

Real Python

Strings and Character Data in Python

by John Sturtz 8 Comments basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [String Manipulation](#)
 - [String Operators](#)
 - [Built-in String Functions](#)
 - [String Indexing](#)
 - [String Slicing](#)
 - [Specifying a Stride in a String Slice](#)
 - [Interpolating Variables Into a String](#)
 - [Modifying Strings](#)
 - [Built-in String Methods](#)
- [bytes Objects](#)
 - [Defining a Literal bytes Object](#)
 - [Defining a bytes Object With the Built-in bytes\(\) Function](#)
 - [Operations on bytes Objects](#)
 - [bytearray Objects](#)
- [Conclusion](#)

A promotional graphic for Redis Labs. It features the Redis logo (a red cube with a white star) and the Python logo (a yellow diamond with a blue 'P'). The text "Enhance Python with Redis" is centered between them. To the right is a blue button with the text "Explore Redis Labs". Below the main text is the Redis Labs logo with the tagline "HOME OF REDIS".

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Strings and Character Data in Python](#)

In the tutorial on [Basic Data Types in Python](#), you learned how to define **strings**: objects that contain sequences of character data. Processing character data is integral to programming. It is a rare application that doesn't need to manipulate strings at least to some extent.

Here's what you'll learn in this tutorial: Python provides a rich set of operators, functions, and methods for working with strings. When you are finished with this tutorial, you will know how to access and extract portions of

strings, and also be familiar with the methods that are available to manipulate and modify string data.

You will also be introduced to two other Python objects used to represent raw byte data, the bytes and bytearray types.

 **Take the Quiz:** Test your knowledge with our interactive “Python Strings and Character Data” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

String Manipulation

The sections below highlight the operators, methods, and functions that are available for working with strings.

String Operators

You have already seen the operators + and * applied to numeric operands in the tutorial on [Operators and Expressions in Python](#). These two operators can be applied to strings as well.

The + Operator

The + operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

```
Python >>>
>>> s = 'foo'
>>> t = 'bar'
>>> u = 'baz'

>>> s + t
'foobar'
>>> s + t + u
'foobarbaz'

>>> print('Go team' + '!!!!')
Go team!!!
```

The * Operator

The * operator creates multiple copies of a string. If s is a string and n is an integer, either of the following expressions returns a string consisting of n concatenated copies of s:

```
s * n
n * s
```

Here are examples of both forms:

```
Python >>>
>>> s = 'foo.'
>>> s * 4
'foo.foo.foo.foo.'
>>> 4 * s
'foo.foo.foo.foo.'
```

The multiplier operand n must be an integer. You'd think it would be required to be a positive integer, but amazingly, it can be zero or negative, in which case the result is an empty string:

```
Python >>>
>>> 'foo' * -8
''
```

If you were to create a string variable and initialize it to the empty string by assigning it the value 'foo' * -8, anyone would rightly think you were a bit daft. But it would work.

The `in` Operator

Python also provides a membership operator that can be used with strings. The `in` operator returns `True` if the first operand is contained within the second, and `False` otherwise:

```
Python >>>
>>> s = 'foo'
>>> s in 'That\'s food for thought.'
True
>>> s in 'That\'s good for now.'
False
```

There is also a `not in` operator, which does the opposite:

```
Python >>>
>>> 'z' not in 'abc'
True
>>> 'z' not in 'xyz'
False
```

Built-in String Functions

As you saw in the tutorial on [Basic Data Types in Python](#), Python provides many functions that are built-in to the interpreter and always available. Here are a few that work with strings:

Function	Description
<code>chr()</code>	Converts an integer to a character
<code>ord()</code>	Converts a character to an integer
<code>len()</code>	Returns the length of a string
<code>str()</code>	Returns a string representation of an object

These are explored more fully below.

`ord(c)`

Returns an integer value for the given character.

At the most basic level, computers store all information as numbers. To represent character data, a translation scheme is used which maps each character to its representative number.

The simplest scheme in common use is called [ASCII](#). It covers the common Latin characters you are probably most accustomed to working with. For these characters, `ord(c)` returns the ASCII value for character `c`:

```
Python >>>
>>> ord('a')
97
>>> ord('#')
35
```

ASCII is fine as far as it goes. But there are many different languages in use in the world and countless symbols and glyphs that appear in digital media. The full set of characters that potentially may need to be represented in computer code far surpasses the ordinary Latin letters, numbers, and symbols you usually see.

[Unicode](#) is an ambitious standard that attempts to provide a numeric code for every possible character, in every possible language, on every possible platform. Python 3 supports Unicode extensively, including allowing Unicode characters within strings.

For More Information: See [Unicode & Character Encodings in Python: A Painless Guide](#) and [Python's Unicode Support](#) in the Python documentation.

As long as you stay in the domain of the common characters, there is little practical difference between ASCII and Unicode. But the `ord()` function will return numeric values for [Unicode characters](#) as well:

Python

```
>>> ord('€')
8364
>>> ord('Σ')
8721
```

>>>

`chr(n)`

Returns a character value for the given integer.

`chr()` does the reverse of `ord()`. Given a numeric value `n`, `chr(n)` returns a string representing the character that corresponds to `n`:

Python

```
>>> chr(97)
'a'
>>> chr(35)
'#'
```

>>>

`chr()` handles Unicode characters as well:

Python

```
>>> chr(8364)
'€'
>>> chr(8721)
'Σ'
```

>>>

`len(s)`

Returns the length of a string.

With `len()`, you can check Python string length. `len(s)` returns the number of characters in `s`:

Python

```
>>> s = 'I am a string.'
>>> len(s)
14
```

>>>

`str(obj)`

Returns a string representation of an object.

Virtually any object in Python can be rendered as a string. `str(obj)` returns the string representation of object `obj`:

```
Python >>> str(49.2)
'49.2'
>>> str(3+4j)
'(3+4j)'
>>> str(3 + 29)
'32'
>>> str('foo')
'foo'
```

String Indexing

Often in programming languages, individual items in an ordered set of data can be accessed directly using a numeric index or key value. This process is referred to as indexing.

In Python, strings are ordered sequences of character data, and thus can be indexed in this way. Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets ([]).

String indexing in Python is zero-based: the first character in the string has index 0, the next has index 1, and so on. The index of the last character will be the length of the string minus one.

For example, a schematic diagram of the indices of the string 'foobar' would look like this:

String Indices

The individual characters can be accessed by index as follows:

```
Python >>> s = 'foobar'
>>> s[0]
'f'
>>> s[1]
'o'
>>> s[3]
'b'
>>> len(s)
6
>>> s[len(s)-1]
'r'
```

Attempting to index beyond the end of the string results in an error:

```
Python >>> s[6]
```

```
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    s[6]
IndexError: string index out of range
```

String indices can also be specified with negative numbers, in which case indexing occurs from the end of the string backward: `-1` refers to the last character, `-2` the second-to-last character, and so on. Here is the same diagram showing both the positive and negative indices into the string `'foobar'`:

Positive and Negative String Indices

Here are some examples of negative indexing:

```
Python >>>
>>> s = 'foobar'
>>> s[-1]
'r'
>>> s[-2]
'a'
>>> len(s)
6
>>> s[-len(s)]
'f'
```

Attempting to index with negative numbers beyond the start of the string results in an error:

```
Python >>>
>>> s[-7]
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    s[-7]
IndexError: string index out of range
```

For any non-empty string `s`, `s[len(s)-1]` and `s[-1]` both return the last character. There isn't any index that makes sense for an empty string.

String Slicing

Python also allows a form of indexing syntax that extracts substrings from a string, known as string slicing. If `s` is a string, an expression of the form `s[m:n]` returns the portion of `s` starting with position `m`, and up to but not including position `n`:

```
Python >>>
>>> s = 'foobar'
>>> s[2:5]
'oba'
```

Remember: String indices are zero-based. The first character in a string has index `0`. This applies to both standard indexing and slicing.

Again, the second index specifies the first character that is not included in the result—the character `'r'` (`s[5]`) in the example above. That may seem slightly unintuitive, but it produces this result which makes sense: the expression `s[m:n]` will return a substring that is $n - m$ characters in length, in this case, $5 - 2 = 3$.

If you omit the first index, the slice starts at the beginning of the string. Thus, `s[:m]` and `s[0:m]` are equivalent:

Python

>>>

```
>>> s = 'foobar'  
>>> s[:4]  
'foob'  
>>> s[0:4]  
'foob'
```

Similarly, if you omit the second index as in `s[n:]`, the slice extends from the first index through the end of the string. This is a nice, concise alternative to the more cumbersome `s[n:len(s)]`:

Python

>>>

```
>>> s = 'foobar'  
>>> s[2:]  
'obar'  
>>> s[2:len(s)]  
'obar'
```

For any string `s` and any integer `n` ($0 \leq n \leq \text{len}(s)$), `s[:n] + s[n:]` will be equal to `s`:

Python

>>>

```
>>> s = 'foobar'  
>>> s[:4] + s[4:]  
'foobar'  
>>> s[:4] + s[4:] == s  
True
```

Omitting both indices returns the original string, in its entirety. Literally. It's not a copy, it's a reference to the original string:

Python

>>>

```
>>> s = 'foobar'  
>>> t = s[:]  
>>> id(s)  
59598496  
>>> id(t)  
59598496  
>>> s is t  
True
```

If the first index in a slice is greater than or equal to the second index, Python returns an empty string. This is yet another obfuscated way to generate an empty string, in case you were looking for one:

Python

>>>

```
>>> s[2:2]  
''  
>>> s[4:2]  
''
```

Negative indices can be used with slicing as well. `-1` refers to the last character, `-2` the second-to-last, and so on, just as with simple indexing. The diagram below shows how to slice the substring `'oob'` from the string `'foobar'` using both positive and negative indices:

String Slicing with Positive and Negative Indices

Here is the corresponding Python code:

```
Python >>> s = 'foobar'

>>> s[-5:-2]
'oob'
>>> s[1:4]
'oob'
>>> s[-5:-2] == s[1:4]
True >>>
```

Specifying a Stride in a String Slice

There is one more variant of the slicing syntax to discuss. Adding an additional : and a third index designates a stride (also called a step), which indicates how many characters to jump after retrieving each character in the slice.

For example, for the string 'foobar', the slice 0:6:2 starts with the first character and ends with the last character (the whole string), and every second character is skipped. This is shown in the following diagram:

String Indexing with Stride

Similarly, 1:6:2 specifies a slice starting with the second character (index 1) and ending with the last character, and again the stride value 2 causes every other character to be skipped:

Another String Indexing with Stride

The illustrative REPL code is shown here:

```
Python >>> s = 'foobar'

>>> s[0:6:2]
'foa'
>>> s[1:6:2]
'obr' >>>
```

As with any slicing, the first and second indices can be omitted, and default to the first and last characters respectively:

```
Python >>> s = '12345' * 5

>>> s
'1234512345123451234512345' >>>
```

```
'1234512345123451234512345'  
>>> s[::5]  
'11111'  
>>> s[4::5]  
'55555'
```

You can specify a negative stride value as well, in which case Python steps backward through the string. In that case, the starting/first index should be greater than the ending/second index:

```
Python  
>>> s = 'foobar'  
>>> s[5:0:-2]  
'rbo'
```

In the above example, `5:0:-2` means “start at the last character and step backward by 2, up to but not including the first character.”

When you are stepping backward, if the first and second indices are omitted, the defaults are reversed in an intuitive way: the first index defaults to the end of the string, and the second index defaults to the beginning. Here is an example:

```
Python  
>>> s = '12345' * 5  
>>> s  
'1234512345123451234512345'  
>>> s[::-5]  
'55555'
```

This is a common paradigm for reversing a string:

```
Python  
>>> s = 'If Comrade Napoleon says it, it must be right.'  
>>> s[::-1]  
'.thgir eb tsum ti ,ti syas noelopaN edarmoC fi'
```

Interpolating Variables Into a String

In Python version 3.6, a new string formatting mechanism was introduced. This feature is formally named the Formatted String Literal, but is more usually referred to by its nickname **f-string**.

The formatting capability provided by f-strings is extensive and won’t be covered in full detail here. If you want to learn more, you can check out the Real Python article [Python 3’s f-Strings: An Improved String Formatting Syntax \(Guide\)](#). There is also a tutorial on Formatted Output coming up later in this series that digs deeper into f-strings.

One simple feature of f-strings you can start using right away is variable interpolation. You can specify a variable name directly within an f-string literal, and Python will replace the name with the corresponding value.

For example, suppose you want to display the result of an arithmetic calculation. You can do this with a straightforward `print()` statement, separating numeric values and string literals by commas:

```
Python  
>>> n = 20  
>>> m = 25  
>>> prod = n * m  
>>> print('The product of', n, 'and', m, 'is', prod)  
The product of 20 and 25 is 500
```

But this is cumbersome. To accomplish the same thing using an f-string:

- Specify either a lowercase `f` or uppercase `F` directly before the opening quote of the string literal. This tells

Python it is an f-string instead of a standard string.

- Specify any variables to be interpolated in curly braces {}).

Recast using an f-string, the above example looks much cleaner:

```
Python >>>
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print(f'The product of {n} and {m} is {prod}')
The product of 20 and 25 is 500
```

Any of Python's three quoting mechanisms can be used to define an f-string:

```
Python >>>
>>> var = 'Bark'
>>> print(f'A dog says {var}!')
A dog says Bark!
>>> print(f"A dog says {var}!")
A dog says Bark!
>>> print(f'''A dog says {var}!''')
A dog says Bark!
```

Modifying Strings

In a nutshell, you can't. Strings are one of the data types Python considers immutable, meaning not able to be changed. In fact, all the data types you have seen so far are immutable. (Python does provide data types that are mutable, as you will soon see.)

A statement like this will cause an error:

```
Python >>>
>>> s = 'foobar'
>>> s[3] = 'x'
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    s[3] = 'x'
TypeError: 'str' object does not support item assignment
```

In truth, there really isn't much need to modify strings. You can usually easily accomplish what you want by generating a copy of the original string that has the desired change in place. There are very many ways to do this in Python. Here is one possibility:

```
Python >>>
>>> s = s[:3] + 'x' + s[4:]
>>> s
'fooxar'
```

There is also a built-in string method to accomplish this:

```
Python >>>
>>> s = 'foobar'
>>> s.replace('b', 'x')
>>> s
'fooxar'
```

Read on for more information about built-in string methods!

Built-in String Methods

You learned in the tutorial on [Variables in Python](#) that Python is a highly object-oriented language. Every item of data in a Python program is an object.

You are also familiar with functions: callable procedures that you can invoke to perform specific tasks.

Methods are similar to functions. A method is a specialized type of callable procedure that is tightly associated with an object. Like a function, a method is called to perform a distinct task, but it is invoked on a specific object and has knowledge of its target object during execution.

The syntax for invoking a method on an object is as follows:

Python

```
obj.foo(<args>)
```

This invokes method `.foo()` on object `obj`. `<args>` specifies the arguments passed to the method (if any).

You will explore much more about defining and calling methods later in the discussion of object-oriented programming. For now, the goal is to present some of the more commonly used built-in methods Python supports for operating on string objects.

In the following method definitions, arguments specified in square brackets (`[]`) are optional.

Case Conversion

Methods in this group perform case conversion on the target string.

`s.capitalize()`

Capitalizes the target string.

`s.capitalize()` returns a copy of `s` with the first character converted to uppercase and all other characters converted to lowercase:

Python

>>>

```
>>> s = 'fo0 BaR BAZ quX'  
>>> s.capitalize()  
'Foo bar baz qux'
```

Non-alphabetic characters are unchanged:

Python

>>>

```
>>> s = 'foo123#BAR#. '  
>>> s.capitalize()  
'Foo123#bar#. '
```

`s.lower()`

Converts alphabetic characters to lowercase.

`s.lower()` returns a copy of `s` with all alphabetic characters converted to lowercase:

Python

>>>

```
>>> 'FOO Bar 123 baz qUX'.lower()  
'foo bar 123 baz qux'
```

```
s.swapcase()
```

Swaps case of alphabetic characters.

s.swapcase() returns a copy of s with uppercase alphabetic characters converted to lowercase and vice versa:

Python

>>>

```
>>> 'FOO Bar 123 baz qUX'.swapcase()  
'foo bAR 123 BAZ Qux'
```

```
s.title()
```

Converts the target string to “title case.”

s.title() returns a copy of s in which the first letter of each word is converted to uppercase and remaining letters are lowercase:

Python

>>>

```
>>> 'the sun also rises'.title()  
'The Sun Also Rises'
```

This method uses a fairly simple algorithm. It does not attempt to distinguish between important and unimportant words, and it does not handle apostrophes, possessives, or acronyms gracefully:

Python

>>>

```
>>> "what's happened to ted's IBM stock?".title()  
"what'S Happened To Ted'S Ibm Stock?"
```

```
s.upper()
```

Converts alphabetic characters to uppercase.

s.upper() returns a copy of s with all alphabetic characters converted to uppercase:

Python

>>>

```
>>> 'FOO Bar 123 baz qUX'.upper()  
'FOO BAR 123 BAZ QUX'
```

Find and Replace

These methods provide various means of searching the target string for a specified substring.

Each method in this group supports optional `<start>` and `<end>` arguments. These are interpreted as for string slicing: the action of the method is restricted to the portion of the target string starting at character position `<start>` and proceeding up to but not including character position `<end>`. If `<start>` is specified but `<end>` is not, the method applies to the portion of the target string from `<start>` through the end of the string.

```
s.count(<sub>[, <start>[, <end>]])
```

Counts occurrences of a substring in the target string.

s.count(<sub>) returns the number of non-overlapping occurrences of substring <sub> in s:

Python

>>>

```
>>> 'foo goo moo'.count('oo')
3
```

The count is restricted to the number of occurrences within the substring indicated by <start> and <end>, if they are specified:

Python

>>>

```
>>> 'foo goo moo'.count('oo', 0, 8)
2
```

s.endswith(<suffix>[, <start>[, <end>]])

Determines whether the target string ends with a given substring.

s.endswith(<suffix>) returns True if s ends with the specified <suffix> and False otherwise:

Python

>>>

```
>>> 'foobar'.endswith('bar')
True
>>> 'foobar'.endswith('baz')
False
```

The comparison is restricted to the substring indicated by <start> and <end>, if they are specified:

Python

>>>

```
>>> 'foobar'.endswith('oob', 0, 4)
True
>>> 'foobar'.endswith('oob', 2, 4)
False
```

s.find(<sub>[, <start>[, <end>]])

Searches the target string for a given substring.

You can use .find() to see if a Python string contains a particular substring. s.find(<sub>) returns the lowest index in s where substring <sub> is found:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.find('foo')
0
```

This method returns -1 if the specified substring is not found:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.find('grault')
-1
```

The search is restricted to the substring indicated by <start> and <end>, if they are specified:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.find('foo', 4)
8
>>> 'foo bar foo baz foo qux'.find('foo', 4, 7)
-1
```

```
s.index(<sub>[, <start>[, <end>]])
```

Searches the target string for a given substring.

This method is identical to `.find()`, except that it raises an exception if `<sub>` is not found rather than returning `-1`:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.index('grault')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'foo bar foo baz foo qux'.index('grault')
ValueError: substring not found
```

```
s.rfind(<sub>[, <start>[, <end>]])
```

Searches the target string for a given substring starting at the end.

`s.rfind(<sub>)` returns the highest index in `s` where substring `<sub>` is found:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.rfind('foo')
16
```

As with `.find()`, if the substring is not found, `-1` is returned:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.rfind('grault')
-1
```

The search is restricted to the substring indicated by `<start>` and `<end>`, if they are specified:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.rfind('foo', 0, 14)
8
>>> 'foo bar foo baz foo qux'.rfind('foo', 10, 14)
-1
```

```
s.rindex(<sub>[, <start>[, <end>]])
```

Searches the target string for a given substring starting at the end.

This method is identical to `.rfind()`, except that it raises an exception if `<sub>` is not found rather than returning `-1`:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.rindex('grault')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    'foo bar foo baz foo qux'.rindex('grault')
ValueError: substring not found
```

`s.startswith(<prefix>[, <start>[, <end>]])`

Determines whether the target string starts with a given substring.

When you use the Python `.startswith()` method, `s.startswith(<suffix>)` returns `True` if `s` starts with the specified `<suffix>` and `False` otherwise:

Python

>>>

```
>>> 'foobar'.startswith('foo')
True
>>> 'foobar'.startswith('bar')
False
```

The comparison is restricted to the substring indicated by `<start>` and `<end>`, if they are specified:

Python

>>>

```
>>> 'foobar'.startswith('bar', 3)
True
>>> 'foobar'.startswith('bar', 3, 2)
False
```

Character Classification

Methods in this group classify a string based on the characters it contains.

`s.isalnum()`

Determines whether the target string consists of alphanumeric characters.

`s.isalnum()` returns `True` if `s` is nonempty and all its characters are alphanumeric (either a letter or a number), and `False` otherwise:

Python

>>>

```
>>> 'abc123'.isalnum()
True
>>> 'abc$123'.isalnum()
False
>>> ''.isalnum()
False
```

`s.isalpha()`

Determines whether the target string consists of alphabetic characters.

`s.isalpha()` returns `True` if `s` is nonempty and all its characters are alphabetic, and `False` otherwise:

Python

>>>

```
>>> 'ABCabc'.isalpha()
True
>>> 'abc123'.isalpha()
False
```

`s.isdialt()`

Determines whether the target string consists of digit characters.

You can use the `.isdigit()` Python method to check if your string is made of only digits. `s.isdigit()` returns `True` if `s` is nonempty and all its characters are numeric digits, and `False` otherwise:

Python

```
>>> '123'.isdigit()
True
>>> '123abc'.isdigit()
False
```

>>>

`s.isidentifier()`

Determines whether the target string is a valid Python identifier.

`s.isidentifier()` returns `True` if `s` is a valid Python identifier according to the language definition, and `False` otherwise:

Python

```
>>> 'foo32'.isidentifier()
True
>>> '32foo'.isidentifier()
False
>>> 'foo$32'.isidentifier()
False
```

>>>

Note: `.isidentifier()` will return `True` for a string that matches a [Python keyword](#) even though that would not actually be a valid identifier:

Python

```
>>> 'and'.isidentifier()
True
```

>>>

You can test whether a string matches a Python keyword using a function called `iskeyword()`, which is contained in a module called `keyword`. One possible way to do this is shown below:

Python

```
>>> from keyword import iskeyword
>>> iskeyword('and')
True
```

>>>

If you really want to ensure that a string would serve as a valid Python identifier, you should check that `.isidentifier()` is `True` and that `iskeyword()` is `False`.

See [Python Modules and Packages—An Introduction](#) to read more about Python modules.

`s.islower()`

Determines whether the target string's alphabetic characters are lowercase.

`s.islower()` returns `True` if `s` is nonempty and all the alphabetic characters it contains are lowercase, and `False` otherwise. Non-alphabetic characters are ignored:

Python

>>>

```
>>> 'abc'.islower()
True
>>> 'abc1$d'.islower()
True
>>> 'Abc1$D'.islower()
False
```

s.isprintable()

Determines whether the target string consists entirely of printable characters.

`s.isprintable()` returns `True` if `s` is empty or all the alphabetic characters it contains are printable. It returns `False` if `s` contains at least one non-printable character. Non-alphabetic characters are ignored:

Python

>>>

```
>>> 'a\tb'.isprintable()
False
>>> 'a b'.isprintable()
True
>>> ''.isprintable()
True
>>> 'a\nb'.isprintable()
False
```

Note: This is the only `.isxxxx()` method that returns `True` if `s` is an empty string. All the others return `False` for an empty string.

s.isspace()

Determines whether the target string consists of whitespace characters.

`s.isspace()` returns `True` if `s` is nonempty and all characters are whitespace characters, and `False` otherwise.

The most commonly encountered whitespace characters are space ' ', tab '\t', and newline '\n':

Python

>>>

```
>>> '\t \n '.isspace()
True
>>> ' a '.isspace()
False
```

However, there are a few other ASCII characters that qualify as whitespace, and if you account for Unicode characters, there are quite a few beyond that:

Python

>>>

```
>>> '\f\u2005\r'.isspace()
True
```

('\f' and '\r' are the escape sequences for the ASCII Form Feed and Carriage Return characters; '\u2005' is the escape sequence for the Unicode Four-Per-Em Space.)

s.istitle()

Determines whether the target string is title cased.

`s.istitle()` returns `True` if `s` is nonempty, the first alphabetic character of each word is uppercase, and all other alphabetic characters in each word are lowercase. It returns `False` otherwise:

Python

>>>

```
>>> 'This Is A Title'.istitle()
True
>>> 'This is a title'.istitle()
False
>>> 'Give Me The ##@ Ball!'.istitle()
True
```

Note: Here is how the Python documentation describes `.istitle()`, in case you find this more intuitive:
“Uppercase characters may only follow uncased characters and lowercase characters only cased ones.”

`s.isupper()`

Determines whether the target string's alphabetic characters are uppercase.

`s.isupper()` returns `True` if `s` is nonempty and all the alphabetic characters it contains are uppercase, and `False` otherwise. Non-alphabetic characters are ignored:

Python

>>>

```
>>> 'ABC'.isupper()
True
>>> 'ABC1$D'.isupper()
True
>>> 'Abc1$D'.isupper()
False
```

String Formatting

Methods in this group modify or enhance the format of a string.

`s.center(<width>[, <fill>])`

Centers a string in a field.

