

## 9. Classes — Python 3.13.5 documentation

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

## 9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

## 9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense, the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case, there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.



A *scope* is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

### 9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"
```

```

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)

```

The output of the example code is:

```

After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam

```

## 9.3. A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax

The simplest form of class definition looks like this:

```

class ClassName:
    <statement-1>
    ...
    <statement-N>

```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

### 9.3.2. Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore, a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example:

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

# Usage:
x = Complex(3.0, -4.5)
x.r, x.i # (3.0, -4.5)
```

### 9.3.3. Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `f` is a function in the class `MyClass`. `x.f()` will return `'hello world'`. Note that `x.f` is a *method object*, not a function object.

---

*(The document continues with detailed sections on method objects, class and instance variables, inheritance, private variables, and other advanced topics.)*

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

You can also see that there was no previous binding for `spam` before the `global spam` assignment.

You can also see that there was no previous binding for `spam` before the `global spam` assignment.

Note how the *local* assignment (which is default) didn't change `scope_test's` binding of `spam`. The `nonlocal` assignment changed `scope_test's` binding of `spam`, and the `global` assignment changed the module-level

binding.

You can also see that there was no previous binding for `spam` before the `global spam` assignment.

Note how the *local* assignment (which is default) didn't change *scope\_test's* binding of *spam*. The `nonlocal` assignment changed *scope\_test's* binding of *spam*, and the `global` assignment changed the module-level binding.

## 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

# 9. Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

## 9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

## 9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.



By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should

be rebound there.

### 9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `[**maximize**](../library/functions.html#maximize)` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `[**z.real**](../library/builtins.html#z.real)`, `[**real**]` (`../library/builtins.html#z.real`) is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `[**modname.funcname**]` (`../library/builtins.html#modname.funcname`), `[**modname**]` (`../library/builtins.html#modname`) is a module object and `[**funcname**]` (`../library/builtins.html#funcname`) is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then attribute lookups and assignments go directly to the next-to-last scope containing the module’s global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced

by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

*(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)*

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense, the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.



By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case, there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should

be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```



# Hello

## Hello

### Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

[Hello](#)

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

---

## 9. Classes — Python 3.13.5 documentation

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

## 9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

## 9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module’s global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` or `nonlocal` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

## 9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
```

```
do_nonlocal()
print("After nonlocal assignment:", spam)
do_global()
print("After global assignment:", spam)
```

```
scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense, the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 9.3. A First Look at Classes¶

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

### 9.3.1. Class Definition Syntax¶

The simplest form of class definition looks like this:

```
class ClassName:
    <statement>-1>
    ...
    <statement>-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)



In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName`).

### 9.3.2. Class Objects¶

Class objects support two kinds of operations: attribute references and instantiation.

*Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
```



```

        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i
# (3.0, -4.5)

```

### 9.3.3. Instance Objects¶

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

*Data attributes* correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following code will print the value `16`, without leaving a trace:

```

x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter

```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

### 9.3.4. Method Objects¶

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```

xf = x.f
while True:
    print(xf())

```

will continue to print `'hello world'` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of `n` arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called

with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

## 9.3.5. Class and Instance Variables¶

Generally speaking, instance variables are for data unique to each instance, and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
d.kind # 'canine'
e.kind # 'canine'
d.name # 'Fido'
e.name # 'Buddy'
```

As discussed in [A Word About Names and Objects](#), shared data can have possibly surprising effects with involving *mutable* objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:
    tricks = [] # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over', 'play dead']
e.tricks # ['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = [] # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks # ['roll over']
e.tricks # ['play dead']
```

## 9.4. Random Remarks¶

If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
class Warehouse:
    purpose = 'storage'
    region = 'west'

w1 = Warehouse()
print(w1.purpose, w1.region)  # storage west
w2 = Warehouse()
w2.region = 'east'
print(w2.purpose, w2.region)  # storage east
```

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x + y)

class C:
    f = f1

    def g(self):
        return 'hello world'
```

```
h = g
```

Now `f`, `g`, and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

## 9.5. Inheritance ¶

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement>-1>
    ...
    <statement>-N>
```

The name `BaseClassName` must be defined in a namespace accessible from the scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively *virtual*.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

### 9.5.1. Multiple Inheritance¶

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement>-1>
    ...
    <statement>-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the `super` call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see [The Python 2.3 Method Resolution Order](#).