`s.center(<width>)` returns a string consisting of `s` centered in a field of width `<width>`. By default, padding consists of the ASCII space character:

Python

>>>

```
>>> 'foo'.center(10)
'    foo    '
```

If the optional `<fill>` argument is specified, it is used as the padding character:

Python

>>>

```
>>> 'bar'.center(10, '-')
'---bar---'
```

If `s` is already at least as long as `<width>`, it is returned unchanged:

Python

>>>

```
>>> 'foo'.center(2)
'foo'
```

```
s.expandtabs(tabsize=8)
```

`expand_tabs`

`s.expandtabs()` replaces each tab character ('\t') with spaces. By default, spaces are filled in assuming a tab stop at every eighth column:

Python

>>>

```
>>> 'a\tb\tc'.expandtabs()
'a      b      c'
>>> 'aaa\tbbb\tc'.expandtabs()
'aaa      bbb      c'
```

`tabsize` is an optional keyword parameter specifying alternate tab stop columns:

Python

>>>

```
>>> 'a\tb\tc'.expandtabs(4)
'a    b    c'
>>> 'aaa\tbbb\tc'.expandtabs(tabsize=4)
'aaa  bbb  c'
```

```
s.ljust(<width>[, <fill>])
```

Left-justifies a string in field.

`s.ljust(<width>)` returns a string consisting of `s` left-justified in a field of width `<width>`. By default, padding consists of the ASCII space character:

Python

>>>

```
>>> 'foo'.ljust(10)
'foo      '
```

If the optional `<fill>` argument is specified, it is used as the padding character:

Python

>>>

```
>>> 'foo'.ljust(10, '-')
'foo-----'
```

If `s` is already at least as long as `<width>`, it is returned unchanged:

Python

>>>

```
>>> 'foo'.ljust(2)  
'foo'
```

s.lstrip([<chars>])

Trims leading characters from a string.

`s.lstrip()` returns a copy of `s` with any whitespace characters removed from the left end:

Python

3

```
>>> '    foo bar baz    '.lstrip()
'foo bar baz    '
>>> '\t\nfoo\t\nbar\t\nbaz'.lstrip()
'foo\t\nbar\t\nbaz'
```

```
>>> 'foo\nbar\nbaz'.lstrip()
'foo\nbar\nbaz'
```

If the optional <chars> argument is specified, it is a string that specifies the set of characters to be removed:

Python

>>>

```
>>> 'http://www.realpython.com'.lstrip('/:pth')
'www.realpython.com'
```

```
s.replace(<old>, <new>[, <count>])
```

Replaces occurrences of a substring within a string.

In Python, to remove a character from a string, you can use the Python string .replace() method.

s.replace(<old>, <new>) returns a copy of s with all occurrences of substring <old> replaced by <new>:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.replace('foo', 'grault')
'grault bar grault baz grault qux'
```

If the optional <count> argument is specified, a maximum of <count> replacements are performed, starting at the left end of s:

Python

>>>

```
>>> 'foo bar foo baz foo qux'.replace('foo', 'grault', 2)
'grault bar grault baz foo qux'
```

```
s.rjust(<width>[, <fill>])
```

Right-justifies a string in a field.

s.rjust(<width>) returns a string consisting of s right-justified in a field of width <width>. By default, padding consists of the ASCII space character:

Python

>>>

```
>>> 'foo'.rjust(10)
'      foo'
```

If the optional <fill> argument is specified, it is used as the padding character:

Python

>>>

```
>>> 'foo'.rjust(10, '-')
'-----foo'
```

If s is already at least as long as <width>, it is returned unchanged:

Python

>>>

```
>>> 'foo'.rjust(2)
'foo'
```

```
s.rstrip([<chars>])
```

Trims trailing characters from a string.

`s.rstrip()` returns a copy of `s` with any whitespace characters removed from the right end:

Python

>>>

```
>>> ' foo bar baz '.rstrip()
'foo bar baz'
>>> 'foo\t\nbar\t\nbaz\t\n'.rstrip()
'foo\t\nbar\t\nbaz'
```

If the optional `<chars>` argument is specified, it is a string that specifies the set of characters to be removed:

Python

>>>

```
>>> 'foo.$$$;'.rstrip(';$.')
'foo'
```

`s.strip([<chars>])`

Strips characters from the left and right ends of a string.

`s.strip()` is essentially equivalent to invoking `s.lstrip()` and `s.rstrip()` in succession. Without the `<chars>` argument, it removes leading and trailing whitespace:

Python

>>>

```
>>> s = ' foo bar baz\t\t\t'
>>> s = s.lstrip()
>>> s = s.rstrip()
>>> s
'foo bar baz'
```

As with `.lstrip()` and `.rstrip()`, the optional `<chars>` argument specifies the set of characters to be removed:

Python

>>>

```
>>> 'www.realpython.com'.strip('w.moc')
'realpython'
```

Note: When the return value of a string method is another string, as is often the case, methods can be invoked in succession by chaining the calls:

Python

>>>

```
>>> ' foo bar baz\t\t\t'.lstrip().rstrip()
'foo bar baz'
>>> ' foo bar baz\t\t\t'.strip()
'foo bar baz'

>>> 'www.realpython.com'.lstrip('w.moc').rstrip('w.moc')
'realpython'
>>> 'www.realpython.com'.strip('w.moc')
'realpython'
```

`s.zfill(<width>)`

Pads a string on the left with zeros

LEADS A STRING ON THE LEFT WITH ZEROS.

`s.zfill(<width>)` returns a copy of `s` left-padded with '0' characters to the specified `<width>`:

Python

>>>

```
>>> '42'.zfill(5)
'00042'
```

If `s` contains a leading sign, it remains at the left edge of the result string after zeros are inserted:

Python

>>>

```
>>> '+42'.zfill(8)
'+0000042'
>>> '-42'.zfill(8)
'-0000042'
```

If `s` is already at least as long as `<width>`, it is returned unchanged:

Python

>>>

```
>>> '-42'.zfill(3)
'-42'
```

`.zfill()` is most useful for string representations of numbers, but Python will still happily zero-pad a string that isn't:

Python

>>>

```
>>> 'foo'.zfill(6)
'000foo'
```

Converting Between Strings and Lists

Methods in this group convert between a string and some composite data type by either pasting objects together to make a string, or by breaking a string up into pieces.

These methods operate on or return **iterables**, the general Python term for a sequential collection of objects. You will explore the inner workings of iterables in much more detail in the upcoming tutorial on definite iteration.

Many of these methods return either a list or a tuple. These are two similar composite data types that are prototypical examples of iterables in Python. They are covered in the next tutorial, so you're about to learn about them soon! Until then, simply think of them as sequences of values. A list is enclosed in square brackets ([]), and a tuple is enclosed in parentheses (()).

With that introduction, let's take a look at this last group of string methods.

`s.join(<iterable>)`

Concatenates strings from an iterable.

`s.join(<iterable>)` returns the string that results from concatenating the objects in `<iterable>` separated by `s`.

Note that `.join()` is invoked on `s`, the separator string. `<iterable>` must be a sequence of string objects as well.

Some sample code should help clarify. In the following example, the separator `s` is the string ', ', and `<iterable>` is a list of string values:

Python

>>>

```
>>> ', '.join(['foo', 'bar', 'baz', 'qux'])
'foo, bar, baz, qux'
```

The result is a single string consisting of the list objects separated by commas.

In the next example, <iterable> is specified as a single string value. When a string value is used as an iterable, it is interpreted as a list of the string's individual characters:

```
Python >>>
>>> list('corge')
['c', 'o', 'r', 'g', 'e']

>>> ':' .join('corge')
'c:o:r:g:e'
```

Thus, the result of ':'.join('corge') is a string consisting of each character in 'corge' separated by ':'.

This example fails because one of the objects in <iterable> is not a string:

```
Python >>>
>>> '-' .join(['foo', 23, 'bar'])
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '-' .join(['foo', 23, 'bar'])
TypeError: sequence item 1: expected str instance, int found
```

That can be remedied, though:

```
Python >>>
>>> '-' .join(['foo', str(23), 'bar'])
'foo---23---bar'
```

As you will soon see, many composite objects in Python can be construed as iterables, and .join() is especially useful for creating strings from them.

s.partition(<sep>)

Divides a string based on a separator.

s.partition(<sep>) splits s at the first occurrence of string <sep>. The return value is a three-part tuple consisting of:

- The portion of s preceding <sep>
- <sep> itself
- The portion of s following <sep>

Here are a couple examples of .partition() in action:

```
Python >>>
>>> 'foo.bar' .partition('.')
('foo', '.', 'bar')
>>> 'foo@@bar@@baz' .partition('@@')
('foo', '@@', 'bar@@baz')
```

If <sep> is not found in s, the returned tuple contains s followed by two empty strings:

```
Python >>>
>>> 'foo.bar' .partition('@@')
('foo.bar', '', '')
```

Remember: Lists and tuples are covered in the next tutorial.

```
s.rpartition(<sep>)
```

Divides a string based on a separator.

`s.rpartition(<sep>)` functions exactly like `s.partition(<sep>)`, except that `s` is split at the last occurrence of `<sep>` instead of the first occurrence:

Python

>>>

```
>>> 'foo@@bar@@baz'.partition('@@')
('foo', ' @@', 'bar@@baz')

>>> 'foo@@bar@@baz'.rpartition('@@')
('foo@@bar', ' @@', 'baz')
```

```
s.rsplit(sep=None, maxsplit=-1)
```

Splits a string into a list of substrings.

Without arguments, `s.rsplit()` splits `s` into substrings delimited by any sequence of whitespace and returns the substrings as a list:

Python

>>>

```
>>> 'foo bar baz qux'.rsplit()
['foo', 'bar', 'baz', 'qux']
>>> 'foo\n\tbar    baz\r\fqux'.rsplit()
['foo', 'bar', 'baz', 'qux']
```

If `<sep>` is specified, it is used as the delimiter for splitting:

Python

>>>

```
>>> 'foo.bar.baz.qux'.rsplit(sep='.')
['foo', 'bar', 'baz', 'qux']
```

(If `<sep>` is specified with a value of `None`, the string is split delimited by whitespace, just as though `<sep>` had not been specified at all.)

When `<sep>` is explicitly given as a delimiter, consecutive delimiters in `s` are assumed to delimit empty strings, which will be returned:

Python

>>>

```
>>> 'foo...bar'.rsplit(sep='.')
['foo', '', '', 'bar']
```

This is not the case when `<sep>` is omitted, however. In that case, consecutive whitespace characters are combined into a single delimiter, and the resulting list will never contain empty strings:

Python

>>>

```
>>> 'foo\t\t\tbar'.rsplit()
['foo', 'bar']
```

If the optional keyword parameter `<maxsplit>` is specified, a maximum of that many splits are performed, starting from the right end of `s`:

Python

>>>

```
>>> 'www.realpython.com'.rsplit(sep='.', maxsplit=1)
['www.realpython', 'com']
```

The default value for `<maxsplit>` is `-1`, which means all possible splits should be performed—the same as if

The default value for `<maxsplit>` is `-1`, which means all possible splits should be performed – the same as if `<maxsplit>` is omitted entirely:

Python

>>>

```
>>> 'www.realpython.com'.rsplit(sep='.', maxsplit=-1)
['www', 'realpython', 'com']
>>> 'www.realpython.com'.rsplit(sep='.')
['www', 'realpython', 'com']
```

`s.split(sep=None, maxsplit=-1)`

Splits a string into a list of substrings.

`s.split()` behaves exactly like `s.rsplit()`, except that if `<maxsplit>` is specified, splits are counted from the left end of `s` rather than the right end:

Python

>>>

```
>>> 'www.realpython.com'.split('.', maxsplit=1)
['www', 'realpython.com']
>>> 'www.realpython.com'.rsplit('.', maxsplit=1)
['www.realpython', 'com']
```

If `<maxsplit>` is not specified, `.split()` and `.rsplit()` are indistinguishable.

`s.splitlines([<keepends>])`

Breaks a string at line boundaries.

`s.splitlines()` splits `s` up into lines and returns them in a list. Any of the following characters or character sequences is considered to constitute a line boundary:

Escape Sequence	Character
\n	Newline
\r	Carriage Return
\r\n	Carriage Return + Line Feed
\v or \x0b	Line Tabulation
\f or \x0c	Form Feed
\x1c	File Separator
\x1d	Group Separator
\x1e	Record Separator
\x85	Next Line (C1 Control Code)
\u2028	Unicode Line Separator
\u2029	Unicode Paragraph Separator

Here is an example using several different line separators:

Python

>>>

```
>>> 'foo\nbar\r\nbaz\fqux\u2028quux'.splitlines()
['foo', 'bar', 'baz', 'qux', 'quux']
```

If consecutive line boundary characters are present in the string, they are assumed to delimit blank lines, which will appear in the result list:

Python

>>>

```
>>> 'foo\f\f\fbar'.splitlines()
['foo', '', '', 'bar']
```

If the optional <keepends> argument is specified and is truthy, then the lines boundaries are retained in the result strings:

Python

>>>

```
>>> 'foo\nbar\nbaz\nquux'.splitlines(True)
['foo\n', 'bar\n', 'baz\n', 'quux']
>>> 'foo\nbar\nbaz\nquux'.splitlines(1)
['foo\n', 'bar\n', 'baz\n', 'quux']
```

bytes Objects

The bytes object is one of the core built-in types for manipulating binary data. A bytes object is an immutable sequence of single [byte](#) values. Each element in a bytes object is a small integer in the range 0 to 255.

Defining a Literal bytes Object

A bytes literal is defined in the same way as a string literal with the addition of a 'b' prefix:

Python

>>>

```
>>> b = b'foo bar baz'
>>> b
b'foo bar baz'
>>> type(b)
<class 'bytes'>
```

As with strings, you can use any of the single, double, or triple quoting mechanisms:

Python

>>>

```
>>> b'Contains embedded "double" quotes'
b'Contains embedded "double" quotes'

>>> b"Contains embedded 'single' quotes"
b"Contains embedded 'single' quotes"

>>> b'''Contains embedded "double" and 'single' quotes'''
b'Contains embedded "double" and \single\ quotes'

>>> b""""Contains embedded "double" and 'single' quotes"""
b'Contains embedded "double" and \single\ quotes'
```

Only ASCII characters are allowed in a bytes literal. Any character value greater than 127 must be specified using an appropriate escape sequence:

Python

>>>

```
>>> b = b'foo\xddbar'
>>> b
b'foo\xddbar'
>>> b[3]
221
>>> int(0xdd)
221
```

The 'r' prefix may be used on a bytes literal to disable processing of escape sequences, as with strings:

Python

>>>

```
>>> b = rb'foo\xddbar'  
>>> b  
b'foo\\xddbar'  
>>> b[3]  
92  
>>> chr(92)  
'\\'
```

Defining a bytes Object With the Built-in bytes() Function

The bytes() function also creates a bytes object. What sort of bytes object gets returned depends on the argument(s) passed to the function. The possible forms are shown below.

`bytes(<s>, <encoding>)`

Creates a bytes object from a string.

`bytes(<s>, <encoding>)` converts string <s> to a bytes object, using `str.encode()` according to the specified <encoding>:

Python

>>>

```
>>> b = bytes('foo.bar', 'utf8')  
>>> b  
b'foo.bar'  
>>> type(b)  
<class 'bytes'>
```

Technical Note: In this form of the bytes() function, the <encoding> argument is required. “Encoding” refers to the manner in which characters are translated to integer values. A value of "utf8" indicates Unicode Transformation Format **UTF-8**, which is an encoding that can handle every possible Unicode character. UTF-8 can also be indicated by specifying "UTF8", "utf-8", or "UTF-8" for <encoding>.

See the [Unicode documentation](#) for more information. As long as you are dealing with common Latin-based characters, UTF-8 will serve you fine.

`bytes(<size>)`

Creates a bytes object consisting of null (0x00) bytes.

`bytes(<size>)` defines a bytes object of the specified <size>, which must be a positive integer. The resulting bytes object is initialized to null (0x00) bytes:

Python

>>>

```
>>> b = bytes(8)  
>>> b  
b'\x00\x00\x00\x00\x00\x00\x00\x00'
```

`bytes(<iterable>)`

Creates a bytes object from an iterable.

`bytes(<iterable>)` defines a bytes object from the sequence of integers generated by `<iterable>.iterable`. `<iterable>` must be an iterable that generates a sequence of integers n in the range $0 \leq n \leq 255$:

Python

>>>

```
>>> b = bytes([100, 102, 104, 106, 108])
>>> b
b'dfhjl'
>>> type(b)
<class 'bytes'>
>>> b[2]
104
```

Operations on bytes Objects

Like strings, bytes objects support the common sequence operations:

- The `in` and `not in` operators:

Python

>>>

```
>>> b = b'abcde'
>>> b'cd' in b
True
>>> b'foo' not in b
True
```

- The concatenation (+) and replication (*) operators:

Python

>>>

```
>>> b = b'abcde'
>>> b + b'fghi'
b'abcdefghi'
>>> b * 3
b'abcdeabcdeabcde'
```

- Indexing and slicing:

Python

>>>

```
>>> b = b'abcde'
>>> b[2]
99
>>> b[1:3]
b'bc'
```

- Built-in functions:

Python

>>>

```
>>> len(b)
5
>>> min(b)
97
>>> max(b)
101
```

Many of the methods defined for string objects are valid for bytes objects as well:

```

Python >>>
>>> b = b'foo,bar,foo,baz,foo,qux'
>>> b.count(b'foo')
3

>>> b.endswith(b'qux')
True

>>> b.find(b'baz')
12

>>> b.split(sep=b',')
[b'foo', b'bar', b'foo', b'baz', b'foo', b'qux']

>>> b.center(30, b'-' )
b'---foo,bar,foo,baz,foo,qux----'

```

Notice, however, that when these operators and methods are invoked on a bytes object, the operand and arguments must be bytes objects as well:

```

Python >>>
>>> b = b'foo.bar'

>>> b + '.baz'
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    b + '.baz'
TypeError: can't concat bytes to str
>>> b + b'.baz'
b'foo.bar.baz'

>>> b.split(sep='.')
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    b.split(sep='.')
TypeError: a bytes-like object is required, not 'str'
>>> b.split(sep=b'.')
[b'foo', b'bar']

```

Although a bytes object definition and representation is based on ASCII text, it actually behaves like an immutable sequence of small integers in the range 0 to 255, inclusive. That is why a single element from a bytes object is displayed as an integer:

```

Python >>>
>>> b = b'foo\xddbar'
>>> b[3]
221
>>> hex(b[3])
'0xdd'
>>> min(b)
97
>>> max(b)
221

```

A slice is displayed as a bytes object though, even if it is only one byte long:

```

Python >>>
>>> b[2:3]
b'c'

```

You can convert a bytes object into a list of integers with the built-in `list()` function:

```

Python >>>
>>> list(b)
[97, 98, 99, 100, 101]

```

Hexadecimal numbers are often used to specify binary data because two hexadecimal digits correspond directly to a single byte. The bytes class supports two additional methods that facilitate conversion to and from a string of hexadecimal digits.

bytes.fromhex(<s>)

Returns a bytes object constructed from a string of hexadecimal values.

bytes.fromhex(<s>) returns the bytes object that results from converting each pair of hexadecimal digits in <s> to the corresponding byte value. The hexadecimal digit pairs in <s> may optionally be separated by whitespace, which is ignored:

Python

```
>>> b = bytes.fromhex(' aa 68 4682cc ')
>>> b
b'\xaahF\x82\xcc'
>>> list(b)
[170, 104, 70, 130, 204]
```

>>>

Note: This method is a class method, not an object method. It is bound to the bytes class, not a bytes object. You will delve much more into the distinction between classes, objects, and their respective methods in the upcoming tutorials on [object-oriented programming](#). For now, just observe that this method is invoked on the bytes class, not on object b.

b.hex()

Returns a string of hexadecimal value from a bytes object.

b.hex() returns the result of converting bytes object b into a string of hexadecimal digit pairs. That is, it does the reverse of .fromhex():

Python

```
>>> b = bytes.fromhex(' aa 68 4682cc ')
>>> b
b'\xaahF\x82\xcc'

>>> b.hex()
'aa684682cc'
>>> type(b.hex())
<class 'str'>
```

>>>

Note: As opposed to .fromhex(), .hex() is an object method, not a class method. Thus, it is invoked on an object of the bytes class, not on the class itself.

bytearray Objects

Python supports another binary sequence type called the bytearray. bytearray objects are very like bytes objects, despite some differences:

- There is no dedicated syntax built into Python for defining a bytearray literal, like the 'b' prefix that may be used to define a bytes object. A bytearray object is always created using the bytearray() built-in function:

Python

```
>>> ba = bytearray('foo.bar.baz', 'UTF-8')
>>> ba
bytearray(b'foo.bar.baz')
```

>>>

```
>>> bytearray(6)
bytearray(b'\x00\x00\x00\x00\x00\x00')

>>> bytearray([100, 102, 104, 106, 108])
bytearray(b'dfhj1')
```

- `bytearray` objects are mutable. You can modify the contents of a `bytearray` object using indexing and slicing:

```
Python >>>

>>> ba = bytearray('foo.bar.baz', 'UTF-8')
>>> ba
bytearray(b'foo.bar.baz')

>>> ba[5] = 0xee
>>> ba
bytearray(b'foo.bxeer.baz')

>>> ba[8:11] = b'qux'
>>> ba
bytearray(b'foo.bxeer.qux')
```

A `bytearray` object may be constructed directly from a `bytes` object as well:

```
Python >>>

>>> ba = bytearray(b'foo')
>>> ba
bytearray(b'foo')
```

Conclusion

This tutorial provided an in-depth look at the many different mechanisms Python provides for **string** handling, including string operators, built-in functions, indexing, slicing, and built-in methods. You also were introduced to the `bytes` and `bytearray` types.

These types are the first types you have examined that are composite—built from a collection of smaller parts. Python provides several composite built-in types. In the next tutorial, you will explore two of the most frequently used: **lists** and **tuples**.

 **Take the Quiz:** Test your knowledge with our interactive “Python Strings and Character Data” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

[« Operators and Expressions
in Python](#)

[Strings in Python](#)

[Lists and Tuples in Python »](#)

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Strings and Character Data in Python](#)

About John Sturtz

John is an avid Pythonista and a member of the Real Python tutorial team.

[» More about John](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Dan](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [Strings and Character Data in Python](#)



Splitting, Concatenating, and Joining Strings in Python

by [Kyle Stratis](#) 10 Comments basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Splitting Strings](#)
 - [Splitting Without Parameters](#)
 - [Specifying Separators](#)
 - [Limiting Splits With Maxsplit](#)
- [Concatenating and Joining Strings](#)
 - [Concatenating With the + Operator](#)
 - [Going From a List to a String in Python With .join\(\)](#)
- [Tying It All Together](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Splitting, Concatenating, and Joining Strings in Python](#)

There are few guarantees in life: death, taxes, and programmers needing to deal with strings. Strings can come in many forms. They could be unstructured text, usernames, product descriptions, database column names, or really anything else that we describe using language.

With the near-ubiquity of string data, it's important to master the tools of the trade when it comes to strings. Luckily, Python makes string manipulation very simple, especially when compared to other languages and even older versions of Python.

In this article, you will learn some of the most fundamental string operations: splitting, concatenating, and joining. Not only will you learn how to use these tools, but you will walk away with a deeper understanding of how they work under the hood.

 **Take the Quiz:** Test your knowledge with our interactive “Splitting, Concatenating, and Joining Strings in Python” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

Free Bonus: [Click here to get a Python Cheat Sheet](#) and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Splitting Strings

In Python, strings are represented as `str` objects, which are **immutable**: this means that the object as represented in memory can not be directly altered. These two facts can help you learn (and then remember) how to use `.split()`.

Have you guessed how those two features of strings relate to splitting functionality in Python? If you guessed that `.split()` is an **instance method** because strings are a special type, you would be correct! In some other languages (like Perl), the original string serves as an input to a standalone `.split()` function rather than a method called on the string itself.

Note: Ways to Call String Methods

String methods like `.split()` are mainly shown here as instance methods that are called on strings. They can also be called as static methods, but this isn't ideal because it's more "wordy." For the sake of completeness, here's an example:

Python

```
# Avoid this:  
str.split('a,b,c', ',')
```

This is bulky and unwieldy when you compare it to the preferred usage:

Python

```
# Do this instead:  
'a,b,c'.split(',')
```

For more on instance, class, and static methods in Python, check out our [in-depth tutorial](#).

What about string immutability? This should remind you that string methods are not **in-place operations**, but they return a *new* object in memory.

Note: In-Place Operations

In-place operations are operations that directly change the object on which they are called. A common example is the `.append()` method that is used on lists: when you call `.append()` on a list, that list is directly changed by adding the input to `.append()` to the same list.

Splitting Without Parameters

Before going deeper, let's look at a simple example:

Python

>>>

```
>>> 'this is my string'.split()  
['this', 'is', 'my', 'string']
```

This is actually a special case of a `.split()` call, which I chose for its simplicity. Without any separator specified, `.split()` will count any whitespace as a separator.

Another feature of the bare call to `.split()` is that it automatically cuts out leading and trailing whitespace, as well as consecutive whitespace. Compare calling `.split()` on the following string without a separator parameter and with having `' '` as the separator parameter:

Python

>>>

```
>>> s = ' this  is my string '
>>> s.split()
['this', 'is', 'my', 'string']
>>> s.split(' ')
[ '', 'this', ' ', ' ', 'is', ' ', 'my', 'string', ' ']
```

The first thing to notice is that this showcases the immutability of strings in Python: subsequent calls to `.split()` work on the original string, not on the list result of the first call to `.split()`.

The second—and the main—thing you should see is that the bare `.split()` call extracts the words in the sentence and discards any whitespace.

Specifying Separators

`.split(' ')`, on the other hand, is much more literal. When there are leading or trailing separators, you'll get an empty string, which you can see in the first and last elements of the resulting list.

Where there are multiple consecutive separators (such as between "this" and "is" and between "is" and "my"), the first one will be used as the separator, and the subsequent ones will find their way into your result list as empty strings.

Note: Separators in Calls to `.split()`

While the above example uses a single space character as a separator input to `.split()`, you aren't limited in the types of characters or length of strings you use as separators. The only requirement is that your separator be a string. You could use anything from "..." to even "separator".

Limiting Splits With Maxsplit

`.split()` has another optional parameter called `maxsplit`. By default, `.split()` will make all possible splits when called. When you give a value to `maxsplit`, however, only the given number of splits will be made. Using our previous example string, we can see `maxsplit` in action:

Python

>>>

```
>>> s = "this is my string"
>>> s.split(maxsplit=1)
['this', 'is my string']
```

As you see above, if you set `maxsplit` to 1, the first whitespace region is used as the separator, and the rest are ignored. Let's do some exercises to test out everything we've learned so far.

Solution: "Try It Yourself: Maxsplit"

Show/Hide

Exercise: "Section Comprehension Check"

Show/Hide

Solution: "Section Comprehension Check"

Show/Hide

Concatenating and Joining Strings

The other fundamental string operation is the opposite of splitting strings: string **concatenation**. If you haven't seen this word, don't worry. It's just a fancy way of saying "gluing together."

Concatenating With the + Operator

Concatenating with the + Operator

There are a few ways of doing this, depending on what you're trying to achieve. The simplest and most common method is to use the plus symbol (+) to add multiple strings together. Simply place a + between as many strings as you want to join together:

Python

```
>>> 'a' + 'b' + 'c'  
'abc'
```

>>>

In keeping with the math theme, you can also multiply a string to repeat it:

Python

```
>>> 'do' * 2  
'dodo'
```

>>>

Remember, strings are immutable! If you concatenate or repeat a string stored in a variable, you will have to assign the new string to another variable in order to keep it.

Python

```
>>> orig_string = 'Hello'  
>>> orig_string + ', world'  
'Hello, world'  
>>> orig_string  
'Hello'  
>>> full_sentence = orig_string + ', world'  
>>> full_sentence  
'Hello, world'
```

>>>

If we didn't have immutable strings, `full_sentence` would instead output 'Hello, world, world'.

Another note is that Python does not do implicit string conversion. If you try to concatenate a string with a non-string type, Python [will raise a `TypeError`](#):

Python

```
>>> 'Hello' + 2  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: must be str, not int
```

>>>

This is because you can only concatenate strings with other strings, which may be new behavior for you if you're coming from a language like JavaScript, which attempts to do implicit type conversion.

Going From a List to a String in Python With `.join()`

There is another, more powerful, way to join strings together. You can go from a list to a string in Python with the `join()` method.

The common use case here is when you have an iterable—like a list—made up of strings, and you want to combine those strings into a single string. Like `.split()`, `.join()` is a string instance method. If all of your strings are in an iterable, which one do you call `.join()` on?

This is a bit of a trick question. Remember that when you use `.split()`, you call it on the string or character you want to split on. The opposite operation is `.join()`, so you call it on the string or character you want to use to join your iterable of strings together:

Python

```
>>> strings = ['do', 're', 'mi']  
>>> ','.join(strings)  
'do,re,mi'
```

>>>

Here, we join each element of the strings list with a comma (,) and call .join() on it rather than the strings list.

Exercise: "Readability Improvement with Join"

Show/Hide

Solution: "Readability Improvement with Join"

Show/Hide

.join() is smart in that it inserts your “joiner” in between the strings in the iterable you want to join, rather than just adding your joiner at the end of every string in the iterable. This means that if you pass an iterable of size 1, you won’t see your joiner:

Python

>>>

```
>>> 'b'.join(['a'])  
'a'
```

Exercise: "Section Comprehension Check"

Show/Hide

Solution: "Section Comprehension Check"

Show/Hide

Tying It All Together

While this concludes this overview of the most basic string operations in Python (splitting, concatenating, and joining), there is still a whole universe of string methods that can make your experiences with manipulating strings much easier.

Once you have mastered these basic string operations, you may want to learn more. Luckily, we have a number of great tutorials to help you complete your mastery of Python’s features that enable smart string manipulation:

- [Python 3’s f-Strings: An Improved String Formatting Syntax](#)
- [Python String Formatting Best Practices](#)
- [Strings and Character Data in Python](#)

 **Take the Quiz:** Test your knowledge with our interactive “Splitting, Concatenating, and Joining Strings in Python” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Splitting, Concatenating, and Joining Strings in Python](#)

About Kyle Stratis

Kyle is a self-taught developer working as a senior data engineer at Vizit Labs. In the past, he founded DanqEx (formerly Nasdanq: the original meme stock exchange) and Encryptid Gaming.

[» More about Kyle](#)

worked on this tutorial are:

[Adriana](#)

[Brad](#)

[Dan](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [Splitting, Concatenating, and Joining Strings in Python](#)



Python 3's f-Strings: An Improved String Formatting Syntax (Guide)

by Joanna Jablonski 54 Comments basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [“Old-school” String Formatting in Python](#)
 - [Option #1: %-formatting](#)
 - [Option #2: str.format\(\)](#)
- [f-Strings: A New and Improved Way to Format Strings in Python](#)
 - [Simple Syntax](#)
 - [Arbitrary Expressions](#)
 - [Multiline f-Strings](#)
 - [Speed](#)
- [Python f-Strings: The Pesky Details](#)
 - [Quotation Marks](#)
 - [Dictionaries](#)
 - [Braces](#)
 - [Backslashes](#)
 - [Inline Comments](#)
- [Go Forth and Format!](#)
- [Further Reading](#)

 **blackfire.io**
Profile & Optimize Python Apps Performance



Now available as
Public Beta
Sign-up for free and
install in minutes!

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python 3's f-Strings: An Improved String Formatting Syntax](#)

As of Python 3.6, f-strings are a great new way to format strings. Not only are they more readable, more concise, and less prone to error than other ways of formatting, but they are also faster!

By the end of this article, you will learn how and why to start using f-strings today.

But first, here's what life was like before f-strings, back when you had to walk to school uphill both ways in the snow.

[Free PDF Download: Python 3 Cheat Sheet](#)

“Old-school” String Formatting in Python

Before Python 3.6, you had two main ways of embedding Python expressions inside string literals for formatting: %-formatting and `str.format()`. You're about to see how to use them and what their limitations are.

Option #1: %-formatting

This is the OG of Python formatting and has been in the language since the very beginning. You can read more in the [Python docs](#). Keep in mind that %-formatting is not recommended by the docs, which contain the following note:

“The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly).

Using the newer formatted string literals or the `str.format()` interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text.” ([Source](#))

How to Use %-formatting

String objects have a built-in operation using the % operator, which you can use to format strings. Here's what that looks like in practice:

```
Python >>>
>>> name = "Eric"
>>> "Hello, %s." % name
'Hello, Eric.'
```

In order to insert more than one variable, you must use a tuple of those variables. Here's how you would do that:

```
Python >>>
>>> name = "Eric"
>>> age = 74
>>> "Hello, %s. You are %s." % (name, age)
'Hello Eric. You are 74.'
```

Why %-formatting Isn't Great

The code examples that you just saw above are readable enough. However, once you start using several parameters and longer strings, your code will quickly become much less easily readable. Things are starting to look a little messy already:

```
Python >>>
>>> first_name = "Eric"
>>> last_name = "Idle"
>>> age = 74
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> Hello %s %s. You are %s. You are a member of the %s. Your first name is %s. Your last name is %s.
```

```
>>> "Hello, %s. You are a %s. You are a member of %s." % (first_name, last_name, age)
'Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.'
```

Unfortunately, this kind of formatting isn't great because it is verbose and leads to errors, like not displaying tuples or dictionaries correctly. Fortunately, there are brighter days ahead.

Option #2: str.format()

This newer way of getting the job done was introduced in Python 2.6. You can check out the [Python docs](#) for more info.

How To Use str.format()

`str.format()` is an improvement on %-formatting. It uses normal function call syntax and is [extensible through the `__format__\(\)` method](#) on the object being converted to a string.

With `str.format()`, the replacement fields are marked by curly braces:

```
Python >>>
>>> "Hello, {}. You are {}.".format(name, age)
'Hello, Eric. You are 74.'
```

You can reference variables in any order by referencing their index:

```
Python >>>
>>> "Hello, {1}. You are {0}.".format(age, name)
'Hello, Eric. You are 74.'
```

But if you insert the variable names, you get the added perk of being able to pass objects and then reference parameters and methods in between the braces:

```
Python >>>
>>> person = {'name': 'Eric', 'age': 74}
>>> "Hello, {name}. You are {age}.".format(name=person['name'], age=person['age'])
'Hello, Eric. You are 74.'
```

You can also use `**` to do this neat trick with dictionaries:

```
Python >>>
>>> person = {'name': 'Eric', 'age': 74}
>>> "Hello, {name}. You are {age}.".format(**person)
'Hello, Eric. You are 74.'
```

`str.format()` is definitely an upgrade when compared with %-formatting, but it's not all roses and sunshine.

Why str.format() Isn't Great

Code using `str.format()` is much more easily readable than code using %-formatting, but `str.format()` can still be quite verbose when you are dealing with multiple parameters and longer strings. Take a look at this:

```
Python >>>
>>> first_name = "Eric"
>>> last_name = "Idle"
>>> age = 74
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> print(("Hello, {first_name} {last_name}. You are {age}. " +
>>>       "You are a {profession}. You were a member of {affiliation}.") \ 
>>>       .format(first_name=first_name, last_name=last_name, age=age, \
>>>             profession=profession, affiliation=affiliation))
'Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.'
```

If you had the variables you wanted to pass to `.format()` in a dictionary, then you could just unpack it with `.format(**some_dict)` and reference the values by key in the string, but there has got to be a better way to do this.

f-Strings: A New and Improved Way to Format Strings in Python

The good news is that f-strings are here to save the day. They slice! They dice! They make julienne fries! Okay, they do none of those things, but they do make formatting easier. They joined the party in Python 3.6. You can read all about it in [PEP 498](#), which was written by Eric V. Smith in August of 2015.

Also called “formatted string literals,” f-strings are string literals that have an `f` at the beginning and curly braces containing expressions that will be replaced with their values. The expressions are evaluated at runtime and then formatted using the `__format__` protocol. As always, the [Python docs](#) are your friend when you want to learn more.

Here are some of the ways f-strings can make your life easier.

Simple Syntax

The syntax is similar to the one you used with `str.format()` but less verbose. Look at how easily readable this is:

```
Python >>>
>>> name = "Eric"
>>> age = 74
>>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

It would also be valid to use a capital letter F:

```
Python >>>
>>> F"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

Do you love f-strings yet? I hope that, by the end of this article, you'll answer [>>> F"Yes!"](#).

Arbitrary Expressions

Because f-strings are evaluated at runtime, you can put any and all valid Python expressions in them. This allows you to do some nifty things.

You could do something pretty straightforward, like this:

```
Python >>>
>>> f"{2 * 37}"
'74'
```

But you could also call functions. Here's an example:

```
Python >>>
>>> def to_lowercase(input):
...     return input.lower()

>>> name = "Eric Idle"
>>> f"{to_lowercase(name)} is funny."
'eric idle is funny.'
```

You also have the option of calling a method directly:

```
Python >>>
>>> f"{name.lower()} is funny."
'eric idle is funny.'
```

You could even use objects created from classes with f-strings. Imagine you had the following class:

Python

```
class Comedian:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __str__(self):
        return f"{self.first_name} {self.last_name} is {self.age}."

    def __repr__(self):
        return f"{self.first_name} {self.last_name} is {self.age}. Surprise!"
```

You'd be able to do this:

Python

```
>>> new_comedian = Comedian("Eric", "Idle", "74")
>>> f"{new_comedian}"
'Eric Idle is 74.'
```

>>>

The `__str__()` and `__repr__()` methods deal with how objects are presented as strings, so you'll need to make sure you include at least one of those methods in your class definition. If you have to pick one, go with `__repr__()` because it can be used in place of `__str__()`.

The string returned by `__str__()` is the informal string representation of an object and should be readable. The string returned by `__repr__()` is the official representation and should be unambiguous. Calling `str()` and `repr()` is preferable to using `__str__()` and `__repr__()` directly.

By default, f-strings will use `__str__()`, but you can make sure they use `__repr__()` if you include the conversion flag `r`:

Python

```
>>> f"{new_comedian}"
'Eric Idle is 74.'
>>> f"{new_comedian!r}"
'Eric Idle is 74. Surprise!'
```

>>>

If you'd like to read some of the conversation that resulted in f-strings supporting full Python expressions, you can do so [here](#).

Multiline f-Strings

You can have multiline strings:

Python

```
>>> name = "Eric"
>>> profession = "comedian"
>>> affiliation = "Monty Python"
>>> message = (
...     f"Hi {name}. "
...     f"You are a {profession}. "
...     f"You were in {affiliation}."
... )
>>> message
'Hi Eric. You are a comedian. You were in Monty Python.'
```

>>>

But remember that you need to place an `f` in front of each line of a multiline string. The following code won't work:

Python

```
>>> message = (
...     f"Hi {name}. "
...     "You are a {profession}. "
```

>>>

```
...     "You were in {affiliation}."  
... )  
>>> message  
'Hi Eric. You are a {profession}. You were in {affiliation}.'
```

If you don't put an `f` in front of each individual line, then you'll just have regular, old, garden-variety strings and not shiny, new, fancy f-strings.

If you want to spread strings over multiple lines, you also have the option of escaping a return with a `\`:

```
Python >>>  
>>> message = f"Hi {name}. \" \\\n...     f"You are a {profession}. \" \\\n...     f"You were in {affiliation}."  
...  
>>> message  
'Hi Eric. You are a comedian. You were in Monty Python.'
```

But this is what will happen if you use `"""`:

```
Python >>>  
>>> message = f"""  
...     Hi {name}.  
...     You are a {profession}.  
...     You were in {affiliation}.  
... """  
...  
>>> message  
'\n    Hi Eric.\n    You are a comedian.\n    You were in Monty Python.\n'
```

Read up on indentation guidelines in [PEP 8](#).

Speed

The `f` in f-strings may as well stand for “fast.”

f-strings are faster than both %-formatting and `str.format()`. As you already saw, f-strings are expressions evaluated at runtime rather than constant values. Here's an excerpt from the docs:

“F-strings provide a way to embed expressions inside string literals, using a minimal syntax. It should be noted that an f-string is really an expression evaluated at run time, not a constant value. In Python source code, an f-string is a literal string, prefixed with `f`, which contains expressions inside braces. The expressions are replaced with their values.” ([Source](#))

At runtime, the expression inside the curly braces is evaluated in its own scope and then put together with the string literal part of the f-string. The resulting string is then returned. That's all it takes.

Here's a speed comparison:

```
Python >>>  
>>> import timeit  
>>> timeit.timeit("""name = "Eric"  
... age = 74  
... '%s is %.2f' % (name, age)""", number = 10000)  
0.003324444866599663
```

```
Python >>>  
>>> timeit.timeit("""name = "Eric"  
... age = 74  
... '%s is %.2f' % (name, age)""", number = 10000)  
0.003324444866599663
```

```
... age = 74
... '{} is {}'.format(name, age)"""
0.004242089427570761
```

Python

>>>

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... f'{name} is {age}."""
0.0024820892040722242
```

As you can see, f-strings come out on top.

However, that wasn't always the case. When they were first implemented, they had some [speed issues](#) and needed to be made faster than `str.format()`. A special [BUILD_STRING opcode](#) was introduced.

Python f-Strings: The Pesky Details

Now that you've learned all about why f-strings are great, I'm sure you want to get out there and start using them. Here are a few details to keep in mind as you venture off into this brave new world.

Quotation Marks

You can use various types of quotation marks inside the expressions. Just make sure you are not using the same type of quotation mark on the outside of the f-string as you are using in the expression.

This code will work:

```
Python
>>> f"{'Eric Idle'}"
'Eric Idle'
```

>>>

This code will also work:

```
Python
>>> f'{"Eric Idle"}'
'Eric Idle'
```

>>>

You can also use triple quotes:

```
Python
>>> f"""Eric Idle"""
'Eric Idle'
```

>>>

```
Python
>>> f'''Eric Idle'''
'Eric Idle'
```

>>>

If you find you need to use the same type of quotation mark on both the inside and the outside of the string, then you can escape with \:

```
Python
>>> f"The \"comedian\" is {name}, aged {age}."
'The "comedian" is Eric Idle, aged 74.'
```

>>>

Dictionaries

Speaking of quotation marks, watch out when you are working with dictionaries. If you are going to use single quotation marks for the keys of the dictionary, then remember to make sure you're using double quotation marks

for the f-strings containing the keys.

This will work:

```
Python >>>
>>> comedian = {'name': 'Eric Idle', 'age': 74}
>>> f"The comedian is {comedian['name']}, aged {comedian['age']}."
The comedian is Eric Idle, aged 74.
```

But this will be a hot mess with a [syntax error](#):

```
Python >>>
>>> comedian = {'name': 'Eric Idle', 'age': 74}
>>> f'The comedian is {comedian['name']}, aged {comedian['age']}.'
  File "<stdin>", line 1
    f'The comedian is {comedian['name']}, aged {comedian['age']}.'^
                                          ^
SyntaxError: invalid syntax
```

If you use the same type of quotation mark around the dictionary keys as you do on the outside of the f-string, then the quotation mark at the beginning of the first dictionary key will be interpreted as the end of the string.

Braces

In order to make a brace appear in your string, you must use double braces:

```
Python >>>
>>> f"{{70 + 4}}"
'{70 + 4}'
```

Note that using triple braces will result in there being only single braces in your string:

```
Python >>>
>>> f"{{{70 + 4}}}"
'{74}'
```

However, you can get more braces to show if you use more than triple braces:

```
Python >>>
>>> f"{{{70 + 4}}}"
'{70 + 4}'
```

Backslashes

As you saw earlier, it is possible for you to use backslash escapes in the string portion of an f-string. However, you can't use backslashes to escape in the expression part of an f-string:

```
Python >>>
>>> f"\"Eric Idle\""
  File "<stdin>", line 1
    f"\"Eric Idle\""
      ^
SyntaxError: f-string expression part cannot include a backslash
```

You can work around this by evaluating the expression beforehand and using the result in the f-string:

```
Python >>>
```

```
>>> name = "Eric Idle"
>>> f"{name}"
'Eric Idle'
```

Inline Comments

Expressions should not include comments using the # symbol. You'll get a syntax error:

```
Python >>>
>>> f"Eric is {2 * 37 #Oh my!}."
  File "<stdin>", line 1
    f"Eric is {2 * 37 #Oh my!}."
                                         ^
SyntaxError: f-string expression part cannot include '#'
```

Go Forth and Format!

You can still use the older ways of formatting strings, but with f-strings, you now have a more concise, readable, and convenient way that is both faster and less prone to error. Simplifying your life by using f-strings is a great reason to start using Python 3.6 if you haven't already made the switch. (If you are still using Python 2, don't forget that [2020](#) will be here soon!)

According to the [Zen of Python](#), when you need to decide how to do something, then “[t]here should be one– and preferably only one –obvious way to do it.” Although f-strings aren’t the only possible way for you to format strings, they are in a great position to become that one obvious way to get the job done.

Further Reading

If you’d like to read an extended discussion about string interpolation, take a look at [PEP 502](#). Also, the [PEP 536 draft](#) has some more thoughts about the future of f-strings.

[Free PDF Download: Python 3 Cheat Sheet](#)

For more fun with strings, check out the following articles:

- [Python String Formatting Best Practices](#) by Dan Bader
- [Practical Introduction to Web Scraping in Python](#) by Colin OKeefe

Happy Pythoning!

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python 3's f-Strings: An Improved String Formatting Syntax](#)

About Joanna Jablonski

Joanna is the Executive Editor of Real Python. She loves natural languages just as much as she loves programming languages!

[» More about Joanna](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Adriana](#)

[David](#)

[Dan](#)

[Jim](#)

[Pavel](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [Python 3's f-Strings: An Improved String Formatting Syntax](#)



Python String Formatting Best Practices

by [Dan Bader](#) 19 Comments [basics](#) [best-practices](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [#1 “Old Style” String Formatting \(% Operator\)](#)
- [#2 “New Style” String Formatting \(.str.format\)](#)
- [#3 String Interpolation / f-Strings \(Python 3.6+\)](#)
- [#4 Template Strings \(Standard Library\)](#)
- [Which String Formatting Method Should You Use?](#)
- [Key Takeaways](#)



Enhance Python with Redis

[Explore Redis Labs](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python String Formatting Tips & Best Practices](#)

Remember the [Zen of Python](#) and how there should be “one obvious way to do something in Python”? You might scratch your head when you find out that there are *four* major ways to do string formatting in Python.

In this tutorial, you’ll learn the four main approaches to string formatting in Python, as well as their strengths and weaknesses. You’ll also get a simple rule of thumb for how to pick the best general purpose string formatting approach in your own programs.

Let’s jump right in, as we’ve got a lot to cover. In order to have a simple toy example for experimentation, let’s assume you’ve got the following variables (or constants, really) to work with:

Python

>>>

```
>>> errno = 50159747054
>>> name = 'Bob'
```

Based on these variables, you’d like to generate an output string containing a simple error message:

Python

>>>

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

That error could really spoil a dev's Monday morning... But we're here to discuss string formatting. So let's get to work.

[Remove ads](#)

#1 “Old Style” String Formatting (% Operator)

Strings in Python have a unique built-in operation that can be accessed with the `%` operator. This lets you do simple positional formatting very easily. If you've ever worked with a `printf`-style function in C, you'll recognize how this works instantly. Here's a simple example:

Python

```
>>> 'Hello, %s' % name
"Hello, Bob"
```

>>>

I'm using the `%s` format specifier here to tell Python where to substitute the value of `name`, represented as a string.

There are other format specifiers available that let you control the output format. For example, it's possible to convert numbers to hexadecimal notation or add whitespace padding to generate nicely formatted tables and reports. (See [Python Docs: “printf-style String Formatting”](#).)

Here, you can use the `%x` format specifier to convert an `int` value to a string and to represent it as a hexadecimal number:

Python

```
>>> '%x' % errno
'badc0ffee'
```

>>>

The “old style” string formatting syntax changes slightly if you want to make multiple substitutions in a single string. Because the `%` operator takes only one argument, you need to wrap the right-hand side in a tuple, like so:

Python

```
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

>>>

It's also possible to refer to variable substitutions by name in your format string, if you pass a mapping to the `%` operator:

Python

```
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
...     "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0ffee error!'
```

>>>

This makes your format strings easier to maintain and easier to modify in the future. You don't have to worry about making sure the order you're passing in the values matches up with the order in which the values are referenced in the format string. Of course, the downside is that this technique requires a little more typing.

I'm sure you've been wondering why this printf-style formatting is called "old style" string formatting. It was technically superseded by "new style" formatting in Python 3, which we're going to talk about next.

#2 "New Style" String Formatting (`str.format()`)

Python 3 introduced a [new way to do string formatting](#) that was also later back-ported to Python 2.7. This "new style" string formatting gets rid of the %-operator special syntax and makes the syntax for string formatting more regular. Formatting is now handled by [calling `.format\(\)` on a string object](#).

You can use `format()` to do simple positional formatting, just like you could with "old style" formatting:

```
Python >>>
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

Or, you can refer to your variable substitutions by name and use them in any order you want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to `format()`:

```
Python >>>
>>> 'Hey {}, there is a 0x{:x} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

This also shows that the syntax to format an `int` variable as a hexadecimal string has changed. Now you need to pass a format spec by adding a `:x` suffix. The format string syntax has become more powerful without complicating the simpler use cases. It pays off to read up on this [string formatting mini-language in the Python documentation](#).

In Python 3, this "new style" string formatting is to be preferred over %-style formatting. While ["old style" formatting has been de-emphasized](#), it has not been deprecated. It is still supported in the latest versions of Python. According to [this discussion on the Python dev email list](#) and [this issue on the Python dev bug tracker](#), %-formatting is going to stick around for a long time to come.

Still, the official Python 3 documentation doesn't exactly recommend "old style" formatting or speak too fondly of it:

"The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the `str.format()` interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text." ([Source](#))

This is why I'd personally try to stick with `str.format` for new code moving forward. Starting with Python 3.6, there's yet another way to format your strings. I'll tell you all about it in the next section.

[Remove ads](#)

#3 String Interpolation / f-Strings (Python 3.6+)

[Python 3.6 added a new string formatting approach](#) called formatted string literals or "[f-strings](#)". This new way of formatting strings lets you use embedded Python expressions inside string constants. Here's a simple example to give you a feel for the feature:

```
Python >>>
>>> f'Hello, {name}!'
'Hello, Bob!'
```

As you can see, this prefixes the string constant with the letter "f"—hence the name "f-strings." This new formatting

syntax is powerful. Because you can embed arbitrary Python expressions, you can even do inline arithmetic with it. Check out this example:

```
Python >>>
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
'Five plus ten is 15 and not 30.'
```

Formatted string literals are a Python parser feature that converts f-strings into a series of string constants and expressions. They then get joined up to build the final string.

Imagine you had the following `greet()` function that contains an f-string:

```
Python >>>
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...
>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

When you disassemble the function and inspect what's going on behind the scenes, you'll see that the f-string in the function gets transformed into something similar to the following:

```
Python >>>
>>> def greet(name, question):
...     return "Hello, " + name + "! How's it " + question + "?"
```

The real implementation is slightly faster than that because it uses the [BUILD_STRING opcode as an optimization](#). But functionally they're the same:

```
Python >>>
>>> import dis
>>> dis.dis(greet)
 2      0 LOAD_CONST          1 ('Hello, ')
 2      2 LOAD_FAST             0 (name)
 4      4 FORMAT_VALUE          0
 6      6 LOAD_CONST          2 ("! How's it ")
 8      8 LOAD_FAST             1 (question)
10     10 FORMAT_VALUE          0
12     12 LOAD_CONST          3 ('?')
14     14 BUILD_STRING          5
16     16 RETURN_VALUE
```

String literals also support the existing format string syntax of the `str.format()` method. That allows you to solve the same formatting problems we've discussed in the previous two sections:

```
Python >>>
>>> f"Hey {name}, there's a {errno:#x} error!"
"Hey Bob, there's a 0xbadc0ffee error!"
```

Python's new formatted string literals are similar to JavaScript's [Template Literals added in ES2015](#). I think they're quite a nice addition to Python, and I've already started using them in my day to day (Python 3) work. You can learn more about formatted string literals in our [in-depth Python f-strings tutorial](#).