## 9.6. Private Variables¶

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

See also:

- The [private name mangling specifications](#) for details and special cases.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```

class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update() but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)

```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

## 9.7. Odds and Ends ¶

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. The idiomatic approach is to use [dataclasses](#) for this purpose:

```

from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int

john = Employee('john', 'computer lab', 1000)
john.dept # 'computer lab'
john.salary # 1000

```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

[Instance method objects](#) have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the [function object](#) corresponding to the method.

## 9.8. Iterators ¶

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one': 1, 'two': 2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
s = 'abc'
it = iter(s)
it
# <str_iterator object at 0x10c90e650>
next(it)
# 'a'
next(it)
# 'b'
next(it)
# 'c'
next(it)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index -= 1
        return self.data[self.index]

rev = Reverse('spam')
iter(rev)
# <__main__.Reverse object at 0x00A1DB50>
```

```

for char in rev:
    print(char)
# m
# a
# p
# s

```

## 9.9. Generators¶

**Generators** are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```

def reverse(data):
    for index in range(len(data) - 1, -1, -1):
        yield data[index]

for char in reverse('golf'):
    print(char)
# g
# l
# o
# f

```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

## 9.10. Generator Expressions¶

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```

sum(i * i for i in range(10)) # sum of squares
xvec = [10, 20, 30]
yvec = [7, 5, 3]
sum(x * y for x, y in zip(xvec, yvec)) # dot product
unique_words = set(word for line in page for word in line.split())

valedictorian = max((student.gpa, student.name) for student in graduates)
data = 'golf'
list(data[i] for i in range(len(data)-1, -1, -1)) # ['f', 'l', 'o', 'g']

```



## Footnotes

[1]: Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

## 9. Classes¶

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

### 9.1. A Word About Names and Objects¶

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### 9.2. Python Scopes and Namespaces¶

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as [abs\(\)](#), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace![1]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `main`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should

be rebound there.

### 9.2.1. Scopes and Namespaces Example¶

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs') # Change current working directory
>>> os.system('mkdir today')      # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
sys.stderr.write('Warning, log file not found starting a new one\n')
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6) # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode() # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip()) # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
...     """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the



docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules’ names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#), which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs') # Change current working directory
>>> os.system('mkdir today')      # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The [argparse](#) module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The [sys](#) module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

```
>>> import sys
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use [sys.exit\(\)](#).

## 10.5. String Pattern Matching

The [re](#) module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The [math](#) module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6) # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode() # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip()) # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware:

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

```
>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite their names, no direct knowledge or handling of XML is needed.
- The [email](#) package is for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of CSV files, widely used by databases and spreadsheets. XML processing is supported by [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#). These modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database accessible with slightly nonstandard SQL syntax.
- Internationalization is supported by modules such as [gettext](#), [locale](#), and the [codecs](#) package.

*This page is licensed under the Python Software Foundation License Version 2.*

*Examples, recipes, and other code are additionally licensed under the Zero Clause BSD License.*

See [History and License](#) for more information.

*The Python Software Foundation is a non-profit corporation.*

[Please donate.](#)

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
# <returns a list of all module functions>
>>> help(os)
# <returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:



```
['demo.py', 'one', 'two', 'three']
```

The [argparse](#) module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to 5 and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination ¶

The [sys](#) module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

```
>>> import sys
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use [sys.exit\(\)](#).

## 10.5. String Pattern Matching ¶

The [re](#) module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics ¶

The [math](#) module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access ¶

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())    # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
...     """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times ¶

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

```
>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression ¶

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement ¶

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time-critical sections in larger blocks of code.

## 10.11. Quality Control ¶

One approach for developing high-quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as copying and pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included ¶

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface¶

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards¶

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments¶

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to 5 and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination¶

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching¶

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics¶

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access ¶

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())    # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail(
...     'soothsayer@example.org',
...     'jcaesar@example.org',
...     """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March."""
... )
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times¶

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

  

```
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression¶

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement¶

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control¶



One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included ¶

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.

- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> import sys
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())    # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#):

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:



```
['demo.py', 'one', 'two', 'three']
```

The [argparse](#) module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The [sys](#) module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

```
>>> import sys
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The [re](#) module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The [math](#) module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())    # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

```
# dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#) and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#), but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by modules including [gettext](#), [locale](#), and the [codecs](#) package.

*Last updated on Jul 05, 2025 (07:10 UTC).*

[Please donate.](#)

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The [argparse](#) module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to 5 and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The [sys](#) module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

```
sys.stderr.write('Warning, log file not found starting a new one\n')
```

The most direct way to terminate a script is to use [sys.exit\(\)](#).

## 10.5. String Pattern Matching

The [re](#) module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The [math](#) module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode() # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip()) # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mail server running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware:



```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time-critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high-quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules’ names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\\Python313'
>>> os.chdir('/server/accesslogs') # Change current working directory
>>> os.system('mkdir today') # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
# returns a list of all module functions
>>> help(os)
# returns an extensive manual page created from the module's docstrings
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For example, consider the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Running `python demo.py one two three` on the command line produces:

```
['demo.py', 'one', 'two', 'three']
```

The [argparse](#) module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, it sets `args.lines` to 5 and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The [sys](#) module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

```
>>> import sys
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use [sys.exit\(\)](#).

## 10.5. String Pattern Matching

The [re](#) module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The [math](#) module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6) # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

## 10.7. Internet Access

There are several modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode() # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip()) # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example requires a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member

extraction for output formatting and manipulation. The module also supports objects that are timezone-aware.

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are supported by modules such as [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#):

```
import zlib
s = b'witch which has which witches wrist watch'
len(s)
41
t = zlib.compress(s)
len(t)
37
zlib.decompress(t)
b'witch which has which witches wrist watch'
zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time-critical sections in larger blocks of code.