#4 Template Strings (Standard Library)

Here's one more tool for string formatting in Python: template strings. It's a simpler and less powerful mechanism, but in some cases this might be exactly what you're looking for.

Let's take a look at a simple greeting example:

```
Python >>>
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

You see here that we need to import the `Template` class from Python's built-in `string` module. Template strings are not a core language feature but they're supplied by the [string module in the standard library](#).

Another difference is that template strings don't allow format specifiers. So in order to get the previous error string example to work, you'll need to manually transform the `int` error number into a hex-string:

```
Python >>>
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

That worked great.

So when should you use template strings in your Python programs? In my opinion, the best time to use template strings is when you're handling formatted strings generated by users of your program. Due to their reduced complexity, template strings are a safer choice.

The more complex formatting mini-languages of the other string formatting techniques might introduce security vulnerabilities to your programs. For example, it's [possible for format strings to access arbitrary variables in your program](#).

That means, if a malicious user can supply a format string, they can potentially leak secret keys and other sensitive information! Here's a simple proof of concept of how this attack might be used against your code:

```
Python >>>
>>> # This is our super secret key:
>>> SECRET = 'this-is-a-secret'

>>> class Error:
...     def __init__(self):
...         pass

>>> # A malicious user can craft a format string that
>>> # can read data from the global namespace:
>>> user_input = '{error.__init__.globals__[SECRET]}'

>>> # This allows them to exfiltrate sensitive information,
>>> # like the secret key:
>>> err = Error()
>>> user_input.format(error=err)
'this-is-a-secret'
```

See how a hypothetical attacker was able to extract our secret string by accessing the `__globals__` dictionary from a

See how a hypothetical attacker was able to extract our secret string by accessing the `__globals__` dictionary from a malicious format string? Scary, huh? Template strings close this attack vector. This makes them a safer choice if you're handling format strings generated from user input:

Python

>>>

```
>>> user_input = '${error.__init__.globals__[SECRET]}'  
>>> Template(user_input).substitute(error=err)  
ValueError:  
"Invalid placeholder in string: line 1, col 1"
```

 [Remove ads](#)

Which String Formatting Method Should You Use?

I totally get that having so much choice for how to format your strings in Python can feel very confusing. This is an excellent cue to bust out this handy flowchart infographic I've put together for you:

Python String Formatting Rule of Thumb (Image: [Click to Tweet](#))

This flowchart is based on the rule of thumb that I apply when I'm writing Python:

Python String Formatting Rule of Thumb: If your format strings are user-supplied, use [Template Strings \(#4\)](#) to avoid security issues. Otherwise, use [Literal String Interpolation/f-Strings \(#3\)](#) if you're on Python 3.6+, and ["New Style" str.format \(#2\)](#) if you're not.

Key Takeaways

- Perhaps surprisingly, there's more than one way to handle string formatting in Python.
- Each method has its individual pros and cons. Your use case will influence which method you should use.
- If you're having trouble deciding which string formatting method to use, try our *Python String Formatting Rule of Thumb*.

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python String Formatting Tips & Best Practices](#)

About Dan Bader

Dan Bader is the owner and editor in chief of Real Python and the main developer of the realpython.com learning platform. Dan has been writing code for more than 20 years and holds master's degree in computer science.

[» More about Dan](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [best-practices](#) [python](#)

Recommended Video Course: [Python String Formatting Tips & Best Practices](#)



Python Exceptions: An Introduction

by Said van de Klundert 23 Comments basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Exceptions versus Syntax Errors](#)
- [Raising an Exception](#)
- [The AssertionError Exception](#)
- [The try and except Block: Handling Exceptions](#)
- [The else Clause](#)
- [Cleaning Up After Using finally](#)
- [Summing Up](#)



Your Guided Tour Through the Python 3.9 Interpreter »

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Introduction to Python Exceptions](#)

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. In this article, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.



Exceptions versus Syntax Errors

[Syntax errors](#) occur when the parser detects an incorrect statement. Observe the following example:

Python

```
>>> print( 0 / 0 )
      File "<stdin>", line 1
        print( 0 / 0 )
                  ^
SyntaxError: invalid syntax
```

The arrow indicates where the parser ran into the **syntax error**. In this example, there was one bracket too many. Remove it and run your code again:

Python

```
>>> print( 0 / 0 )
      Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This time, you ran into an **exception error**. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message `exception error`, Python details what type of exception error was encountered. In this case, it was a `ZeroDivisionError`. Python comes with [various built-in exceptions](#) as well as the possibility to create self-defined exceptions.

Raising an Exception

We can use `raise` to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

Use `raise` to force an exception:



If you want to throw an error when a certain condition occurs using `raise`, you could go about it like this:

Python

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

When you run this code, the output will be the following:

Python

```
Traceback (most recent call last):
File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

The `AssertionError` Exception

Instead of waiting for a program to crash midway, you can also start by [making an assertion in Python](#). We assert that a certain condition is met. If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an `AssertionError` exception.

Assert that a condition is met:

`assert:`



Test if condition is True

Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

Python

```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

Python

```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: This code runs on Linux only.
```

In this example, throwing an `AssertionError` exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

The try and except Block: Handling Exceptions

The `try` and `except` block in Python is used to catch and handle exceptions. Python executes code following the `try` statement as a “normal” part of the program. The code that follows the `except` statement is the program’s response to any exceptions in the preceding `try` clause.

As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The `except` clause determines how your program responds to exceptions.

The following function can help you understand the `try` and `except` block:

Python

```
def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')
```

The `linux_interaction()` can only run on a Linux system. The `assert` in this function will throw an `AssertionError` exception if you call it on an operating system other than Linux.

You can give the function a try using the following code:

Python

```
try:
    linux_interaction()
except:
    pass
```

The way you handled the error here is by handing out a `pass`. If you were to run this code on a Windows machine, you would get the following output:

Shell

You got nothing. The good thing here is that the program did not crash. But it would be nice to see if some type of exception occurred whenever you ran your code. To this end, you can change the `pass` into something that would generate an informative message, like so:

Python

```
try:
    linux_interaction()
except:
    print('Linux function was not executed')
```

Execute this code on a Windows machine:

Shell

```
Linux function was not executed
```

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the `AssertionError` and output that message to screen:

Python

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
    print('The linux_interaction() function was not executed')
```

Running this function on a Windows machine outputs the following:

Shell

```
Function can only run on Linux systems.
The linux_interaction() function was not executed
```

The first message is the `AssertionError`, informing you that the function can only be executed on a Linux machine. The second message tells you which function was not executed.

In the previous example, you called a function that you wrote yourself. When you executed the function, you caught the `AssertionError` exception and printed it to screen.

Here's another example where you open a file and use a built-in exception:

Python

```
try:  
    with open('file.log') as file:  
        read_data = file.read()  
except:  
    print('Could not open file.log')
```

If `file.log` does not exist, this block of code will output the following:

Shell

```
Could not open file.log
```

This is an informative message, and our program will still continue to run. In the [Python docs](#), you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

Exception `FileNotFoundException`

Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT.

To catch this type of exception and print it to screen, you could use the following code:

Python

```
try:  
    with open('file.log') as file:  
        read_data = file.read()  
except FileNotFoundError as fnf_error:  
    print(fnf_error)
```

In this case, if `file.log` does not exist, the output will be the following:

Shell

```
[Errno 2] No such file or directory: 'file.log'
```

You can have more than one function call in your `try` clause and anticipate catching various exceptions. A thing to note here is that the code in the `try` clause will stop as soon as an exception is encountered.

Warning: Catching `Exception` hides all errors...even those which are completely unexpected. This is why you should avoid bare `except` clauses in your Python programs. Instead, you'll want to refer to *specific exception classes* you want to catch and handle. You can learn more about why this is a good idea [in this tutorial](#).

Look at the following code. Here, you first call the `linux_interaction()` function and then try to open a file:

Python

```
try:  
    linux_interaction()  
    with open('file.log') as file:  
        read_data = file.read()  
except FileNotFoundError as fnf_error:  
    print(fnf_error)  
except AssertionError as error:  
    print(error)  
    print('Linux linux_interaction() function was not executed')
```

If the file does not exist, running this code on a Windows machine will output the following:

Shell

```
Function can only run on Linux systems.  
Linux linux_interaction() function was not executed
```

Inside the try clause, you ran into an exception immediately and did not get to the part where you attempt to open `file.log`. Now look at what happens when you run the code on a Linux machine:

Shell

```
[Errno 2] No such file or directory: 'file.log'
```

Here are the key takeaways:

- A try clause is executed up until the point where the first exception is encountered.
- Inside the except clause, or the exception handler, you determine how the program responds to the exception.
- You can anticipate multiple exceptions and differentiate how the program should respond to them.
- [Avoid using bare except clauses.](#)

The else Clause

In Python, using the `else` statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.

Look at the following example:

Python

```
try:  
    linux_interaction()  
except AssertionError as error:  
    print(error)  
else:  
    print('Executing the else clause.')
```

If you were to run this code on a Linux system, the output would be the following:

Shell

```
Doing something.  
Executing the else clause.
```

Because the program did not run into any exceptions, the `else` clause was executed.

You can also try to run code inside the `else` clause and catch possible exceptions there as well:

Python

```
try:  
    linux_interaction()  
except AssertionError as error:  
    print(error)  
else:  
    try:  
        with open('file.log') as file:  
            read_data = file.read()  
    except FileNotFoundError as fnf_error:  
        print(fnf_error)
```

If you were to execute this code on a Linux machine, you would get the following result:

Shell

```
Doing something.  
[Errno 2] No such file or directory: 'file.log'
```

From the output, you can see that the `linux_interaction()` function ran. Because no exceptions were encountered, an attempt to open `file.log` was made. That file did not exist, and instead of opening the file, you caught the `FileNotFoundException` exception.

Cleaning Up After Using `finally`

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the `finally` clause.

Have a look at the following example:

Python

```
try:  
    linux_interaction()  
except AssertionError as error:  
    print(error)  
else:  
    try:  
        with open('file.log') as file:  
            read_data = file.read()  
    except FileNotFoundError as fnf_error:  
        print(fnf_error)  
finally:  
    print('Cleaning up, irrespective of any exceptions.')
```

In the previous code, everything in the `finally` clause will be executed. It does not matter if you encounter an exception somewhere in the `try` or `else` clauses. Running the previous code on a Windows machine would output the following:

Shell

```
Function can only run on Linux systems.  
Cleaning up, irrespective of any exceptions.
```

Summing Up

After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python. In this article, you saw the following options:

- `raise` allows you to throw an exception at any time.
- `assert` enables you to verify if a certain condition is met and throw an exception if it isn't.
- In the `try` clause, all statements are executed until an exception is encountered.
- `except` is used to catch and handle the exception(s) that are encountered in the `try` clause.
- `else` lets you code sections that should run only when no exceptions are encountered in the `try` clause.
- `finally` enables you to execute sections of code that should always run, with or without any previously encountered exceptions.

[Free PDF Download: Python 3 Cheat Sheet](#)

Hopefully, this article helped you understand the basic tools that Python has to offer when dealing with exceptions.

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Introduction to Python Exceptions](#)

About Said van de Klundert

Said is a network engineer, Python enthusiast, and a guest author at Real Python.

[» More about Said](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

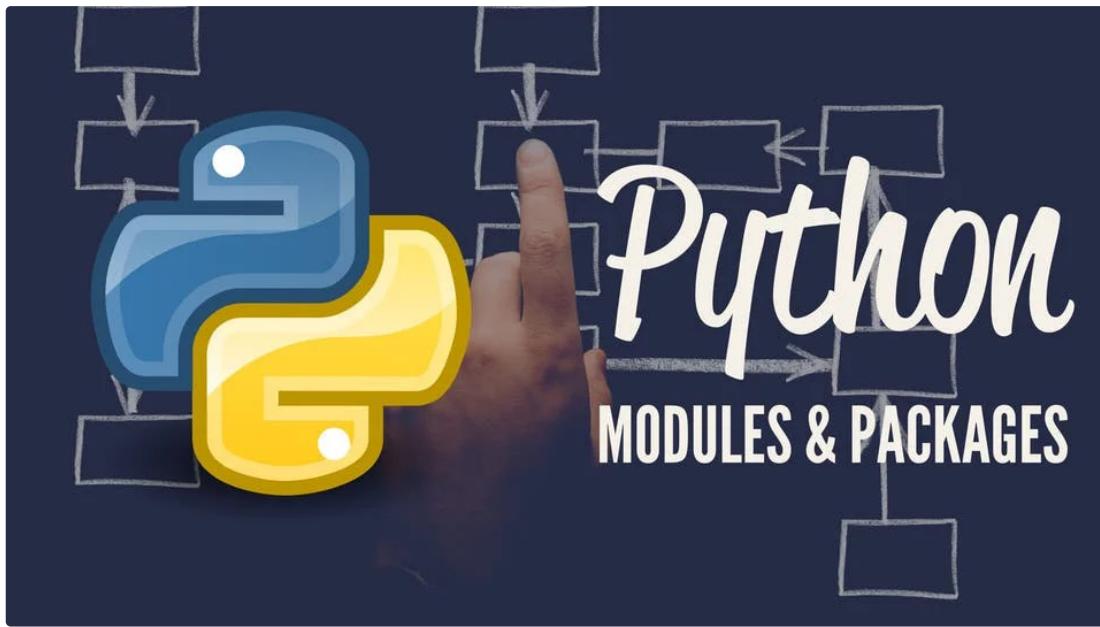
[Adriana](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [Introduction to Python Exceptions](#)



Python Modules and Packages – An Introduction

by John Sturtz 38 Comments basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Python Modules: Overview](#)
- [The Module Search Path](#)
- [The import Statement](#)
 - [import <module_name>](#)
 - [from <module_name> import <name\(s\)>](#)
 - [from <module_name> import <name> as <salt_name>](#)
 - [import <module_name> as <alt_name>](#)
- [The dir\(\) Function](#)
- [Executing a Module as a Script](#)
- [Reloading a Module](#)
- [Python Packages](#)
- [Package Initialization](#)
- [Importing * From a Package](#)
- [Subpackages](#)
- [Conclusion](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Modules and Packages: An Introduction](#)

This article explores Python **modules** and Python **packages**, two mechanisms that facilitate **modular programming**.

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application.

There are several advantages to **modularizing** code in a large application:

• **Simplicity**: Rather than focusing on the entire problem at hand, a module typically focuses on one relatively

- **Simplicity.** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.
- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to duplicate code.
- **Scoping:** Modules typically define a separate [namespace](#), which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the [Zen of Python](#) is *Namespaces are one honking great idea —let's do more of those!*)

Functions, modules and packages are all constructs in Python that promote code modularization.

[Free PDF Download: Python 3 Cheat Sheet](#)

[Remove ads](#)

Python Modules: Overview

There are actually three different ways to define a **module** in Python:

1. A module can be written in Python itself.
2. A module can be written in C and loaded dynamically at run-time, like the [re \(regular expression\)](#) module.
3. A **built-in** module is intrinsically contained in the interpreter, like the [itertools module](#).

A module's contents are accessed the same way in all three cases: with the `import` statement.

Here, the focus will mostly be on modules that are written in Python. The cool thing about modules written in Python is that they are exceedingly straightforward to build. All you need to do is create a file that contains legitimate Python code and then give the file a name with a `.py` extension. That's it! No special syntax or voodoo is necessary.

For example, suppose you have created a file called `mod.py` containing the following:

`mod.py`

Python

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')
```

```
class Foo:  
    pass
```

Several objects are defined in mod.py:

- s (a string)
- a (a list)
- foo() (a function)
- Foo (a class)

Assuming mod.py is in an appropriate location, which you will learn more about shortly, these objects can be accessed by **importing** the module as follows:

```
Python >>>  
  
>>> import mod  
>>> print(mod.s)  
If Comrade Napoleon says it, it must be right.  
>>> mod.a  
[100, 200, 300]  
>>> mod.foo(['quux', 'corge', 'grault'])  
arg = ['quux', 'corge', 'grault']  
>>> x = mod.Foo()  
>>> x  
<mod.Foo object at 0x03C181F0>
```

The Module Search Path

Continuing with the above example, let's take a look at what happens when Python executes the statement:

```
Python  
import mod
```

When the interpreter executes the above `import` statement, it searches for `mod.py` in a list of directories assembled from the following sources:

- The directory from which the input script was run or the **current directory** if the interpreter is being run interactively
- The list of directories contained in the `PYTHONPATH` environment variable, if it is set. (The format for `PYTHONPATH` is OS-dependent but should mimic the `PATH` environment variable.)
- An installation-dependent list of directories configured at the time Python is installed

The resulting search path is accessible in the Python variable `sys.path`, which is obtained from a module named `sys`:

```
Python >>>  
  
>>> import sys  
>>> sys.path  
['', 'C:\\\\Users\\\\john\\\\Documents\\\\Python\\\\doc', 'C:\\\\Python36\\\\Lib\\\\idlelib',  
'C:\\\\Python36\\\\python36.zip', 'C:\\\\Python36\\\\DLLs', 'C:\\\\Python36\\\\lib',  
'C:\\\\Python36', 'C:\\\\Python36\\\\lib\\\\site-packages']
```

Note: The exact contents of `sys.path` are installation-dependent. The above will almost certainly look slightly different on your computer.

Thus, to ensure your module is found, you need to do one of the following:

- Put `mod.py` in the directory where the input script is located or the **current directory**, if interactive
- Modify the `PYTHONPATH` environment variable to contain the directory where `mod.py` is located before starting the interpreter
 - **Or:** Put `mod.py` in one of the directories already contained in the `PYTHONPATH` variable

- Put `mod.py` in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS

There is actually one additional option: you can put the module file in any directory of your choice and then modify `sys.path` at run-time so that it contains that directory. For example, in this case, you could put `mod.py` in directory `C:\Users\john` and then issue the following statements:

Python

>>>

```
>>> sys.path.append(r'C:\Users\john')
>>> sys.path
['', 'C:\Users\john\Documents\Python\doc', 'C:\Python36\Lib\idlelib',
'C:\Python36\python36.zip', 'C:\Python36\DLLs', 'C:\Python36\lib',
'C:\Python36', 'C:\Python36\lib\site-packages', 'C:\Users\john']
>>> import mod
```

Once a module has been imported, you can determine the location where it was found with the module's `__file__` attribute:

Python

>>>

```
>>> import mod
>>> mod.__file__
'C:\Users\john\mod.py'

>>> import re
>>> re.__file__
'C:\Python36\lib\re.py'
```

The directory portion of `__file__` should be one of the directories in `sys.path`.

[Remove ads](#)

The import Statement

Module contents are made available to the caller with the `import` statement. The `import` statement takes many different forms, shown below.

`import <module_name>`

The simplest form is the one already shown above:

Python

```
import <module_name>
```

Note that this *does not* make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.

The statement `import <module_name>` only places `<module_name>` in the caller's symbol table. The *objects* that are defined in the module *remain in the module's private symbol table*.

From the caller, objects in the module are only accessible when prefixed with `<module_name>` via **dot notation**, as illustrated below.

After the following `import` statement, `mod` is placed into the local symbol table. Thus, `mod` has meaning in the caller's local context:

```
Python >>>
>>> import mod
>>> mod
<module 'mod' from 'C:\\\\Users\\\\john\\\\Documents\\\\Python\\\\doc\\\\mod.py'>
```

But `s` and `foo` remain in the module's private symbol table and are not meaningful in the local context:

```
Python >>>
>>> s
NameError: name 's' is not defined
>>> foo('quux')
NameError: name 'foo' is not defined
```

To be accessed in the local context, names of objects defined in the module must be prefixed by `mod`:

```
Python >>>
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo('quux')
arg = quux
```

Several comma-separated modules may be specified in a single `import` statement:

```
Python
import <module_name>[, <module_name> ...]
```

`from <module_name> import <name(s)>`

An alternate form of the `import` statement allows individual objects from the module to be imported *directly into the caller's symbol table*:

```
Python
from <module_name> import <name(s)>
```

Following execution of the above statement, `<name(s)>` can be referenced in the caller's environment without the `<module_name>` prefix:

```
Python >>>
>>> from mod import s, foo
>>> s
'If Comrade Napoleon says it, it must be right.'
>>> foo('quux')
arg = quux

>>> from mod import Foo
>>> x = Foo()
>>> x
<mod.Foo object at 0x02E3AD50>
```

Because this form of `import` places the object names directly into the caller's symbol table, any objects that already exist with the same name will be *overwritten*:

```
Python >>>
>>> a = ['foo', 'bar', 'baz']
>>> a
```

```
>>> a
['foo', 'bar', 'baz']

>>> from mod import a
>>> a
[100, 200, 300]
```

It is even possible to indiscriminately `import` everything from a module at one fell swoop:

Python

```
from <module_name> import *
```

This will place the names of *all* objects from `<module_name>` into the local symbol table, with the exception of any that begin with the underscore (`_`) character.

For example:

Python

```
>>> from mod import *
>>> s
'If Comrade Napoleon says it, it must be right.'
>>> a
[100, 200, 300]
>>> foo
<function foo at 0x03B449C0>
>>> Foo
<class 'mod.Foo'>
```

>>>

This isn't necessarily recommended in large-scale production code. It's a bit dangerous because you are entering names into the local symbol table *en masse*. Unless you know them all well and can be confident there won't be a conflict, you have a decent chance of overwriting an existing name inadvertently. However, this syntax is quite handy when you are just mucking around with the interactive interpreter, for testing or discovery purposes, because it quickly gives you access to everything a module has to offer without a lot of typing.

[Remove ads](#)

```
from <module_name> import <name> as <alt_name>
```

It is also possible to `import` individual objects but enter them into the local symbol table with alternate names:

Python

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> ...]
```

This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names:

Python

```
>>> s = 'foo'
>>> a = ['foo', 'bar', 'baz']

>>> from mod import s as string, a as alist
>>> s
'foo'
>>> string
'If Comrade Napoleon says it, it must be right.'
>>> a
['foo', 'bar', 'baz']
>>> alist
[100, 200, 300]
```

>>>

```
import <module_name> as <alt_name>
```

You can also import an entire module under an alternate name:

Python

```
import <module_name> as <alt_name>
```

Python

>>>

```
>>> import mod as my_module
>>> my_module.a
[100, 200, 300]
>>> my_module.foo('qux')
arg = qux
```

Module contents can be imported from within a [function definition](#). In that case, the `import` does not occur until the function is *called*:

Python

>>>

```
>>> def bar():
...     from mod import foo
...     foo('corge')
...
>>> bar()
arg = corge
```

However, **Python 3** does not allow the indiscriminate `import *` syntax from within a function:

Python

>>>

```
>>> def bar():
...     from mod import *
...
SyntaxError: import * only allowed at module level
```

Lastly, a [try statement with an except ImportError clause](#) can be used to guard against unsuccessful `import` attempts:

Python

>>>

```
>>> try:
...     # Non-existent module
...     import baz
... except ImportError:
...     print('Module not found')
...
Module not found
```

Python

>>>

```
>>> try:
...     # Existing module, but non-existent object
...     from mod import baz
... except ImportError:
...     print('Object not found in module')
...
Object not found in module
```

The `dir()` Function

The built-in function `dir()` returns a list of defined names in a namespace. Without arguments, it produces an

empty list. With one argument, it lists the names defined in that object's namespace.

alphabetically sorted list of names in the current **local symbol table**:

Python

>>>

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__']

>>> qux = [1, 2, 3, 4, 5]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'qux']

>>> class Bar():
...     pass
...
>>> x = Bar()
>>> dir()
['Bar', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'qux', 'x']
```

Note how the first call to `dir()` above lists several names that are automatically defined and already in the namespace when the interpreter starts. As new names are defined (`qux`, `Bar`, `x`), they appear on subsequent invocations of `dir()`.

This can be useful for identifying what exactly has been added to the namespace by an import statement:

Python

>>>

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__']

>>> import mod
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'mod']
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo([1, 2, 3])
arg = [1, 2, 3]

>>> from mod import a, Foo
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'a', 'mod']
>>> a
[100, 200, 300]
>>> x = Foo()
>>> x
<mod.Foo object at 0x002EAD50>

>>> from mod import s as string
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'a', 'mod', 'string', 'x']
>>> string
'If Comrade Napoleon says it, it must be right.'
```

When given an argument that is the name of a module, `dir()` lists the names defined in the module.

which given an argument that is the name of a module, dir() lists the names defined in the module.

Python

>>>

```
>>> import mod
>>> dir(mod)
['__Foo__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'a', 'foo', 's']
```

Python

>>>

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__']
>>> from mod import *
>>> dir()
['__Foo__', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'a', 'foo', 's']
```

[Remove ads](#)

Executing a Module as a Script

Any .py file that contains a **module** is essentially also a Python **script**, and there isn't any reason it can't be executed like one.

Here again is mod.py as it was defined above:

mod.py

Python

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

This can be run as a script:

Windows Console

```
C:\Users\john\Documents>python mod.py
C:\Users\john\Documents>
```

There are no errors, so it apparently worked. Granted, it's not very interesting. As it is written, it only *defines* objects. It doesn't *do* anything with them, and it doesn't generate any output.

Let's modify the above Python module so it does generate some output when run as a script:

mod.py

Python

```
s = "If Comrade Napoleon says it, it must be right."
```

```

a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

print(s)
print(a)
foo('quux')
x = Foo()
print(x)

```

Now it should be a little more interesting:

Windows Console

```

C:\Users\john\Documents>python mod.py
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x02F101D0>

```

Unfortunately, now it also generates output when imported as a module:

Python

>>>

```

>>> import mod
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<mod.Foo object at 0x0169AD50>

```

This is probably not what you want. It isn't usual for a module to generate output when it is imported.

Wouldn't it be nice if you could distinguish between when the file is loaded as a module and when it is run as a standalone script?

Ask and ye shall receive.

When a .py file is imported as a module, Python sets the special **dunder** variable `__name__` to the name of the module. However, if a file is run as a standalone script, `__name__` is (creatively) set to the string '`__main__`'. Using this fact, you can discern which is the case at run-time and alter behavior accordingly:

mod.py

Python

```

s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(s)
    print(a)
    foo('quux')
    x = Foo()
    print(x)

```

Now, if you run as a script, you get output:

Windows Console

```
C:\Users\john\Documents>python mod.py
Executing as standalone script
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x03450690>
```

But if you import as a module, you don't:

Python

>>>

```
>>> import mod
>>> mod.foo('grault')
arg = grault
```

Modules are often designed with the capability to run as a standalone script for purposes of testing the functionality that is contained within the module. This is referred to as **unit testing**. For example, suppose you have created a module `fact.py` containing a **factorial** function, as follows:

fact.py

Python

```
def fact(n):
    return 1 if n == 1 else n * fact(n-1)

if (__name__ == '__main__'):
    import sys
    if len(sys.argv) > 1:
        print(fact(int(sys.argv[1])))
```

The file can be treated as a module, and the `fact()` function imported:

Python

>>>

```
>>> from fact import fact
>>> fact(6)
720
```

But it can also be run as a standalone by passing an integer argument on the command-line for testing:

Windows Console

```
C:\Users\john\Documents>python fact.py 6
720
```

[Remove ads](#)

Reloading a Module

For reasons of efficiency, a module is only loaded once per interpreter session. That is fine for function and class definitions, which typically make up the bulk of a module's contents. But a module can contain executable

statements as well, usually for initialization. Be aware that these statements will only be executed the *first time* a module is imported.

Consider the following file `mod.py`:

`mod.py`

Python

```
a = [100, 200, 300]
print('a =', a)
```

Python

>>>

```
>>> import mod
a = [100, 200, 300]
>>> import mod
>>> import mod

>>> mod.a
[100, 200, 300]
```

The `print()` statement is not executed on subsequent imports. (For that matter, neither is the assignment statement, but as the final display of the value of `mod.a` shows, that doesn't matter. Once the assignment is made, it sticks.)

If you make a change to a module and need to reload it, you need to either restart the interpreter or use a function called `reload()` from module `importlib`:

Python

>>>

```
>>> import mod
a = [100, 200, 300]

>>> import mod

>>> import importlib
>>> importlib.reload(mod)
a = [100, 200, 300]
<module 'mod' from 'C:\\\\Users\\\\john\\\\Documents\\\\Python\\\\doc\\\\mod.py'>
```

Python Packages

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Creating a **package** is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Consider the following arrangement:

Here, there is a directory named `pkg` that contains two modules, `mod1.py` and `mod2.py`. The contents of the modules are:

`mod1.py`

```
Python
```

```
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

mod2.py

```
Python
```

```
def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

Given this structure, if the `pkg` directory resides in a location where it can be found (in one of the directories contained in `sys.path`), you can refer to the two **modules** with **dot notation** (`pkg.mod1`, `pkg.mod2`) and import them with the syntax you are already familiar with:

```
Python
```

```
import <module_name>[, <module_name> ...]
```

```
Python
```

>>>

```
>>> import pkg.mod1, pkg.mod2
>>> pkg.mod1.foo()
[mod1] foo()
>>> x = pkg.mod2.Bar()
>>> x
<pkg.mod2.Bar object at 0x033F7290>
```

```
Python
```

```
from <module_name> import <name(s)>
```

```
Python
```

>>>

```
>>> from pkg.mod1 import foo
>>> foo()
[mod1] foo()
```

```
Python
```

```
from <module_name> import <name> as <alt_name>
```

```
Python
```

>>>

```
>>> from pkg.mod2 import Bar as Qux
>>> x = Qux()
>>> x
<pkg.mod2.Bar object at 0x036DFFD0>
```

You can import modules with these statements as well:

```
Python
```

```
from <package_name> import <modules_name>[, <module_name> ...]
from <package_name> import <module_name> as <alt_name>
```

```
Python
```

>>>

```
>>> from pkg import mod1
>>> mod1.foo()
[mod1] foo()
```

```
>>> from pkg import mod2 as quux
>>> quux.bar()
[mod2] bar()
```

You can technically import the package as well:

```
Python >>>
>>> import pkg
>>> pkg
<module 'pkg' (namespace)>
```

But this is of little avail. Though this is, strictly speaking, a syntactically correct Python statement, it doesn't do much of anything useful. In particular, it *does not place* any of the modules in `pkg` into the local namespace:

```
Python >>>
>>> pkg.mod1
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    pkg.mod1
AttributeError: module 'pkg' has no attribute 'mod1'
>>> pkg.mod1.foo()
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    pkg.mod1.foo()
AttributeError: module 'pkg' has no attribute 'mod1'
>>> pkg.mod2.Bar()
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    pkg.mod2.Bar()
AttributeError: module 'pkg' has no attribute 'mod2'
```

To actually import the modules or their contents, you need to use one of the forms shown above.

[Remove ads](#)

Package Initialization

If a file named `__init__.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

For example, consider the following `__init__.py` file:

`__init__.py`

```
Python
print(f'Invoking __init__.py for {__name__}')
A = ['quux', 'corge', 'grault']
```

Let's add this file to the `pkg` directory from the above example:

Now when the package is imported, the global list A is initialized:

Python

```
>>> import pkg
Invoking __init__.py for pkg
>>> pkg.A
['quux', 'corge', 'grault']
```

>>>

A **module** in the package can access the global variable by importing it in turn:

mod1.py

Python

```
def foo():
    from pkg import A
    print('[mod1] foo() / A = ', A)

class Foo:
    pass
```

Python

```
>>> from pkg import mod1
Invoking __init__.py for pkg
>>> mod1.foo()
[mod1] foo() / A =  ['quux', 'corge', 'grault']
```

>>>

`__init__.py` can also be used to effect automatic importing of modules from a package. For example, earlier you saw that the statement `import pkg` only places the name `pkg` in the caller's local symbol table and doesn't import any modules. But if `__init__.py` in the `pkg` directory contains the following:

`__init__.py`

Python

```
print(f'Invoking __init__.py for {__name__}')
import pkg.mod1, pkg.mod2
```

then when you execute `import pkg`, modules `mod1` and `mod2` are imported automatically:

Python

```
>>> import pkg
Invoking __init__.py for pkg
>>> pkg.mod1.foo()
[mod1] foo()
>>> pkg.mod2.bar()
[mod2] bar()
```

>>>

Note: Much of the Python documentation states that an `__init__.py` file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of `__init__.py` signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it **had** to be present.

Starting with **Python 3.3**, [Implicit Namespace Packages](#) were introduced. These allow for the creation of a package without any `__init__.py` file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.

Importing * From a Package

For the purposes of the following discussion, the previously defined package is expanded to contain some additional modules:

There are now four modules defined in the `pkg` directory. Their contents are as shown below:

mod1.py

Python

```
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

mod2.py

Python

```
def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

mod3.py

Python

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass
```

mod4.py

Python

```
def qux():
    print('[mod4] qux()')

class Qux:
    pass
```

(Imaginative, aren't they?)

You have already seen that when `import *` is used for a **module**, *all* objects from the module are imported into the local symbol table, except those whose names begin with an underscore, as always:

```

Python >>>
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg.mod3 import *

>>> dir()
['Baz', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'baz']
>>> baz()
[mod3] baz()
>>> Baz
<class 'pkg.mod3.Baz'>

```

The analogous statement for a **package** is this:

```

Python
from <package_name> import *

```

What does that do?

```

Python >>>
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

```

Hmph. Not much. You might have expected (assuming you had any expectations at all) that Python would dive down into the package directory, find all the modules it could, and import them all. But as you can see, by default that is not what happens.

Instead, Python follows this convention: if the `__init__.py` file in the **package** directory contains a **list** named `__all__`, it is taken to be a list of modules that should be imported when the statement `from <package_name> import *` is encountered.

For the present example, suppose you create an `__init__.py` in the `pkg` directory like this:

`pkg/__init__.py`

```

Python
__all__ = [
    'mod1',
    'mod2',
    'mod3',
    'mod4'
]

```

Now `from pkg import *` imports all four modules:

```

Python >>>

```

```

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'mod1', 'mod2', 'mod3', 'mod4']
>>> mod2.bar()
[mod2] bar()
>>> mod4.Qux
<class 'pkg.mod4.Qux'>

```

Using `import *` still isn't considered terrific form, any more for **packages** than for **modules**. But this facility at least gives the creator of the package some control over what happens when `import *` is specified. (In fact, it provides the capability to disallow it entirely, simply by declining to define `__all__` at all. As you have seen, the default behavior for packages is to import nothing.)

By the way, `__all__` can be defined in a **module** as well and serves the same purpose: to control what is imported with `import *`. For example, modify `mod1.py` as follows:

pkg/mod1.py

Python

```

__all__ = ['foo']

def foo():
    print('[mod1] foo()')

class Foo:
    pass

```

Now an `import *` statement from `pkg.mod1` will only import what is contained in `__all__`:

Python

>>>

```

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg.mod1 import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'foo']

>>> foo()
[mod1] foo()
>>> Foo
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    Foo
NameError: name 'Foo' is not defined

```

`foo()` (the function) is now defined in the local namespace, but `Foo` (the class) is not, because the latter is not in `__all__`.

In summary, `__all__` is used by both **packages** and **modules** to control what is imported when `import *` is specified. But *the default behavior differs*:

- For a package, when `__all__` is not defined, `import *` does not import anything.
- For a module, when `__all__` is not defined, `import *` imports everything (except—you guessed it—names starting with an underscore).

[Remove ads](#)

Subpackages

Packages can contain nested **subpackages** to arbitrary depth. For example, let's make one more modification to the example **package** directory as follows:

The four modules (`mod1.py`, `mod2.py`, `mod3.py` and `mod4.py`) are defined as previously. But now, instead of being lumped together into the `pkg` directory, they are split out into two **subpackage** directories, `sub_pkg1` and `sub_pkg2`.

Importing still works the same as shown previously. Syntax is similar, but additional **dot notation** is used to separate **package** name from **subpackage** name:

```
Python >>>
>>> import pkg.sub_pkg1.mod1
>>> pkg.sub_pkg1.mod1.foo()
[mod1] foo()

>>> from pkg.sub_pkg1 import mod2
>>> mod2.bar()
[mod2] bar()

>>> from pkg.sub_pkg2.mod3 import baz
>>> baz()
[mod3] baz()

>>> from pkg.sub_pkg2.mod4 import qux as grault
>>> grault()
[mod4] qux()
```

In addition, a module in one **subpackage** can reference objects in a **sibling subpackage** (in the event that the sibling contains some functionality that you need). For example, suppose you want to import and execute function `foo()` (defined in module `mod1`) from within module `mod3`. You can either use an **absolute import**:

pkg/sub_pkg2/mod3.py

```
Python
def baz():
    print('[mod3] baz()')

class Baz:
    pass

from pkg.sub_pkg1.mod1 import foo
foo()
```

Python

>>>

```
>>> from pkg.sub_pkg2 import mod3
[mod1] foo()
>>> mod3.foo()
[mod1] foo()
```

Or you can use a **relative import**, where .. refers to the package one level up. From within `mod3.py`, which is in subpackage `sub_pkg2`,

- .. evaluates to the parent package (`pkg`), and
- ..`sub_pkg1` evaluates to subpackage `sub_pkg1` of the parent package.

`pkg/sub_pkg2/mod3.py`

Python

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass

from .. import sub_pkg1
print(sub_pkg1)

from ..sub_pkg1.mod1 import foo
foo()
```

Python

>>>

```
>>> from pkg.sub_pkg2 import mod3
<module 'pkg.sub_pkg1' (namespace)>
[mod1] foo()
```

Conclusion

In this tutorial, you covered the following topics:

- How to create a Python **module**
- Locations where the Python interpreter searches for a module
- How to obtain access to the objects defined in a module with the `import` statement
- How to create a module that is executable as a standalone script
- How to organize modules into **packages** and **subpackages**
- How to control package initialization

[Free PDF Download: Python 3 Cheat Sheet](#)

This will hopefully allow you to better understand how to gain access to the functionality available in the many third-party and built-in modules available in Python.

Additionally, if you are developing your own application, creating your own **modules** and **packages** will help you organize and modularize your code, which makes coding, maintenance, and debugging easier.

If you want to learn more, check out the following documentation at [Python.org](#):

- [The import system](#)
- [The Python tutorial: Modules](#)

Happy Pythoning!



This tutorial has a related video course created by the Real Python team. Watch it together with

About John Sturtz

John is an avid Pythonista and a member of the Real Python tutorial team.

[» More about John](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

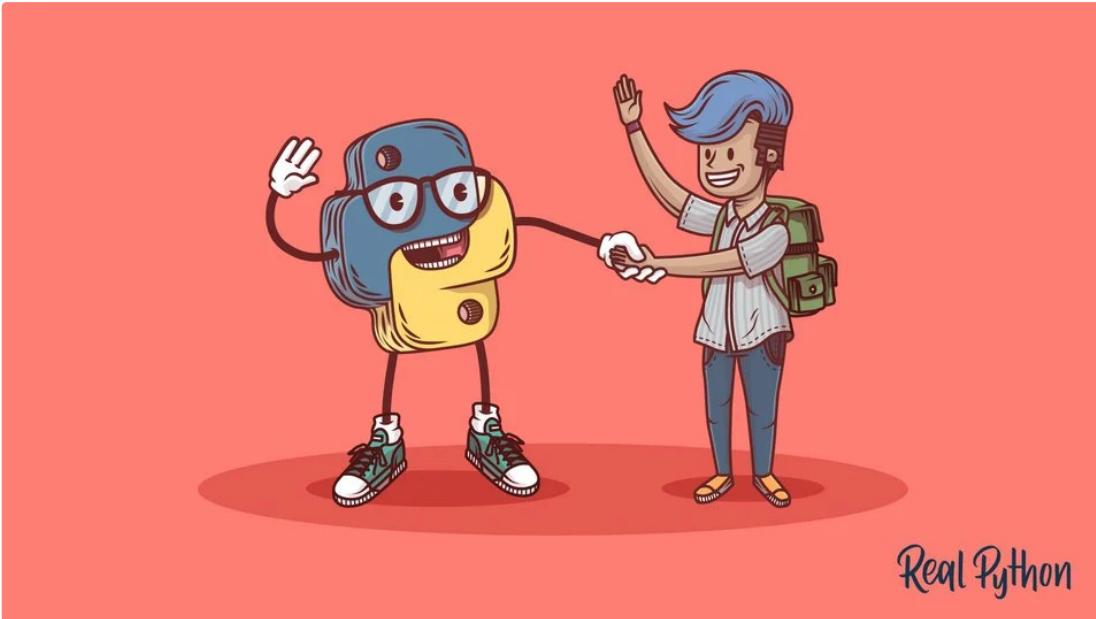
[Dan](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [Python Modules and Packages: An Introduction](#)



Namespaces and Scope in Python

by [John Sturtz](#) ⌚ Jul 29, 2020 💬 1 Comment 🏷️ basics 🏷️ python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Namespaces in Python](#)
 - [The Built-In Namespace](#)
 - [The Global Namespace](#)
 - [The Local and Enclosing Namespaces](#)
- [Variable Scope](#)
- [Python Namespace Dictionaries](#)
 - [The `globals\(\)` function](#)
 - [The `locals\(\)` function](#)
- [Modify Variables Out of Scope](#)
 - [The `global` Declaration](#)
 - [The `nonlocal` Declaration](#)
 - [Best Practices](#)
- [Conclusion](#)



Your Guided Tour Through the Python 3.9 Interpreter »

This tutorial covers Python **namespaces**, the structures used to organize the symbolic names assigned to objects in a Python program.

The previous tutorials in this series have emphasized the importance of **objects** in Python. Objects are everywhere! Virtually everything that your Python program creates or acts on is an object.

An **assignment statement** creates a **symbolic name** that you can use to reference an object. The statement `x = 'foo'` creates a symbolic name `x` that refers to the string object `'foo'`.

In a program of any complexity, you'll create hundreds or thousands of such names, each pointing to a specific object. How does Python keep track of all these names so that they don't interfere with one another?

In this tutorial, you'll learn:

- How Python organizes symbolic names and objects in namespaces

• How Python organizes symbolic names and objects in namespaces

- When Python creates a new namespace
- How namespaces are implemented
- How **variable scope** determines symbolic name visibility

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Namespaces in Python

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a [dictionary](#) in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

Namespaces are one honking great idea—let's do more of those!

— [The Zen of Python](#), by Tim Peters

As Tim Peters suggests, namespaces aren't just great. They're *honking* great, and Python uses them extensively. In a Python program, there are four types of namespaces:

1. Built-In
2. Global
3. Enclosing
4. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they're no longer needed. Typically, many namespaces will exist at any given time.

The Built-In Namespace

The **built-in namespace** contains the names of all of Python's built-in objects. These are available at all times when Python is running. You can list the objects in the built-in namespace with the following command:

Python

>>>

```
>>> dir(__builtins__)
['ArithmetError', 'AssertionError', 'AttributeError',
 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
 'ImportError', 'ImportWarning', 'IndexError', 'IndentationError',
 'KeyError', 'KeyboardInterrupt', 'MemoryError', 'NameError', 'NotADirectoryError',
 'NotImplementedError', 'NotImplementedWarning', 'OverflowError', 'PendingDeprecationWarning',
 'RuntimeError', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemWarning',
 'TypeError', 'ValueError', 'Warning']
```

```
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FileExistsError', 'FileNotFoundException',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
'RuntimetypeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
['__doc__', '__import__', '__loader__', '__name__', '__package__',
 '__spec__', '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']
```

You'll see some objects here that you may recognize from previous tutorials—for example, the [StopIteration](#) exception, [built-in functions](#) like `max()` and `len()`, and object types like `int` and `str`.

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

The Global Namespace

The **global namespace** contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any **module** that your program loads with the [import](#) statement. For further reading on main functions and modules in Python, see these resources:

- [Defining Main Functions in Python](#)
- [Python Modules and Packages—An Introduction](#)
- [Course: Python Modules and Packages](#)

You'll explore modules in more detail in a future tutorial in this series. For the moment, when you see the term *global namespace*, think of the one belonging to the main program.

The Local and Enclosing Namespaces

As you learned in the previous tutorial on [functions](#), the interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Functions don't exist independently from one another only at the level of the main program. You can also define one function inside another:

```
Python >>> def f():
...     print('Start f()')
...
...     def g():
...         print('Start g()')
...         print('End g()')
...         return
...
...     g()
...
10 ...
```

```
11 ...     print('End f()')
12 ...     return
13 ...
14
15 >>> f()
16 Start f()
17 Start g()
18 End g()
19 End f()
```

In this example, function `g()` is defined within the body of `f()`. Here's what's happening in this code:

- **Lines 1 to 12** define `f()`, the **enclosing** function.
- **Lines 4 to 7** define `g()`, the **enclosed** function.
- On **line 15**, the main program calls `f()`.
- On **line 9**, `f()` calls `g()`.

When the main program calls `f()`, Python creates a new namespace for `f()`. Similarly, when `f()` calls `g()`, `g()` gets its own separate namespace. The namespace created for `g()` is the **local namespace**, and the namespace created for `f()` is the **enclosing namespace**.

Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

Variable Scope

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they're all maintained separately and won't interfere with one another.

But that raises a question: Suppose you refer to the name `x` in your code, and `x` exists in several namespaces. How does Python know which one you mean?

The answer lies in the concept of **scope**. The [scope](#) of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

Further Reading: See the Wikipedia page on [scope in computer programming](#) for a very thorough discussion of variable scope in programming languages.

To return to the above question, if your code refers to the name `x`, then Python searches for `x` in the following namespaces in the order shown:

1. **Local:** If you refer to `x` inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
2. **Enclosing:** If `x` isn't in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
3. **Global:** If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. **Built-in:** If it can't find `x` anywhere else, then the interpreter tries the built-in scope.

This is the **LEGB rule** as it's commonly called in Python literature (although the term doesn't actually appear in the [Python documentation](#)). The interpreter searches for a name from the inside out, looking in the **local**, **enclosing**, **global**, and finally the **built-in** scope:

If the interpreter doesn't find the name in any of these locations, then Python raises a [NameError exception](#).

Examples

Several examples of the LEGB rule appear below. In each case, the innermost enclosed function `g()` attempts to display the value of a variable named `x` to the console. Notice how each example prints a different value for `x` depending on its scope.

Example 1: Single Definition

In the first example, `x` is defined in only one location. It's outside both `f()` and `g()`, so it resides in the global scope:

```
Python >>>
1  >>> x = 'global'
2
3  >>> def f():
4  ...
5  ...     def g():
6  ...         print(x)
7  ...
8  ...     g()
9  ...
10
11 >>> f()
12 global
```

The `print()` statement on **line 6** can refer to only one possible `x`. It displays the `x` object defined in the global namespace, which is the string '`global`'.

Example 2: Double Definition

In the next example, the definition of `x` appears in two places, one outside `f()` and one inside `f()` but outside `g()`:

```
Python >>>
1  >>> x = 'global'
2
3  >>> def f():
4  ...     x = 'enclosing'
5  ...
6  ...     def g():
7  ...         print(x)
8  ...
9  ...     g()
10 ...
11
12 >>> f()
13 enclosing
```

As in the previous example, `g()` refers to `x`. But this time, it has two definitions to choose from:

- **Line 1** defines `x` in the global scope.

- **Line 4** defines `x` again in the enclosing scope.

According to the LEGB rule, the interpreter finds the value from the enclosing scope before looking in the global scope. So the `print()` statement on **line 7** displays 'enclosing' instead of 'global'.

Example 3: Triple Definition

Next is a situation in which `x` is defined here, there, and everywhere. One definition is outside `f()`, another one is inside `f()` but outside `g()`, and a third is inside `g()`:

Python

```

1 >>> x = 'global'
2
3 >>> def f():
4 ...     x = 'enclosing'
5 ...
6 ...     def g():
7 ...         x = 'local'
8 ...         print(x)
9 ...
10 ...     g()
11 ...
12
13 >>> f()
14 local

```

>>>

Now the `print()` statement on **line 8** has to distinguish between three different possibilities:

- **Line 1** defines `x` in the global scope.
- **Line 4** defines `x` again in the enclosing scope.
- **Line 7** defines `x` a third time in the scope that's local to `g()`.

Here, the LEGB rule dictates that `g()` sees its own locally defined value of `x` first. So the `print()` statement displays 'local'.

Example 4: No Definition

Last, we have a case in which `g()` tries to print the value of `x`, but `x` isn't defined anywhere. That won't work at all:

Python

```

1 >>> def f():
2 ...
3 ...     def g():
4 ...         print(x)
5 ...
6 ...     g()
7 ...
8
9 >>> f()
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12   File "<stdin>", line 6, in f
13   File "<stdin>", line 4, in g
14 NameError: name 'x' is not defined

```

>>>

This time, Python doesn't find `x` in any of the namespaces, so the `print()` statement on **line 4** generates a `NameError` exception.

Python Namespace Dictionaries

Earlier in this tutorial, when [namespaces were first introduced](#), you were encouraged to think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. In fact, for global and local namespaces, that's precisely what they are! Python really does implement these namespaces as dictionaries.

Note: The built-in namespace doesn't behave like a dictionary. Python implements it as a module.

Python provides built-in functions called `globals()` and `locals()` that allow you to access global and local namespace dictionaries.

The `globals()` function

The built-in function `globals()` returns a reference to the current global namespace dictionary. You can use it to access the objects in the global namespace. Here's an example of what it looks like when the main program starts:

```
Python >>>
>>> type(globals())
<class 'dict'>

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```

As you can see, the interpreter has put several entries in `globals()` already. Depending on your Python version and operating system, it may look a little different in your environment. But it should be similar.

Now watch what happens when you define a variable in the global scope:

```
Python >>>
>>> x = 'foo'

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
 'x': 'foo'}
```

After the assignment statement `x = 'foo'`, a new item appears in the global namespace dictionary. The dictionary key is the object's name, `x`, and the dictionary value is the object's value, `'foo'`.

You would typically access this object in the usual way, by referring to its symbolic name, `x`. But you can also access it indirectly through the global namespace dictionary:

```
Python >>>
1 |   1 >>> x
2 |   2 'foo'
3 |   3 >>> globals()['x']
4 |   4 'foo'
5 |
6 |   6 >>> x is globals()['x']
7 |   7 True
```

The [is comparison](#) on [line 6](#) confirms that these are in fact the same object.

You can create and modify entries in the global namespace using the `globals()` function as well:

```
Python >>>
1 |   1 >>> globals()['y'] = 100
2 |
3 |   2 >>> globals()
4 |   3 {'__name__': '__main__', '__doc__': None, '__package__': None,
5 |   4 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
6 |   5 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
7 |   6 'x': 'foo', 'y': 100}
8 |
9 |   9 >>> y
```

```

10 | 100
11 |
12 |>>> globals()['y'] = 3.14159
13 |
14 |>>> y
15 | 3.14159

```

The statement on **line 1** has the same effect as the assignment statement `y = 100`. The statement on **line 12** is equivalent to `y = 3.14159`.

It's a little off the beaten path to create and modify objects in the global scope this way when simple assignment statements will do. But it works, and it illustrates the concept nicely.

The `locals()` function

Python also provides a corresponding built-in function called `locals()`. It's similar to `globals()` but accesses objects in the local namespace instead:

```

Python >>>
>>> def f(x, y):
...     s = 'foo'
...     print(locals())
...
>>> f(10, 0.5)
{'s': 'foo', 'y': 0.5, 'x': 10}

```

When called within `f()`, `locals()` returns a dictionary representing the function's local namespace. Notice that, in addition to the locally defined variable `s`, the local namespace includes the function parameters `x` and `y` since these are local to `f()` as well.