## 10.11. Quality Control

Writing tests for each function during development and running them frequently improves software quality. The [doctest](#) module provides a tool for scanning a module and validating embedded tests in docstrings:

```

def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests

The unittest module allows larger, more comprehensive tests to be maintained separately:

import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Invokes all tests from the command line

```

## 10.12. Batteries Included

Python has a “batteries included” philosophy, with extensive and robust capabilities in its larger packages. Examples include:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules for remote procedure calls that do not require direct XML handling.
- The [email](#) package for managing email messages, MIME, and RFC 2822-compliant message documents.
- The [json](#) package for parsing JSON data. The [csv](#) module supports reading and writing CSV files.
- XML processing via [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#).
- The [sqlite3](#) module for an embedded SQL database.
- Internationalization modules like [gettext](#), [locale](#), and [codecs](#).

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:



```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10) # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6) # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode() # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip()) # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail(
...     'soothsayer@example.org',
...     'jcaesar@example.org',
...     """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time-critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#), which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.

- Internationalization is supported by modules including [gettext](#), [locale](#), and the [codecs](#) package.

---

© [Python Software Foundation](#), 2001-2025.

This page is licensed under the Python Software Foundation License Version 2.

Examples, recipes, and other code in the documentation are additionally licensed under the Zero Clause BSD License.

See [History and License](#) for more information.

The Python Software Foundation is a non-profit corporation.

[Please donate](#).

*Last updated on Jul 05, 2025 (07:10 UTC).*

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
# returns a list of all module functions
>>> help(os)
# returns an extensive manual page created from the module's docstrings
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
sys.stderr.write('Warning, log file not found starting a new one\n')
# Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
from urllib.request import urlopen

with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
    for line in response:
        line = line.decode()    # Convert bytes to a str
        if line.startswith('datetime'):
            print(line.rstrip())    # Remove trailing newline
# datetime: 2022-01-01T01:36:47.689215+00:00

import smtplib
server = smtplib.SMTP('localhost')
server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
    """To: jcaesar@example.org
From: soothsayer@example.org

Beware the Ides of March.
""")
server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times



The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

```
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
import zlib
s = b'witch which has which witches wrist watch'
len(s)
# 41
t = zlib.compress(s)
len(t)
# 37
zlib.decompress(t)
# b'witch which has which witches wrist watch'
zlib.crc32(s)
# 226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> import sys
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())    # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time-critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high-quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#), but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#), which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Values format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

*For more information, see the full documentation.*

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:



```
['demo.py', 'one', 'two', 'three']
```

The [argparse](#) module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The [sys](#) module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
sys.stderr.write('Warning, log file not found starting a new one\n')
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The [re](#) module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The [math](#) module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())    # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#) and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#) module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules’ names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

# 10. Brief Tour of the Standard Library

## 10.1. Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\Python313'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
# returns a list of all module functions
>>> help(os)
# returns an extensive manual page created from the module's docstrings
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

## 10.2. File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

## 10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file'
)
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

## 10.4. Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

## 10.5. String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

## 10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
```

```
>>> math.log(1024, 2)
10.0
```

The [random](#) module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)
4
```

The [statistics](#) module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

## 10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are [urllib.request](#) for retrieving data from URLs and [smtplib](#) for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()    # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip()) # Remove trailing newline
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

## 10.8. Dates and Times

The [datetime](#) module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member

extraction for output formatting and manipulation. The module also supports objects that are timezone-aware.

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'
```

```
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

## 10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: [zlib](#), [gzip](#), [bz2](#), [lzma](#), [zipfile](#), and [tarfile](#).

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

## 10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The [timeit](#) module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to [timeit](#)'s fine level of granularity, the [profile](#) and [pstats](#) modules provide tools for identifying time-critical sections in larger blocks of code.

## 10.11. Quality Control

One approach for developing high-quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.



The [doctest](#) module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod() # automatically validate the embedded tests
```

The [unittest](#) module is not as effortless as the [doctest](#), but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

## 10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The [xmlrpc.client](#) and [xmlrpc.server](#) modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The [email](#) package is a library for managing email messages, including MIME and other [RFC 2822](#)-based message documents. Unlike [smtplib](#) and [poplib](#) which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The [json](#) package provides robust support for parsing this popular data interchange format. The [csv](#) module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the [xml.etree.ElementTree](#), [xml.dom](#), and [xml.sax](#) packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The [sqlite3](#) module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including [gettext](#), [locale](#), and the [codecs](#) package.