If you call `locals()` outside a function in the main program, then it behaves the same as `globals()`.

Deep Dive: A Subtle Difference Between `globals()` and `locals()`

There's one small difference between `globals()` and `locals()` that's useful to know about.

`globals()` returns an actual reference to the dictionary that contains the global namespace. That means if you call `globals()`, save the return value, and subsequently define additional variables, then those new variables will show up in the dictionary that the saved return value points to:

```

Python >>>
1 |>>> g = globals()
2 |>>> g
3 |{'__name__': '__main__', '__doc__': None, '__package__': None,
4 |'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
5 |'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
6 |'g': {...}}
7 |
8 |>>> x = 'foo'
9 |>>> y = 29
10|>>> g
11|{'__name__': '__main__', '__doc__': None, '__package__': None,
12|'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
13|'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
14|'g': {...}, 'x': 'foo', 'y': 29}

```

Here, `g` is a reference to the global namespace dictionary. After the assignment statements on **lines 8 and 9**, `x` and `y` appear in the dictionary that `g` points to.

`locals()`, on the other hand, returns a dictionary that is a current copy of the local namespace, not a reference to it. Further additions to the local namespace won't affect a previous return value from `locals()`.

until you call it again. Also, you can't modify objects in the actual local namespace using the return value from `locals()`:

Python

>>>

```
1 >>> def f():
2 ...     s = 'foo'
3 ...     loc = locals()
4 ...     print(loc)
5 ...
6 ...     x = 20
7 ...     print(loc)
8 ...
9 ...     loc['s'] = 'bar'
10 ...    print(s)
11 ...
12
13 >>> f()
14 {'s': 'foo'}
15 {'s': 'foo'}
16 foo
```

In this example, `loc` points to the return value from `locals()`, which is a copy of the local namespace. The statement `x = 20` on **line 6** adds `x` to the local namespace but *not* to the copy that `loc` points to. Similarly, the statement on **line 9** modifies the value for key '`s`' in the copy that `loc` points to, but this has no effect on the value of `s` in the actual local namespace.

It's a subtle difference, but it could cause you trouble if you don't remember it.

Modify Variables Out of Scope

Earlier in this series, in the tutorial on [user-defined Python functions](#), you learned that argument passing in Python is a bit like [pass-by-value](#) and a bit like [pass-by-reference](#). Sometimes a function can modify its argument in the calling environment by making changes to the corresponding parameter, and sometimes it can't:

- An **immutable** argument can never be modified by a function.
- A **mutable** argument can't be redefined wholesale, but it can be modified in place.

Note: For more information on modifying function arguments, see [Pass-By-Value vs Pass-By-Reference in Pascal](#) and [Pass-By-Value vs Pass-By-Reference in Python](#).

A similar situation exists when a function tries to modify a variable outside its local scope. A function can't modify an immutable object outside its local scope at all:

Python

>>>

```
1 >>> x = 20
2 >>> def f():
3 ...     x = 40
4 ...     print(x)
5 ...
6
7 >>> f()
8 40
9 >>> x
10 20
```

When `f()` executes the assignment `x = 40` on **line 3**, it creates a new local [reference](#) to an integer object whose value is 40. At that point, `f()` loses the reference to the object named `x` in the global namespace. So the assignment statement doesn't affect the global object.

Note that when `f()` executes `print(x)` on **line 4**, it displays 40, the value of its own local `x`. But after `f()` terminates, `x` in the global scope is still 20.

A function can modify an object of mutable type that's outside its local scope if it modifies the object in place:

Python

>>>

...>>> x = [1, 2, 3]

```
>>> my_list = ['foo', 'bar', 'baz']
>>> def f():
...     my_list[1] = 'quux'
...
>>> f()
>>> my_list
['foo', 'quux', 'baz']
```

In this case, `my_list` is a list, and lists are mutable. `f()` can make changes inside `my_list` even though it's outside the local scope.

But if `f()` tries to reassign `my_list` entirely, then it will create a new local object and won't modify the global `my_list`:

```
Python >>>
>>> my_list = ['foo', 'bar', 'baz']
>>> def f():
...     my_list = ['qux', 'quux']
...
>>> f()
>>> my_list
['foo', 'bar', 'baz']
```

This is similar to what happens when `f()` tries to modify a mutable function argument.

The global Declaration

What if you really do need to modify a value in the global scope from within `f()`? This is possible in Python using the `global` declaration:

```
Python >>>
>>> x = 20
>>> def f():
...     global x
...     x = 40
...     print(x)
...
>>> f()
40
>>> x
40
```

The `global x` statement indicates that while `f()` executes, references to the name `x` will refer to the `x` that is in the global namespace. That means the assignment `x = 40` doesn't create a new reference. It assigns a new value to `x` in the global scope instead:

The global Declaration

As you've already seen, `globals()` returns a reference to the global namespace dictionary. If you wanted to, instead of using a `global` statement, you could accomplish the same thing using `globals()`:

```
>>> x = 20
>>> def f():
...     globals()['x'] = 40
...     print(x)
...
>>> f()
40
>>> x
40
```

There isn't much reason to do it this way since the `global` declaration arguably makes the intent clearer. But it does provide another illustration of how `globals()` works.

If the name specified in the `global` declaration doesn't exist in the global scope when the function starts, then a combination of the `global` statement and an assignment will create it:

```
Python >>>
1 |>>> y
2 |Traceback (most recent call last):
3 |  File "<pyshell#79>", line 1, in <module>
4 |    y
5 |NameError: name 'y' is not defined
6 |
7 |>>> def g():
8 |...     global y
9 |...     y = 20
10 |...
11 |
12 |>>> g()
13 |>>> y
14 |20
```

In this case, there's no object named `y` in the global scope when `g()` starts, but `g()` creates one with the `global y` statement on [line 8](#).

You can also specify several comma-separated names in a single `global` declaration:

```
Python >>>
1 |>>> x, y, z = 10, 20, 30
2 |
3 |>>> def f():
4 |...     global x, y, z
5 |...
```

Here, `x`, `y`, and `z` are all declared to refer to objects in the global scope by the single `global` statement on [line 4](#).

A name specified in a `global` declaration can't appear in the function prior to the `global` statement:

```
Python >>>
1 |>>> def f():
2 |...     print(x)
3 |...     global x
4 |...
5 |  File "<stdin>", line 3
6 |SyntaxError: name 'x' is used prior to global declaration
```

The intent of the `global x` statement on [line 3](#) is to make references to `x` refer to an object in the global scope. But

The intent of the global x statement on line 5 is to make references to x refer to an object in the global scope. But the print() statement on line 2 refers to x prior to the global declaration. This raises a SyntaxError exception.

The nonlocal Declaration

A similar situation exists with nested function definitions. The global declaration allows a function to access and modify an object in the global scope. What if an enclosed function needs to modify an object in the enclosing scope? Consider this example:

Python

```
>>> def f():
...     x = 20
...
...     def g():
...         x = 40
...
...     g()
...     print(x)
...
>>> f()
20
```

>>>

In this case, the first definition of x is in the enclosing scope, not the global scope. Just as g() can't directly modify a variable in the global scope, neither can it modify x in the enclosing function's scope. Following the assignment x = 40 on line 5, x in the enclosing scope remains 20.

The [global keyword](#) isn't a solution for this situation:

Python

```
>>> def f():
...     x = 20
...
...     def g():
...         global x
...         x = 40
...
...     g()
...     print(x)
...
>>> f()
20
```

>>>

Since x is in the enclosing function's scope, not the global scope, the global keyword doesn't work here. After g() terminates, x in the enclosing scope remains 20.

In fact, in this example, the global x statement not only fails to provide access to x in the enclosing scope, but it also creates an object called x in the global scope whose value is 40:

Python

```
>>> def f():
...     x = 20
...
...     def g():
...         global x
...         x = 40
...
...     g()
...     print(x)
...
>>> f()
40
```

>>>

```

...
...     def g():
...         global x
...         x = 40
...
...
...     g()
...     print(x)
...

>>> f()
20
>>> x
40

```

To modify `x` in the enclosing scope from inside `g()`, you need the analogous keyword `nonlocal`. Names specified after the `nonlocal` keyword refer to variables in the nearest enclosing scope:

Python

```

1 >>> def f():
2 ...     x = 20
3 ...
4 ...     def g():
5 ...         nonlocal x
6 ...         x = 40
7 ...
8 ...     g()
9 ...     print(x)
10 ...
11
12 >>> f()
13 40

```

>>>

After the `nonlocal x` statement on [line 5](#), when `g()` refers to `x`, it refers to the `x` in the nearest enclosing scope, whose definition is in `f()` on [line 2](#):

The nonlocal Declaration

The `print()` statement at the end of `f()` on [line 9](#) confirms that the call to `g()` has changed the value of `x` in the enclosing scope to 40.

Best Practices

Even though Python provides the `global` and `nonlocal` keywords, it's not always advisable to use them.

When a function modifies data outside the local scope, either with the `global` or `nonlocal` keyword or by directly modifying a mutable type in place, it's a kind of [side effect](#) similar to when a function modifies one of its arguments. Widespread modification of global variables is generally considered unwise, not only in Python but also in other programming languages.

As with many things, this is somewhat a matter of style and preference. There are times when judicious use of global variable modification can reduce program complexity.

In Python, using the `global` keyword at least makes it explicit that the function is modifying a global variable. In many languages, a function can modify a global variable just by assignment, without announcing it in any way. This can make it very difficult to track down where global data is being modified.

All in all, modifying variables outside the local scope usually isn't necessary. There's almost always a better way, usually with function return values.

Conclusion

Virtually everything that a Python program uses or acts on is an object. Even a short program will create many different objects. In a more complex program, they'll probably number in the thousands. Python has to keep track of all these objects and their names, and it does so with **namespaces**.

In this tutorial, you learned:

- What the different **namespaces** are in Python
- When Python creates a new namespace
- What structure Python uses to implement namespaces
- How namespaces define **scope** in a Python program

Many programming techniques take advantage of the fact that every function in Python has its own namespace. In the next two tutorials in this series, you'll explore two of these techniques: **functional programming** and **recursion**.

[« Regular Expressions:
Regexes in Python \(Part 2\)](#)

[Namespaces and Scope in
Python](#)

About John Sturtz

John is an avid Pythonista and a member of the Real Python tutorial team.

[» More about John](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

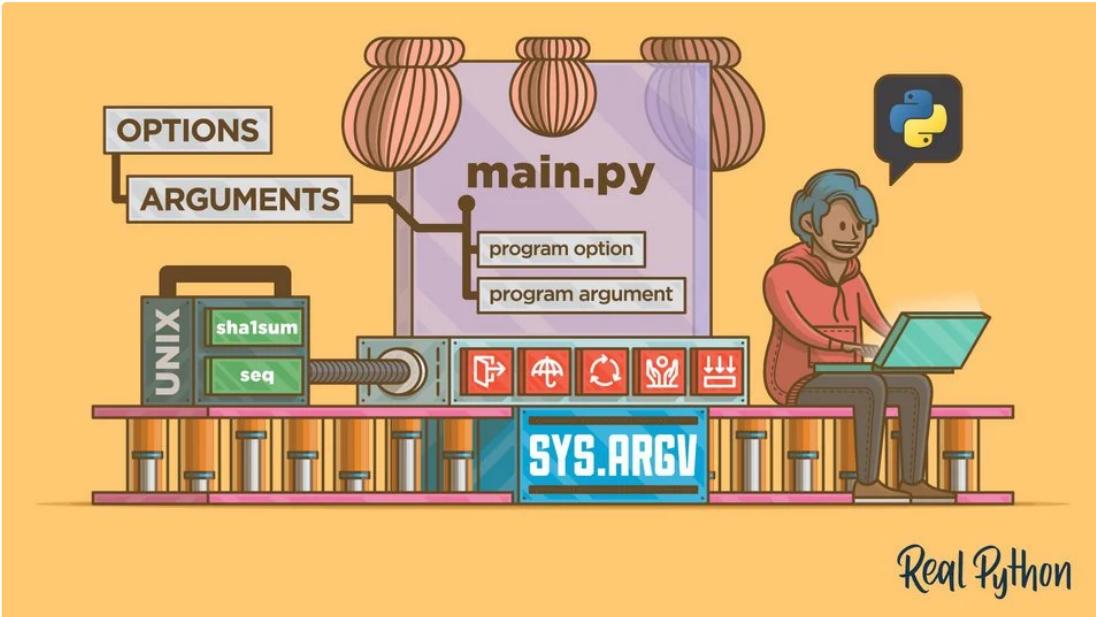
[Bryan](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)



Real Python

Python Command Line Arguments

by Andre Burgaud · Feb 05, 2020 · 4 Comments · best-practices · intermediate · tools

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [The Command Line Interface](#)
- [The C Legacy](#)
- [Two Utilities From the Unix World](#)
 - [sha1sum](#)
 - [seq](#)
- [The sys.argv Array](#)
 - [Displaying Arguments](#)
 - [Reversing the First Argument](#)
 - [Mutating sys.argv](#)
 - [Escaping Whitespace Characters](#)
 - [Handling Errors](#)
 - [Calculating the sha1sum](#)
- [The Anatomy of Python Command Line Arguments](#)
 - [Standards](#)
 - [Options](#)
 - [Arguments](#)
 - [Subcommands](#)
 - [Windows](#)
 - [Visuals](#)
- [A Few Methods for Parsing Python Command Line Arguments](#)
 - [Regular Expressions](#)
 - [File Handling](#)
 - [Standard Input](#)
 - [Standard Output and Standard Error](#)
 - [Custom Parsers](#)
- [A Few Methods for Validating Python Command Line Arguments](#)
 - [Type Validation With Python Data Classes](#)
 - [Custom Validation](#)
- [The Python Standard Library](#)
 - [argparse](#)
 - [getopt](#)

- [A Few External Python Packages](#)
 - [Click](#)
 - [Python Prompt Toolkit](#)
- [Conclusion](#)
- [Additional Resources](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Command Line Interfaces in Python](#)

Adding the capability of processing **Python command line arguments** provides a user-friendly interface to your text-based command line program. It's similar to what a graphical user interface is for a visual application that's manipulated by graphical elements or widgets.

Python exposes a mechanism to capture and extract your Python command line arguments. These values can be used to modify the behavior of a program. For example, if your program processes data read from a file, then you can pass the name of the file to your program, rather than hard-coding the value in your source code.

By the end of this tutorial, you'll know:

- **The origins** of Python command line arguments
- **The underlying support** for Python command line arguments
- **The standards** guiding the design of a command line interface
- **The basics** to manually customize and handle Python command line arguments
- **The libraries** available in Python to ease the development of a complex command line interface

If you want a user-friendly way to supply Python command line arguments to your program without importing a dedicated library, or if you want to better understand the common basis for the existing libraries that are dedicated to building the Python command line interface, then keep on reading!

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

The Command Line Interface

A [command line interface \(CLI\)](#) provides a way for a user to interact with a program running in a text-based [shell](#) interpreter. Some examples of shell interpreters are [Bash](#) on Linux or [Command Prompt](#) on Windows. A command line interface is enabled by the shell interpreter that exposes a [command prompt](#). It can be characterized by the following elements:

- A **command** or program
- Zero or more command line **arguments**
- An **output** representing the result of the command
- Textual documentation referred to as **usage** or **help**

Not every command line interface may provide all these elements, but this list isn't exhaustive, either. The complexity of the command line ranges from the ability to pass a single argument, to numerous arguments and options, much like a [Domain Specific Language](#). For example, some programs may launch web documentation from the command line or start an [interactive shell interpreter](#) like Python.

The two following examples with the Python command illustrates the description of a command line interface:

Shell

```
$ python -c "print('Real Python')"
Real Python
```

In this first example, the Python interpreter takes option -c for **command**, which says to execute the Python command line arguments following the option -c as a Python program.

Another example shows how to invoke Python with -h to display the help:

Shell

```
$ python -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b      : issue warnings about str(bytes_instance), str(bytarray_instance)
          and comparing bytes/bytarray with str. (-bb: issue errors)
[ ... complete help text not shown ... ]
```

Try this out in your terminal to see the complete help documentation.

The C Legacy

Python command line arguments directly inherit from the [C](#) programming language. As [Guido Van Rossum](#) wrote in [An Introduction to Python for Unix/C Programmers](#) in 1993, C had a strong influence on Python. Guido mentions the definitions of literals, identifiers, operators, and statements like break, continue, or return. The use of Python command line arguments is also strongly influenced by the C language.

To illustrate the similarities, consider the following C program:

C

```
1 // main.c
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5     printf("Arguments count: %d\n", argc);
6     for (int i = 0; i < argc; i++) {
7         printf("Argument %6d: %s\n", i, argv[i]);
8     }
9     return 0;
10 }
```

Line 4 defines [main\(\)](#), which is the entry point of a C program. Take good note of the parameters:

1. **argc** is an integer representing the number of arguments of the program.
2. **argv** is an array of pointers to characters containing the name of the program in the first element of the array, followed by the arguments of the program, if any, in the remaining elements of the array.

You can compile the code above on Linux with `gcc -o main main.c`, then execute with `./main` to obtain the following:

Shell

```
$ gcc -o main main.c
$ ./main
Arguments count: 1
Argument      0: ./main
```

Unless explicitly expressed at the command line with the option `-o`, [a.out](#) is the default name of the executable generated by the **gcc** compiler. It stands for **assembler output** and is reminiscent of the executables that were generated on older UNIX systems. Observe that the name of the executable `./main` is the sole argument.

Let's spice up this example by passing a few Python command line arguments to the same program:

Shell

```
$ ./main Python Command Line Arguments
Arguments count: 5
Argument      0: ./main
```

```
Argument      1: Python
Argument      2: Command
Argument      3: Line
Argument      4: Arguments
```

The output shows that the number of arguments is 5, and the list of arguments includes the name of the program, `main`, followed by each word of the phrase "Python Command Line Arguments", which you passed at the command line.

Note: `argc` stands for **argument count**, while `argv` stands for **argument vector**. To learn more, check out [A Little C Primer/C Command Line Arguments](#).

The compilation of `main.c` assumes that you used a Linux or a Mac OS system. On Windows, you can also compile this C program with one of the following options:

- **Windows Subsystem for Linux (WSL):** It's available in a few Linux distributions, like [Ubuntu](#), [OpenSUSE](#), and [Debian](#), among others. You can install it from the Microsoft Store.
- **Windows Build Tools:** This includes the Windows command line build tools, the Microsoft C/C++ compiler [cl.exe](#), and a compiler front end named [clang.exe](#) for C/C++.
- **Microsoft Visual Studio:** This is the main Microsoft integrated development environment (IDE). To learn more about IDEs that can be used for both Python and C on various operating systems, including Windows, check out [Python IDEs and Code Editors \(Guide\)](#).
- **mingw-64 project:** This supports the [GCC compiler](#) on Windows.

If you've installed Microsoft Visual Studio or the Windows Build Tools, then you can compile `main.c` as follows:

Windows Console

```
C:/>cl main.c
```

You'll obtain an executable named `main.exe` that you can start with:

Windows Console

```
C:/>main
Arguments count: 1
Argument      0: main
```

You could implement a Python program, `main.py`, that's equivalent to the C program, `main.c`, you saw above:

Python

```
# main.py
import sys

if __name__ == "__main__":
    print(f"Arguments count: {len(sys.argv)}")
    for i, arg in enumerate(sys.argv):
        print(f"Argument {i+1}: {arg}")
```

You don't see an `argc` variable like in the C code example. It doesn't exist in Python because `sys.argv` is sufficient. You can parse the Python command line arguments in `sys.argv` without having to know the length of the list, and you can call the built-in `len()` if the number of arguments is needed by your program.

Also, note that `enumerate()`, when applied to an iterable, returns an `enumerate` object that can emit pairs associating the index of an element in `sys.argv` to its corresponding value. This allows looping through the content of `sys.argv` without having to maintain a counter for the index in the list.

Execute `main.py` as follows:

Shell

```
$ python main.py Python Command Line Arguments
Arguments count: 5
Argument      0: main nv
```

```
Argument 1: Python
Argument 2: Command
Argument 3: Line
Argument 4: Arguments
```

sys.argv contains the same information as in the C program:

- **The name of the program** main.py is the first item of the list.
- **The arguments** Python, Command, Line, and Arguments are the remaining elements in the list.

With this short introduction into a few arcane aspects of the C language, you're now armed with some valuable knowledge to further grasp Python command line arguments.

Two Utilities From the Unix World

To use Python command line arguments in this tutorial, you'll implement some partial features of two utilities from the Unix ecosystem:

1. [sha1sum](#)
2. [seq](#)

You'll gain some familiarity with these Unix tools in the following sections.

sha1sum

sha1sum calculates [SHA-1 hashes](#), and it's often used to verify the integrity of files. For a given input, a **hash function** always returns the same value. Any minor changes in the input will result in a different hash value. Before you use the utility with concrete parameters, you may try to display the help:

Shell

```
$ sha1sum --help
Usage: sha1sum [OPTION]... [FILE]...
Print or check SHA1 (160-bit) checksums.

With no FILE, or when FILE is -, read standard input.

-b, --binary      read in binary mode
-c, --check       read SHA1 sums from the FILES and check them
--tag            create a BSD-style checksum
-t, --text        read in text mode (default)
-z, --zero        end each output line with NUL, not newline,
                  and disable file name escaping
[ ... complete help text not shown ... ]
```

Displaying the help of a command line program is a common feature exposed in the command line interface.

To calculate the SHA-1 hash value of the content of a file, you proceed as follows:

Shell

```
$ sha1sum main.c
125a0f900ff6f164752600550879cbfab098bc3  main.c
```

The result shows the SHA-1 hash value as the first field and the name of the file as the second field. The command can take more than one file as arguments:

Shell

```
$ sha1sum main.c main.py
125a0f900ff6f164752600550879cbfab098bc3  main.c
d84372fc77a90336b6bb7c5e959bcb1b24c608b4  main.py
```

Thanks to the wildcards expansion feature of the Unix terminal, it's also possible to provide Python command line arguments with wildcard characters. One such a character is the asterisk or star (*):

Shell

```
$ sha1sum main.*  
3f6d5274d6317d580e2ffc1bf52beee0d94bf078  main.c  
f41259ea5835446536d2e71e566075c1c1bfc111  main.py
```

The shell converts `main.*` to `main.c` and `main.py`, which are the two files matching the pattern `main.*` in the current directory, and passes them to `sha1sum`. The program calculates the **SHA1 hash** of each of the files in the argument list. You'll see that, on Windows, the behavior is different. Windows has no wildcard expansion, so the program may have to accommodate for that. Your implementation may need to expand wildcards internally.

Without any argument, `sha1sum` reads from the standard input. You can feed data to the program by typing characters on the keyboard. The input may incorporate any characters, including the carriage return `Enter ↵`. To terminate the input, you must signal the [end of file](#) with `Enter ↵`, followed by the sequence `^Ctrl + D`:

Shell

```
1 $ sha1sum  
2 Real  
3 Python  
4 87263a73c98af453d68ee4aab61576b331f8d9d6 -
```

You first enter the name of the program, `sha1sum`, followed by `Enter ↵`, and then `Real` and `Python`, each also followed by `Enter ↵`. To close the input stream, you type `^Ctrl + D`. The result is the value of the SHA1 hash generated for the text `Real\nPython\n`. The name of the file is `-`. This is a convention to indicate the standard input. The hash value is the same when you execute the following commands:

Shell

```
$ python -c "print('Real\nPython\n', end='')" | sha1sum  
87263a73c98af453d68ee4aab61576b331f8d9d6 -  
$ python -c "print('Real\nPython')" | sha1sum  
87263a73c98af453d68ee4aab61576b331f8d9d6 -  
$ printf "Real\nPython\n" | sha1sum  
87263a73c98af453d68ee4aab61576b331f8d9d6 -
```

Up next, you'll read a short description of `seq`.

seq

`seq` generates a **sequence** of numbers. In its most basic form, like generating the sequence from 1 to 5, you can execute the following:

Shell

```
$ seq 5  
1  
2  
3  
4  
5
```

To get an overview of the possibilities exposed by `seq`, you can display the help at the command line:

Shell

```
$ seq --help  
Usage: seq [OPTION]... LAST  
      or: seq [OPTION]... FIRST LAST  
      or: seq [OPTION]... FIRST INCREMENT LAST  
Print numbers from FIRST to LAST, in steps of INCREMENT.  
  
Mandatory arguments to long options are mandatory for short options too.  
-f, --format=FORMAT      use printf style floating-point FORMAT  
-s, --separator=STRING   use STRING to separate numbers (default: '\n')  
-w, --equal-width        equalize width by padding with leading zeroes  
--help                  display this help and exit
```

```
--version  output version information and exit
[ ... complete help text not shown ... ]
```

For this tutorial, you'll write a few simplified variants of `sha1sum` and `seq`. In each example, you'll learn a different facet or combination of features about Python command line arguments.

On Mac OS and Linux, `sha1sum` and `seq` should come pre-installed, though the features and the help information may sometimes differ slightly between systems or distributions. If you're using Windows 10, then the most convenient method is to run `sha1sum` and `seq` in a Linux environment installed on the [WSL](#). If you don't have access to a terminal exposing the standard Unix utilities, then you may have access to online terminals:

- **Create** a free account on [PythonAnywhere](#) and start a Bash Console.
- **Create** a temporary Bash terminal on [repl.it](#).

These are two examples, and you may find others.

The `sys.argv` Array

Before exploring some accepted conventions and discovering how to handle Python command line arguments, you need to know that the underlying support for all Python command line arguments is provided by `sys.argv`. The examples in the following sections show you how to handle the Python command line arguments stored in `sys.argv` and to overcome typical issues that occur when you try to access them. You'll learn:

- How to **access** the content of `sys.argv`
- How to **mitigate** the side effects of the global nature of `sys.argv`
- How to **process** whitespaces in Python command line arguments
- How to **handle** errors while accessing Python command line arguments
- How to **ingest** the original format of the Python command line arguments passed by bytes

Let's get started!

Displaying Arguments

The `sys` module exposes an array named `argv` that includes the following:

1. `argv[0]` contains the name of the current Python program.
2. `argv[1:]`, the rest of the list, contains any and all Python command line arguments passed to the program.

The following example demonstrates the content of `sys.argv`:

Python

```
1 # argv.py
2 import sys
3
4 print(f"Name of the script : {sys.argv[0]}")
5 print(f"Arguments of the script : {sys.argv[1:]}")
```

Here's how this code works:

- **Line 2** imports the internal Python module `sys`.
- **Line 4** extracts the name of the program by accessing the first element of the list `sys.argv`.
- **Line 5** displays the Python command line arguments by fetching all the remaining elements of the list `sys.argv`.

Note: The `f-string` syntax used in `argv.py` leverages the new debugging specifier in Python 3.8. To read more about this new `f-string` feature and others, check out [Cool New Features in Python 3.8](#).

If your Python version is less than 3.8, then simply remove the equals sign (=) in both `f-strings` to allow the program to execute successfully. The output will only display the value of the variables, not their names.

Execute the script `argv.py` above with a list of arbitrary arguments as follows:

Shell

```
$ python argv.py un deux trois quatre
Name of the script      : sys.argv[0]='argv.py'
Arguments of the script : sys.argv[1:]=['un', 'deux', 'trois', 'quatre']
```

The output confirms that the content of `sys.argv[0]` is the Python script `argv.py`, and that the remaining elements of the `sys.argv` list contains the arguments of the script, `['un', 'deux', 'trois', 'quatre']`.

To summarize, `sys.argv` contains all the `argv.py` Python command line arguments. When the Python interpreter executes a Python program, it parses the command line and populates `sys.argv` with the arguments.

Reversing the First Argument

Now that you have enough background on `sys.argv`, you're going to operate on arguments passed at the command line. The example `reverse.py` reverses the first argument passed at the command line:

Python

```
1 # reverse.py
2
3 import sys
4
5 arg = sys.argv[1]
6 print(arg[::-1])
```

In `reverse.py` the process to reverse the first argument is performed with the following steps:

- **Line 5** fetches the first argument of the program stored at index 1 of `sys.argv`. Remember that the program name is stored at index 0 of `sys.argv`.
- **Line 6** prints the reversed string. `args[::-1]` is a Pythonic way to use a slice operation to [reverse a list](#).

You execute the script as follows:

Shell

```
$ python reverse.py "Real Python"
nohtyP laeR
```

As expected, `reverse.py` operates on `"Real Python"` and reverses the only argument to output `"nohtyP laeR"`. Note that surrounding the multi-word string `"Real Python"` with quotes ensures that the interpreter handles it as a unique argument, instead of two arguments. You'll delve into **argument separators** in a later [section](#).

Mutating `sys.argv`

`sys.argv` is **globally available** to your running Python program. All modules imported during the execution of the process have direct access to `sys.argv`. This global access might be convenient, but `sys.argv` isn't immutable. You may want to implement a more reliable mechanism to expose program arguments to different modules in your Python program, especially in a complex program with multiple files.

Observe what happens if you tamper with `sys.argv`:

Python

```
# argv_pop.py
import sys
print(sys.argv)
sys.argv.pop()
print(sys.argv)
```

You invoke `.pop()` to remove and return the last item in `sys.argv`.

Execute the script above:

Shell

```
$ python argv_pop.py un deux trois quatre
['argv_pop.py', 'un', 'deux', 'trois', 'quatre']
['argv_pop.py', 'un', 'deux', 'trois']
```

Notice that the fourth argument is no longer included in sys.argv.

In a short script, you can safely rely on the global access to sys.argv, but in a larger program, you may want to store arguments in a separate variable. The previous example could be modified as follows:

Python

```
# argv_var_pop.py

import sys

print(sys.argv)
args = sys.argv[1:]
print(args)
sys.argv.pop()
print(sys.argv)
print(args)
```

This time, although sys.argv lost its last element, args has been safely preserved. args isn't global, and you can pass it around to parse the arguments per the logic of your program. The Python package manager, [pip](#), uses this [approach](#). Here's a short excerpt of the pip source code:

Python

```
def main(args=None):
    if args is None:
        args = sys.argv[1:]
```

In this snippet of code taken from the [pip](#) source code, main() saves into args the slice of sys.argv that contains only the arguments and not the file name. sys.argv remains untouched, and args isn't impacted by any inadvertent changes to sys.argv.

Escaping Whitespace Characters

In the reverse.py example you saw [earlier](#), the first and only argument is "Real Python", and the result is "nohtyP laeR". The argument includes a whitespace separator between "Real" and "Python", and it needs to be escaped.

On Linux, whitespaces can be escaped by doing one of the following:

1. **Surrounding** the arguments with single quotes (')
2. **Surrounding** the arguments with double quotes (")
3. **Prefixing** each space with a backslash (\)

Without one of the escape solutions, reverse.py stores two arguments, "Real" in sys.argv[1] and "Python" in sys.argv[2]:

Shell

```
$ python reverse.py Real Python
laeR
```

The output above shows that the script only reverses "Real" and that "Python" is ignored. To ensure both arguments are stored, you'd need to surround the overall string with double quotes ("").

You can also use a backslash (\) to escape the whitespace:

Shell

```
$ python reverse.py Real\ Python
nohtyP laeR
```

With the backslash (\), the command shell exposes a unique argument to Python, and then to reverse.py.

In Unix shells, the [internal field separator \(IFS\)](#) defines characters used as **delimiters**. The content of the shell variable, IFS, can be displayed by running the following command:

Shell

```
$ printf "%q\n" "$IFS"
$' \t\n'
```

From the result above, ' \t\n', you identify three delimiters:

1. **Space** (' ')
2. **Tab** (\t)
3. **Newline** (\n)

Prefixing a space with a backslash (\) bypasses the default behavior of the space as a delimiter in the string "Real Python". This results in one block of text as intended, instead of two.

Note that, on Windows, the whitespace interpretation can be managed by using a combination of double quotes. It's slightly counterintuitive because, in the Windows terminal, a double quote ("") is interpreted as a switch to disable and subsequently to enable special characters like **space**, **tab**, or **pipe** (|).

As a result, when you surround more than one string with double quotes, the Windows terminal interprets the first double quote as a command to **ignore special characters** and the second double quote as one to **interpret special characters**.

With this information in mind, it's safe to assume that surrounding more than one string with double quotes will give you the expected behavior, which is to expose the group of strings as a single argument. To confirm this peculiar effect of the double quote on the Windows command line, observe the following two examples:

Windows Console

```
C:/>python reverse.py "Real Python"
nohtyP laeR
```

In the example above, you can intuitively deduce that "Real Python" is interpreted as a single argument. However, realize what occurs when you use a single double quote:

Windows Console

```
C:/>python reverse.py "Real Python"
nohtyP laeR
```

The command prompt passes the whole string "Real Python" as a single argument, in the same manner as if the argument was "Real Python". In reality, the Windows command prompt sees the unique double quote as a switch to disable the behavior of the whitespaces as separators and passes anything following the double quote as a unique argument.

For more information on the effects of double quotes in the Windows terminal, check out [A Better Way To Understand Quoting and Escaping of Windows Command Line Arguments](#).

Handling Errors

Python command line arguments are **loose strings**. Many things can go wrong, so it's a good idea to provide the users of your program with some guidance in the event they pass incorrect arguments at the command line. For example, reverse.py expects one argument, and if you omit it, then you get an error:

Shell

```
1 $ python reverse.py
2 Traceback (most recent call last):
3   File "reverse.py", line 5, in <module>
4     arg = sys.argv[1]
5 IndexError: list index out of range
```

The Python [exception](#) `IndexError` is raised, and the corresponding [traceback](#) shows that the error is caused by the expression `arg = sys.argv[1]`. The message of the exception is `list index out of range`. You didn't pass an argument at the command line, so there's nothing in the list `sys.argv` at index 1.

This is a common pattern that can be addressed in a few different ways. For this initial example, you'll keep it brief by including the expression `arg = sys.argv[1]` in a `try` block. Modify the code as follows:

Python

```
1 # reverse_exc.py
2
3 import sys
4
5 try:
6     arg = sys.argv[1]
7 except IndexError:
8     raise SystemExit(f"Usage: {sys.argv[0]} <string_to_reverse>")
9 print(arg[::-1])
```

The expression on line 4 is included in a `try` block. Line 8 raises the built-in exception [SystemExit](#). If no argument is passed to `reverse_exc.py`, then the process exits with a status code of 1 after printing the usage. Note the integration of `sys.argv[0]` in the error message. It exposes the name of the program in the usage message. Now, when you execute the same program without any Python command line arguments, you can see the following output:

Shell

```
$ python reverse.py
Usage: reverse.py <string_to_reverse>

$ echo $?
1
```

`reverse.py` didn't have an argument passed at the command line. As a result, the program raises `SystemExit` with an error message. This causes the program to exit with a status of 1, which displays when you print the special variable `$?` with `echo`.

Calculating the sha1sum

You'll write another script to demonstrate that, on [Unix-like](#) systems, Python command line arguments are passed by bytes from the OS. This script takes a string as an argument and outputs the hexadecimal [SHA-1](#) hash of the argument:

Python

```
1 # sha1sum.py
2
3 import sys
4 import hashlib
5
6 data = sys.argv[1]
7 m = hashlib.sha1()
8 m.update(bytes(data, 'utf-8'))
9 print(m.hexdigest())
```

This is loosely inspired by `sha1sum`, but it intentionally processes a string instead of the contents of a file. In `sha1sum.py`, the steps to ingest the Python command line arguments and to output the result are the following:

- **Line 6** stores the content of the first argument in `data`.
- **Line 7** instantiates a SHA1 algorithm

- Line 7 instantiates a SHA1 algorithm.
- Line 8 updates the SHA1 hash object with the content of the first program argument. Note that `hash.update` takes a byte array as an argument, so it's necessary to convert data from a string to a bytes array.
- Line 9 prints a [hexadecimal representation](#) of the SHA1 hash computed on line 8.

When you run the script with an argument, you get this:

Shell

```
$ python sha1sum.py "Real Python"
0554943d034f044c5998f55dac8ee2c03e387565
```

For the sake of keeping the example short, the script `sha1sum.py` doesn't handle missing Python command line arguments. Error handling could be addressed in this script the same way you did it in `reverse_exc.py`.

Note: Checkout [hashlib](#) for more details about the hash functions available in the Python standard library.

From the `sys.argv` [documentation](#), you learn that in order to get the original bytes of the Python command line arguments, you can use `os.fsenconde()`. By directly obtaining the bytes from `sys.argv[1]`, you don't need to perform the string-to-bytes conversion of data:

Python

```
1 # sha1sum_bytes.py
2
3 import os
4 import sys
5 import hashlib
6
7 data = os.fsenconde(sys.argv[1])
8 m = hashlib.sha1()
9 m.update(data)
10 print(m.hexdigest())
```

The main difference between `sha1sum.py` and `sha1sum_bytes.py` are highlighted in the following lines:

- Line 7 populates data with the original bytes passed to the Python command line arguments.
- Line 9 passes data as an argument to `m.update()`, which receives a [bytes-like object](#).

Execute `sha1sum_bytes.py` to compare the output:

Shell

```
$ python sha1sum_bytes.py "Real Python"
0554943d034f044c5998f55dac8ee2c03e387565
```

The hexadecimal value of the SHA1 hash is the same as in the previous `sha1sum.py` example.

The Anatomy of Python Command Line Arguments

Now that you've explored a few aspects of Python command line arguments, most notably `sys.argv`, you're going to apply some of the standards that are regularly used by developers while implementing a command line interface.

Python command line arguments are a subset of the command line interface. They can be composed of different types of arguments:

1. **Options** modify the behavior of a particular command or program.
2. **Arguments** represent the source or destination to be processed.
3. **Subcommands** allow a program to define more than one command with the respective set of options and arguments.

Before you go deeper into the different types of arguments, you'll get an overview of the accepted standards that have been guiding the design of the command line interface and arguments. These have been refined since the advent of the [computer terminal](#) in the mid-1960s.

Standards

A few available **standards** provide some definitions and guidelines to promote consistency for implementing commands and their arguments. These are the main UNIX standards and references:

- [POSIX Utility Conventions](#)
- [GNU Standards for Command Line Interfaces](#)
- [docopt](#)

The standards above define guidelines and nomenclatures for anything related to programs and Python command line arguments. The following points are examples taken from those references:

- **POSIX:**
 - A program or utility is followed by options, option-arguments, and operands.
 - All options should be preceded with a hyphen or minus (-) delimiter character.
 - Option-arguments should not be optional.
- **GNU:**
 - All programs should support two standard options, which are --version and --help.
 - Long-named options are equivalent to the single-letter Unix-style options. An example is --debug and -d.
- **docopt:**
 - Short options can be stacked, meaning that -abc is equivalent to -a -b -c.
 - Long options can have arguments specified after a space or the equals sign (=). The long option --input=ARG is equivalent to --input ARG.

These standards define notations that are helpful when you describe a command. A similar notation can be used to display the usage of a particular command when you invoke it with the option -h or --help.

The GNU standards are very similar to the POSIX standards but provide some modifications and extensions. Notably, they add the **long option** that's a fully named option prefixed with two hyphens (--). For example, to display the help, the regular option is -h and the long option is --help.

Note: You don't need to follow those standards rigorously. Instead, follow the conventions that have been used successfully for years since the advent of UNIX. If you write a set of utilities for you or your team, then ensure that you **stay consistent** across the different utilities.

In the following sections, you'll learn more about each of the command line components, options, arguments, and sub-commands.

Options

An **option**, sometimes called a **flag** or a **switch**, is intended to modify the behavior of the program. For example, the command ls on Linux lists the content of a given directory. Without any arguments, it lists the files and directories in the current directory:

Shell

```
$ cd /dev
$ ls
autofs
block
bsg
btrfs-control
bus
char
console
```

Let's add a few options. You can combine -l and -s into -ls, which changes the information displayed in the terminal:

Shell

```
$ cd /dev
$ ls -ls
```

```
ls -ls  
total 0  
0 crw-r--r-- 1 root root 10, 235 Jul 14 08:10 autofs  
0 drwxr-xr-x 2 root root 260 Jul 14 08:10 block  
0 drwxr-xr-x 2 root root 60 Jul 14 08:10 bsg  
0 crw----- 1 root root 10, 234 Jul 14 08:10 btrfs-control  
0 drwxr-xr-x 3 root root 60 Jul 14 08:10 bus  
0 drwxr-xr-x 2 root root 4380 Jul 14 15:08 char  
0 crw----- 1 root root 5, 1 Jul 14 08:10 console
```

An **option** can take an argument, which is called an **option-argument**. See an example in action with `od` below:

Shell

od stands for **octal dump**. This utility displays data in different printable representations, like octal (which is the default), hexadecimal, decimal, and ASCII. In the example above, it takes the binary file `main` and displays the first 16 bytes of the file in hexadecimal format. The option `-t` expects a type as an option-argument, and `-N` expects the number of input bytes.

In the example above, `-t` is given type `x1`, which stands for hexadecimal and one byte per integer. This is followed by `z` to display the printable characters at the end of the input line. `-N` takes `16` as an option-argument for limiting the number of input bytes to `16`.

Arguments

The **arguments** are also called **operands** or parameters in the POSIX standards. The arguments represent the source or the destination of the data that the command acts on. For example, the command `cp`, which is used to copy one or more files to a file or a directory, takes at least one source and one target:

Shell

```
1 $ ls main  
2 main  
3  
4 $ cp main main2  
5  
6 $ ls -lt  
7 main  
8 main2  
9 ...
```

In line 4, cp takes two arguments:

1. **main**: the source file
 2. **main2**: the target file

It then copies the content of `main` to a new file named `main2`. Both `main` and `main2` are arguments, or operands, of the program `cp`.

Subcommands

The concept of **subcommands** isn't documented in the POSIX or GNU standards, but it does appear in [docopt](#). The standard Unix utilities are small tools adhering to the [Unix philosophy](#). Unix programs are intended to be programs that [do one thing and do it well](#). This means no subcommands are necessary.

By contrast, a new generation of programs, including [git](#), [go](#), [docker](#), and [gcloud](#), come with a slightly different paradigm that embraces subcommands. They're not necessarily part of the Unix landscape as they span several operating systems, and they're deployed with a full ecosystem that requires several commands.

Take `git` as an example. It includes several command-line arguments, each possibly with their own set of options, option-arguments, and arguments. The following examples apply to the `git branch` subcommand branch:

- `git branch` displays the branches of the local git repository.
- `git branch custom_python` creates a local branch `custom_python` in a local repository.
- `git branch -d custom_python` deletes the local branch `custom_python`.
- `git branch --help` displays the help for the `git branch` subcommand.

In the Python ecosystem, `pip` has the concept of subcommands, too. Some pip subcommands include `list`, `install`, `freeze`, or `uninstall`.

Windows

On Windows, the conventions regarding Python command line arguments are slightly different, in particular, those regarding [command line options](#). To validate this difference, take `tasklist`, which is a native Windows executable that displays a list of the currently running processes. It's similar to `ps` on Linux or macOS systems. Below is an example of how to execute `tasklist` in a command prompt on Windows:

Windows Console

```
C:/>tasklist /FI "IMAGENAME eq notepad.exe"

Image Name          PID Session Name      Session#    Mem Usage
=====
notepad.exe        13104 Console                 6      13,548 K
notepad.exe        6584 Console                 6      13,696 K
```

Note that the separator for an option is a forward slash (/) instead of a hyphen (-) like the conventions for Unix systems. For readability, there's a space between the program name, `tasklist`, and the option `/FI`, but it's just as correct to type `tasklist/FI`.

The particular example above executes `tasklist` with a filter to only show the Notepad processes currently running. You can see that the system has two running instances of the Notepad process. Although it's not equivalent, this is similar to executing the following command in a terminal on a Unix-like system:

Shell

```
$ ps -ef | grep vi | grep -v grep
andre    2117      4  0 13:33 pts/1    00:00:00 vi .gitignore
andre    2163  2134  0 13:34 pts/3    00:00:00 vi main.c
```

The `ps` command above shows all the current running `vi` processes. The behavior is consistent with the [Unix Philosophy](#), as the output of `ps` is transformed by two `grep` filters. The first `grep` command selects all the occurrences of `vi`, and the second `grep` filters out the occurrence of `grep` itself.

With the spread of Unix tools making their appearance in the Windows ecosystem, non-Windows-specific conventions are also accepted on Windows.

Visuals

At the start of a Python process, Python command line arguments are split into two categories:

1. **Python options:** These influence the execution of the Python interpreter. For example, adding option `-O` is a means to optimize the execution of a Python program by removing `assert` and `__debug__` statements. There are other [Python options](#) available at the command line.
2. **Python program and its arguments:** Following the Python options (if there are any), you'll find the Python program, which is a file name that usually has the extension `.py`, and its arguments. By convention, those can also be composed of options and arguments.

Take the following command that's intended to execute the program `main.py`, which takes options and arguments. Note that, in this example, the Python interpreter also takes some options, which are `-B` and `-v`.

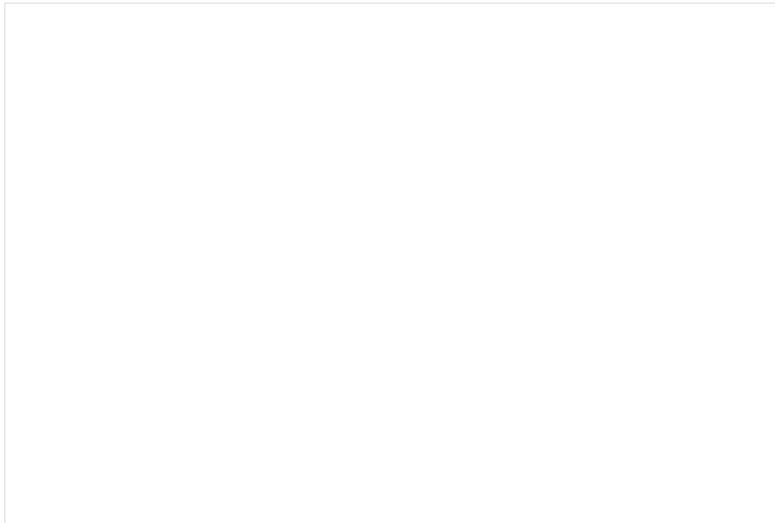
Shell

```
$ python -B -v main.py --verbose --debug un deux
```

In the command line above, the options are Python command line arguments and are organized as follows:

- **The option -B** tells Python not to write .pyc files on the import of source modules. For more details about .pyc files, check out the section [What Does a Compiler Do?](#) in [Your Guide to the CPython Source Code](#).
- **The option -v** stands for **verbose** and tells Python to trace all import statements.
- **The arguments passed to main.py** are fictitious and represent two long options (–verbose and –debug) and two arguments (un and deux).

This example of Python command line arguments can be illustrated graphically as follows:



Within the Python program `main.py`, you only have access to the Python command line arguments inserted by Python in `sys.argv`. The Python options may influence the behavior of the program but are not accessible in `main.py`.

A Few Methods for Parsing Python Command Line Arguments

Now you're going to explore a few approaches to apprehend options, option-arguments, and operands. This is done by **parsing** Python command line arguments. In this section, you'll see some concrete aspects of Python command line arguments and techniques to handle them. First, you'll see an example that introduces a straight approach relying on [list comprehensions](#) to collect and separate options from arguments. Then you will:

- **Use** regular expressions to extract elements of the command line
- **Learn** how to handle files passed at the command line
- **Apprehend** the standard input in a way that's compatible with the Unix tools
- **Differentiate** the regular output of the program from the errors
- **Implement** a custom parser to read Python command line arguments

This will serve as a preparation for options involving modules in the standard libraries or from external libraries that you'll learn about later in this tutorial.

For something uncomplicated, the following pattern, which doesn't enforce ordering and doesn't handle option-arguments, may be enough:

Python

```
# cul.py

import sys

opts = [opt for opt in sys.argv[1:] if opt.startswith("-")]
args = [arg for arg in sys.argv[1:] if not arg.startswith("-")]

if "-c" in opts:
    print(" ".join(arg.capitalize() for arg in args))
elif "-u" in opts:
    print(" ".join(arg.upper() for arg in args))
elif "-l" in opts:
    print(" ".join(arg.lower() for arg in args))
else:
    raise SystemExit(f"Usage: {sys.argv[0]} (-c | -u | -l) <arguments>...")
```

The intent of the program above is to modify the case of the Python command line arguments. Three options are available:

- **-c** to capitalize the arguments
- **-u** to convert the arguments to uppercase
- **-l** to convert the argument to lowercase

The code collects and separates the different argument types using [list comprehensions](#):

- **Line 5** collects all the **options** by filtering on any Python command line arguments starting with a hyphen (-).
- **Line 6** assembles the program **arguments** by filtering out the options.

When you execute the Python program above with a set of options and arguments, you get the following output:

Shell

```
$ python cul.py -c un deux trois
Un Deux Trois
```

This approach might suffice in many situations, but it would fail in the following cases:

- If the order is important, and in particular, if options should appear before the arguments
- If support for option-arguments is needed
- If some arguments are prefixed with a hyphen (-)

You can leverage other options before you resort to a library like `argparse` or `click`.

Regular Expressions

You can use a [regular expression](#) to enforce a certain order, specific options and option-arguments, or even the type of arguments. To illustrate the usage of a regular expression to parse Python command line arguments, you'll implement a Python version of `seq`, which is a program that prints a sequence of numbers. Following the docopt conventions, a specification for `seq.py` could be this:

```

Print integers from <first> to <last>, in steps of <increment>.

Usage:
  python seq.py --help
  python seq.py [-s SEPARATOR] <last>
  python seq.py [-s SEPARATOR] <first> <last>
  python seq.py [-s SEPARATOR] <first> <increment> <last>

Mandatory arguments to long options are mandatory for short options too.
  -s, --separator=STRING use STRING to separate numbers (default: \n)
      --help           display this help and exit

If <first> or <increment> are omitted, they default to 1. When <first> is
larger than <last>, <increment>, if not set, defaults to -1.
The sequence of numbers ends when the sum of the current number and
<increment> reaches the limit imposed by <last>.

```

First, look at a regular expression that's intended to capture the requirements above:

Python

```

1 args_pattern = re.compile(
2     r"""
3     ^
4     (
5         (--(?P<HELP>help).*)|
6         ((?:-s|--separator)\s(?P<SEP>.*?)\s)?
7         ((?P<OP1>-?\d+)(\s(?P<OP2>-?\d+))?( \s(?P<OP3>-?\d+))?
8     )
9     $
10    """,
11    re.VERBOSE,
12 )

```

To experiment with the regular expression above, you may use the snippet recorded on [Regular Expression 101](#). The regular expression captures and enforces a few aspects of the requirements given for seq. In particular, the command may take:

1. **A help option**, in short (-h) or long format (--help), captured as a [named group](#) called **HELP**
2. **A separator option**, -s or --separator, taking an optional argument, and captured as named group called **SEP**
3. **Up to three integer operands**, respectively captured as **OP1**, **OP2**, and **OP3**

For clarity, the pattern args_pattern above uses the flag [re.VERBOSE](#) on line 11. This allows you to spread the regular expression over a few lines to enhance readability. The pattern validates the following:

- **Argument order**: Options and arguments are expected to be laid out in a given order. For example, options are expected before the arguments.
- **Option values****: Only --help, -s, or --separator are expected as options.
- **Argument mutual exclusivity**: The option --help isn't compatible with other options or arguments.
- **Argument type**: Operands are expected to be positive or negative integers.

For the regular expression to be able to handle these things, it needs to see all Python command line arguments in one string. You can collect them using [str.join\(\)](#):

Python

```
arg_line = " ".join(sys.argv[1:])
```

This makes arg_line a string that includes all arguments, except the program name, separated by a space.

Given the pattern args_pattern above, you can extract the Python command line arguments with the following function:

Python

```

def parse(arg_line: str) -> Dict[str, str]:
    args: Dict[str, str] = {}
    if match_object := args_pattern.match(arg_line):
        args = {k: v for k, v in match_object.groupdict().items()}

```

```
args = [k, v for k, v in match_object.groupdict().items()
        if v is not None]
return args
```

The pattern is already handling the order of the arguments, mutual exclusivity between options and arguments, and the type of the arguments. `parse()` is applying `re.match()` to the argument line to extract the proper values and store the data in a dictionary.

The [dictionary](#) includes the names of each group as keys and their respective values. For example, if the `arg_line` value is `--help`, then the dictionary is `{'HELP': 'help'}`. If `arg_line` is `-s T 10`, then the dictionary becomes `{'SEP': 'T', 'OP1': '10'}`. You can expand the code block below to see an implementation of `seq` with regular expressions.

At this point, you know a few ways to extract options and arguments from the command line. So far, the Python command line arguments were only strings or integers. Next, you'll learn how to handle files passed as arguments.

File Handling

It's time now to experiment with Python command line arguments that are expected to be [file names](#). Modify `sha1sum.py` to handle one or more files as arguments. You'll end up with a downgraded version of the original `sha1sum` utility, which takes one or more files as arguments and displays the hexadecimal SHA1 hash for each file, followed by the name of the file:

Python

```
# sha1sum_file.py

import hashlib
import sys

def sha1sum(filename: str) -> str:
    hash = hashlib.sha1()
    with open(filename, mode="rb") as f:
        hash.update(f.read())
    return hash.hexdigest()

for arg in sys.argv[1:]:
    print(f"{sha1sum(arg)} {arg}")
```

`sha1sum()` is applied to the data read from each file that you passed at the command line, rather than the string itself. Take note that `m.update()` takes a [bytes-like object](#) as an argument and that the result of invoking `read()` after opening a file with the mode `rb` will return a [bytes object](#). For more information about handling file content, check out [Reading and Writing Files in Python](#), and in particular, the section [Working With Bytes](#).

The evolution of `sha1sum_file.py` from handling strings at the command line to manipulating the content of files is getting you closer to the original implementation of `sha1sum`:

Shell

```
$ sha1sum main main.c
9a6f82c245f5980082dbf6faac47e5085083c07d  main
125a0f900ff6f164752600550879cbfab098bc3  main.c
```

The execution of the Python program with the same Python command line arguments gives this:

Shell

```
$ python sha1sum_file.py main main.c
9a6f82c245f5980082dbf6faac47e5085083c07d  main
125a0f900ff6f164752600550879cbfab098bc3  main.c
```

Because you interact with the shell interpreter or the Windows command prompt, you also get the benefit of the wildcard expansion provided by the shell. To prove this, you can reuse `main.py`, which displays each argument with

the argument number and its value:

Shell

```
$ python main.py main.*  
Arguments count: 5  
Argument 0: main.py  
Argument 1: main.c  
Argument 2: main.exe  
Argument 3: main.obj  
Argument 4: main.py
```

You can see that the shell automatically performs wildcard expansion so that any file with a base name matching `main`, regardless of the extension, is part of `sys.argv`.

The wildcard expansion isn't available on Windows. To obtain the same behavior, you need to implement it in your code. To refactor `main.py` to work with wildcard expansion, you can use [glob](#). The following example works on Windows and, though it isn't as concise as the original `main.py`, the same code behaves similarly across platforms:

Python

```
1 # main_win.py  
2  
3 import sys  
4 import glob  
5 import itertools  
6 from typing import List  
7  
8 def expand_args(args: List[str]) -> List[str]:  
9     arguments = args[:1]  
10    glob_args = [glob.glob(arg) for arg in args[1:]]  
11    arguments += itertools.chain.from_iterable(glob_args)  
12    return arguments  
13  
14 if __name__ == "__main__":  
15    args = expand_args(sys.argv)  
16    print(f"Arguments count: {len(args)}")  
17    for i, arg in enumerate(args):  
18        print(f"Argument {i:>6}: {arg}")
```

In `main_win.py`, `expand_args` relies on [glob.glob\(\)](#) to process the shell-style wildcards. You can verify the result on Windows and any other operating system:

Windows Console

```
C:/>python main_win.py main.*  
Arguments count: 5  
Argument 0: main_win.py  
Argument 1: main.c  
Argument 2: main.exe  
Argument 3: main.obj  
Argument 4: main.py
```

This addresses the problem of handling files using wildcards like the asterisk (*) or question mark (?), but how about `stdin`?

If you don't pass any parameter to the original `sha1sum` utility, then it expects to read data from the **standard input**. This is the text you enter at the terminal that ends when you type `^Ctrl + D` on Unix-like systems or `^Ctrl + Z` on Windows. These control sequences send an end of file (EOF) to the terminal, which stops reading from `stdin` and returns the data that was entered.

In the next section, you'll add to your code the ability to read from the standard input stream.

Standard Input

When you modify the previous Python implementation of `sha1sum` to handle the standard input using [sys.stdin](#), you'll get closer to the original `sha1sum`:

Python

```
# sha1sum_stdin.py

from typing import List
import hashlib
import pathlib
import sys

def process_file(filename: str) -> bytes:
    return pathlib.Path(filename).read_bytes()

def process_stdin() -> bytes:
    return bytes("".join(sys.stdin), "utf-8")

def sha1sum(data: bytes) -> str:
    sha1_hash = hashlib.sha1()
    sha1_hash.update(data)
    return sha1_hash.hexdigest()

def output_sha1sum(data: bytes, filename: str = "-") -> None:
    print(f"{sha1sum(data)} {filename}")

def main(args: List[str]) -> None:
    if not args:
        args = ["-"]
    for arg in args:
        if arg == "-":
            output_sha1sum(process_stdin(), "-")
        else:
            output_sha1sum(process_file(arg), arg)

if __name__ == "__main__":
    main(sys.argv[1:])
```

Two conventions are applied to this new sha1sum version:

1. Without any arguments, the program expects the data to be provided in the standard input, `sys.stdin`, which is a readable file object.
2. When a hyphen (-) is provided as a file argument at the command line, the program interprets it as reading the file from the standard input.

Try this new script without any arguments. Enter the first aphorism of [The Zen of Python](#), then complete the entry with the keyboard shortcut `^Ctrl + D` on Unix-like systems or `^Ctrl + Z` on Windows:

Shell

```
$ python sha1sum_stdin.py
Beautiful is better than ugly.
ae5705a3efd4488dfc2b4b80df85f60c67d998c4 -
```

You can also include one of the arguments as `stdin` mixed with the other file arguments like so:

Shell

```
$ python sha1sum_stdin.py main.py - main.c
d84372fc77a90336b6bb7c5e959bcb1b24c608b4 main.py
Beautiful is better than ugly.
ae5705a3efd4488dfc2b4b80df85f60c67d998c4 -
125a0ff900ff6f164752600550879cbfab098bc3 main.c
```

Another approach on Unix-like systems is to provide `/dev/stdin` instead of - to handle the standard input:

Shell

```
$ python sha1sum_stdin.py main.py /dev/stdin main.c
d84372fc77a90336b6bb7c5e959bcb1b24c608b4 main.py
Beautiful is better than ugly.
ae5705a3efd4488dfc2b4b80df85f60c67d998c4 /dev/stdin
125a0ff900ff6f164752600550879cbfab098bc3 main.c
```

On Windows there's no equivalent to `/dev/stdin`, so using `-` as a file argument works as expected.

The script `sha1sum_stdin.py` isn't covering all necessary error handling, but you'll cover some of the missing features [later in this tutorial](#).

Standard Output and Standard Error

Command line processing may have a direct relationship with `stdin` to respect the conventions detailed in the previous section. The standard output, although not immediately relevant, is still a concern if you want to adhere to the [Unix Philosophy](#). To allow small programs to be combined, you may have to take into account the three standard streams:

1. `stdin`
2. `stdout`
3. `stderr`

The output of a program becomes the input of another one, allowing you to chain small utilities. For example, if you wanted to sort the aphorisms of the Zen of Python, then you could execute the following:

Shell

```
$ python -c "import this" | sort
Although never is often better than *right* now.
Although practicality beats purity.
Although that way may not be obvious at first unless you're Dutch.
...
```

The output above is truncated for better readability. Now imagine that you have a program that outputs the same data but also prints some debugging information:

Python

```
# zen_sort_debug.py

print("DEBUG >>> About to print the Zen of Python")
import this
print("DEBUG >>> Done printing the Zen of Python")
```

Executing the Python script above gives:

Shell

```
$ python zen_sort_debug.py
DEBUG >>> About to print the Zen of Python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
...
DEBUG >>> Done printing the Zen of Python
```

The ellipsis (...) indicates that the output was truncated to improve readability.

Now, if you want to sort the list of aphorisms, then execute the command as follows:

Shell

```
$ python zen_sort_debug.py | sort
Although never is often better than *right* now.
```

```

Although practicality beats purity.
Although that way may not be obvious at first unless you're Dutch.
Beautiful is better than ugly.
Complex is better than complicated.
DEBUG >>> About to print the Zen of Python
DEBUG >>> Done printing the Zen of Python
Errors should never pass silently.
...

```

You may realize that you didn't intend to have the debug output as the input of the sort command. To address this issue, you want to send traces to the standard errors stream, stderr, instead:

Python

```

# zen_sort_stderr.py
import sys

print("DEBUG >>> About to print the Zen of Python", file=sys.stderr)
import this
print("DEBUG >>> Done printing the Zen of Python", file=sys.stderr)

```

Execute zen_sort_stderr.py to observe the following:

Shell

```

$ python zen_sort_stderr.py | sort
DEBUG >>> About to print the Zen of Python
DEBUG >>> Done printing the Zen of Python

Although never is often better than *right* now.
Although practicality beats purity.
Although that way may not be obvious at first unless you're Dutch
...

```

Now, the traces are displayed to the terminal, but they aren't used as input for the sort command.

Custom Parsers

You can implement seq by relying on a regular expression if the arguments aren't too complex. Nevertheless, the regex pattern may quickly render the maintenance of the script difficult. Before you try getting help from specific libraries, another approach is to create a **custom parser**. The parser is a loop that fetches each argument one after another and applies a custom logic based on the semantics of your program.

A possible implementation for processing the arguments of seq_parse.py could be as follows:

Python

```

1 def parse(args: List[str]) -> Tuple[str, List[int]]:
2     arguments = collections.deque(args)
3     separator = "\n"
4     operands: List[int] = []
5     while arguments:
6         arg = arguments.popleft()
7         if not operands:
8             if arg == "--help":
9                 print(USAGE)
10                sys.exit(0)
11             if arg in ("s", "--separator"):
12                 separator = arguments.popleft()
13                 continue
14             try:
15                 operands.append(int(arg))
16             except ValueError:
17                 raise SystemExit(USAGE)
18             if len(operands) > 3:
19                 raise SystemExit(USAGE)
20
21     return separator, operands

```

parse() is given the list of arguments without the Python file name and uses `collections.deque\(\)` to get the benefit of `.popleft\(\)`, which removes the elements from the left of the collection. As the items of the arguments list

unfold, you apply the logic that's expected for your program. In `parse()` you can observe the following:

- The `while` loop is at the core of the function, and terminates when there are no more arguments to parse, when the help is invoked, or when an error occurs.
- If the `separator` option is detected, then the next argument is expected to be the separator.
- `operands` stores the integers that are used to calculate the sequence. There should be at least one operand and at most three.

A full version of the code for `parse()` is available below:

Click to expand the full example.

Show/Hide

This manual approach of parsing the Python command line arguments may be sufficient for a simple set of arguments. However, it becomes quickly error-prone when complexity increases due to the following:

- **A large number** of arguments
- **Complexity and interdependency** between arguments
- **Validation** to perform against the arguments

The custom approach isn't reusable and requires reinventing the wheel in each program. By the end of this tutorial, you'll have improved on this hand-crafted solution and learned a few better methods.

A Few Methods for Validating Python Command Line Arguments

You've already performed validation for Python command line arguments in a few examples like `seq_regex.py` and `seq_parse.py`. In the first example, you used a regular expression, and in the second example, a custom parser.

Both of these examples took the same aspects into account. They considered the expected `options` as short-form (`-s`) or long-form (`--separator`). They considered the `order` of the arguments so that options would not be placed after `operands`. Finally, they considered the type, integer for the operands, and the number of arguments, from one to three arguments.

Type Validation With Python Data Classes

The following is a proof of concept that attempts to validate the type of the arguments passed at the command line. In the following example, you validate the number of arguments and their respective type:

Python

```
# val_type_dc.py

import dataclasses
import sys
from typing import List, Any

USAGE = f"Usage: python {sys.argv[0]} [--help] | firstname lastname age]"

@dataclasses.dataclass
class Arguments:
    firstname: str
    lastname: str
    age: int = 0

def check_type(obj):
    for field in dataclasses.fields(obj):
        value = getattr(obj, field.name)
        print(
            f"Value: {value}, "
            f"Expected type {field.type} for {field.name}, "
            f"got {type(value)}")
    if type(value) != field.type:
        print("Type Error")
    else:
        print("Type Ok")
```

```

def validate(args: List[str]):
    # If passed to the command line, need to convert
    # the optional 3rd argument from string to int
    if len(args) > 2 and args[2].isdigit():
        args[2] = int(args[2])
    try:
        arguments = Arguments(*args)
    except TypeError:
        raise SystemExit(USAGE)
    check_type(arguments)

def main() -> None:
    args = sys.argv[1:]
    if not args:
        raise SystemExit(USAGE)

    if args[0] == "--help":
        print(USAGE)
    else:
        validate(args)

if __name__ == "__main__":
    main()

```

Unless you pass the --help option at the command line, this script expects two or three arguments:

1. **A mandatory string:** firstname
2. **A mandatory string:** lastname
3. **An optional integer:** age

Because all the items in sys.argv are strings, you need to convert the optional third argument to an integer if it's composed of digits. `str.isdigit()` validates if all the characters in a string are digits. In addition, by constructing the **data class** Arguments with the values of the converted arguments, you obtain two validations:

1. **If the number of arguments** doesn't correspond to the number of mandatory fields expected by Arguments, then you get an error. This is a minimum of two and a maximum of three fields.
2. **If the types after conversion** aren't matching the types defined in the Arguments data class definition, then you get an error.

You can see this in action with the following execution:

Shell

```

$ python val_type_dc.py Guido "Van Rossum" 25
Value: Guido, Expected type <class 'str'> for firstname, got <class 'str'>
Type Ok
Value: Van Rossum, Expected type <class 'str'> for lastname, got <class 'str'>
Type Ok
Value: 25, Expected type <class 'int'> for age, got <class 'int'>
Type Ok

```

In the execution above, the number of arguments is correct and the type of each argument is also correct.

Now, execute the same command but omit the third argument:

Shell

```

$ python val_type_dc.py Guido "Van Rossum"
Value: Guido, Expected type <class 'str'> for firstname, got <class 'str'>
Type Ok
Value: Van Rossum, Expected type <class 'str'> for lastname, got <class 'str'>
Type Ok
Value: 0, Expected type <class 'int'> for age, got <class 'int'>
Type Ok

```

The result is also successful because the field age is defined with a **default value**, 0, so the data class Arguments doesn't require it.

On the contrary, if the third argument isn't of the proper type—say, a string instead of integer—then you get an error:

Shell

```
python val_type_dc.py Guido Van Rossum
Value: Guido, Expected type <class 'str'> for firstname, got <class 'str'>
Type Ok
Value: Van, Expected type <class 'str'> for lastname, got <class 'str'>
Type Ok
Value: Rossum, Expected type <class 'int'> for age, got <class 'str'>
Type Error
```

The expected value Van Rossum, isn't surrounded by quotes, so it's split. The second word of the last name, Rossum, is a string that's handled as the age, which is expected to be an int. The validation fails.

Note: For more details about the usage of data classes in Python, check out [The Ultimate Guide to Data Classes in Python 3.7](#).

Similarly, you could also use a [NamedTuple](#) to achieve a similar validation. You'd replace the data class with a class deriving from `NamedTuple`, and `check_type()` would change as follows:

Python

```
from typing import NamedTuple

class Arguments(NamedTuple):
    firstname: str
    lastname: str
    age: int = 0

def check_type(obj):
    for attr, value in obj._asdict().items():
        print(
            f"Value: {value}, "
            f"Expected type {obj.__annotations__[attr]} for {attr}, "
            f"got {type(value)}"
        )
        if type(value) != obj.__annotations__[attr]:
            print("Type Error")
        else:
            print("Type Ok")
```

A `NamedTuple` exposes functions like `_asdict` that transform the object into a dictionary that can be used for data lookup. It also exposes attributes like `__annotations__`, which is a dictionary storing types for each field, and For more on `__annotations__`, check out [Python Type Checking \(Guide\)](#).

As highlighted in [Python Type Checking \(Guide\)](#), you could also leverage existing packages like [Enforce](#), [Pydantic](#), and [Pytypes](#) for advanced validation.

Custom Validation

Not unlike what you've already explored [earlier](#), detailed validation may require some custom approaches. For example, if you attempt to execute `sha1sum_stdin.py` with an incorrect file name as an argument, then you get the following:

Shell

```
$ python sha1sum_stdin.py bad_file.txt
```

```

Traceback (most recent call last):
  File "sha1sum_stdin.py", line 32, in <module>
    main(sys.argv[1:])
  File "sha1sum_stdin.py", line 29, in main
    output_sha1sum(process_file(arg), arg)
  File "sha1sum_stdin.py", line 9, in process_file
    return pathlib.Path(filename).read_bytes()
  File "/usr/lib/python3.8/pathlib.py", line 1222, in read_bytes
    with self.open(mode='rb') as f:
  File "/usr/lib/python3.8/pathlib.py", line 1215, in open
    return io.open(self, mode, buffering, encoding, errors, newline,
  File "/usr/lib/python3.8/pathlib.py", line 1071, in _opener
    return self._accessor.open(self, flags, mode)
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file.txt'

```

bad_file.txt doesn't exist, but the program attempts to read it.

Revisit main() in sha1sum_stdin.py to handle non-existing files passed at the command line:

Python

```

1 def main(args):
2     if not args:
3         output_sha1sum(process_stdin())
4     for arg in args:
5         if arg == "-":
6             output_sha1sum(process_stdin(), "-")
7             continue
8         try:
9             output_sha1sum(process_file(arg), arg)
10        except FileNotFoundError as err:
11            print(f"{sys.argv[0]}: {arg}: {err.strerror}", file=sys.stderr)

```

To see the complete example with this extra validation, expand the code block below:

Complete Source Code of sha1sum_val.py

Show/Hide

When you execute this modified script, you get this:

Shell

```

$ python sha1sum_val.py bad_file.txt
sha1sum_val.py: bad_file.txt: No such file or directory

```

Note that the error displayed to the terminal is written to stderr, so it doesn't interfere with the data expected by a command that would read the output of sha1sum_val.py:

Shell

```

$ python sha1sum_val.py bad_file.txt main.py | cut -d " " -f 1
sha1sum_val.py: bad_file.txt: No such file or directory
d84372fc77a90336b6bb7c5e959bcb1b24c608b4

```

This command pipes the output of sha1sum_val.py to [cut](#) to only include the first field. You can see that cut ignores the error message because it only receives the data sent to stdout.

The Python Standard Library

Despite the different approaches you took to process Python command line arguments, any complex program might be better off **leveraging existing libraries** to handle the heavy lifting required by sophisticated command line interfaces. As of Python 3.7, there are three command line parsers in the standard library:

1. [argparse](#)
2. [getopt](#)
3. [optparse](#)

The recommended module to use from the standard library is argparse. The standard library also exposes optparse but it's officially deprecated and only mentioned here for your information. It was superseded by argparse in Python 3.2 and you won't see it discussed in this tutorial.

argparse

You're going to revisit sha1sum_val.py, the most recent clone of sha1sum, to introduce the benefits of argparse. To this effect, you'll modify `main()` and add `init_argparse` to instantiate `argparse.ArgumentParser`:

Python

```
1 import argparse
2
3 def init_argparse() -> argparse.ArgumentParser:
4     parser = argparse.ArgumentParser(
5         usage=f"%(prog)s [OPTION] [FILE]...",
6         description="Print or check SHA1 (160-bit) checksums."
7     )
8     parser.add_argument(
9         "-v", "--version", action="version",
10        version = f"{parser.prog} version 1.0.0"
11    )
12    parser.add_argument('files', nargs='*')
13    return parser
14
15 def main() -> None:
16     parser = init_argparse()
17     args = parser.parse_args()
18     if not args.files:
19         output_sha1sum(process_stdin())
20     for file in args.files:
21         if file == "-":
22             output_sha1sum(process_stdin(), "-")
23             continue
24         try:
25             output_sha1sum(process_file(file), file)
26         except (FileNotFoundException, IsADirectoryError) as err:
27             print(f"[{sys.argv[0]}: {file}: {err.strerror}]", file=sys.stderr)
```

For the cost of a few more lines compared to the previous implementation, you get a clean approach to add `--help` and `--version` options that didn't exist before. The expected arguments (the files to be processed) are all available in field `files` of object `argparse.Namespace`. This object is populated on line 17 by calling `parse_args()`.

To look at the full script with the modifications described above, expand the code block below:

Complete Source Code of sha1sum_argparse.py

Show/Hide

To illustrate the immediate benefit you obtain by introducing argparse in this program, execute the following:

Shell

```
$ python sha1sum_argparse.py --help
usage: sha1sum_argparse.py [OPTION] [FILE]...
Print or check SHA1 (160-bit) checksums
```

```
PRIME OR CHECK SHA1 (100-BIT) CHECKSUMS.
```

```
positional arguments:  
  files  
  
optional arguments:  
  -h, --help      show this help message and exit  
  -v, --version   show program's version number and exit
```

To delve into the details of argparse, check out [How to Build Command Line Interfaces in Python With argparse](#).

getopt

getopt finds its origins in the [getopt](#) C function. It facilitates parsing the command line and handling options, option arguments, and arguments. Revisit parse from seq_parse.py to use getopt:

Python

```
def parse():
    options, arguments = getopt.getopt(
        sys.argv[1:],                         # Arguments
        'vhs:',                                # Short option definitions
        ["version", "help", "separator="])       # Long option definitions
    separator = "\n"
    for o, a in options:
        if o in ("-v", "--version"):
            print(VERSION)
            sys.exit()
        if o in ("-h", "--help"):
            print(USAGE)
            sys.exit()
        if o in ("-s", "--separator"):
            separator = a
    if not arguments or len(arguments) > 3:
        raise SystemExit(USAGE)
    try:
        operands = [int(arg) for arg in arguments]
    except ValueError:
        raise SystemExit(USAGE)
    return separator, operands
```

[getopt.getopt\(\)](#) takes the following arguments:

1. The usual arguments list minus the script name, `sys.argv[1:]`
2. A string defining the short options
3. A list of strings for the long options

Note that a short option followed by a colon (:) expects an option argument, and that a long option trailed with an equals sign (=) expects an option argument.

The remaining code of seq_getopt.py is the same as seq_parse.py and is available in the collapsed code block below:

Complete Source Code of seq_getopt.py

Show/Hide

Next, you'll take a look at some external packages that will help you parse Python command line arguments.

A Few External Python Packages

Building upon the existing conventions you saw in this tutorial, there are a few libraries available on the [Python Package Index \(PyPI\)](#) that take many more steps to facilitate the implementation and maintenance of command line interfaces.

The following sections offer a glance at [Click](#) and [Python Prompt Toolkit](#). You'll only be exposed to very limited capabilities of these packages, as they both would require a full tutorial—if not a whole series—to do them justice!

Click

As of this writing, **Click** is perhaps the most advanced library to build a sophisticated command line interface for a Python program. It's used by several Python products, most notably [Flask](#) and [Black](#). Before you try the following example, you need to install Click in either a [Python virtual environment](#) or your local environment. If you're not familiar with the concept of virtual environments, then check out [Python Virtual Environments: A Primer](#).

To install Click, proceed as follows:

Shell

```
$ python -m pip install click
```

So, how could Click help you handle the Python command line arguments? Here's a variation of the seq program using Click:

Python

```
# seq_click.py

import click

@click.command(context_settings=dict(ignore_unknown_options=True))
@click.option("--separator", "-s",
              default="\n",
              help="Text used to separate numbers (default: \\n)")
@click.version_option(version="1.0.0")
@click.argument("operands", type=click.INT, nargs=-1)
def seq(operands, separator) -> str:
    first, increment, last = 1, 1, 1
    if len(operands) == 1:
        last = operands[0]
    elif len(operands) == 2:
        first, last = operands
        if first > last:
            increment = -1
    elif len(operands) == 3:
        first, increment, last = operands
    else:
        raise click.BadParameter("Invalid number of arguments")
    last = last - 1 if first > last else last + 1
    print(separator.join(str(i) for i in range(first, last, increment)))

if __name__ == "__main__":
    seq()
```

Setting `ignore_unknown_options` to True ensures that Click doesn't parse negative arguments as options. Negative integers are valid seq arguments.

As you may have observed, you get a lot for free! A few well-carved [decorators](#) are sufficient to bury the boilerplate code, allowing you to focus on the main code, which is the content of `seq()` in this example.

Note: For more about Python decorators, check out [Primer on Python Decorators](#).

The only import remaining is `click`. The declarative approach of decorating the main command, `seq()`, eliminates repetitive code that's otherwise necessary. This could be any of the following:

- **Defining** a help or usage procedure
- **Handling** the version of the program
- **Capturing** and **setting up** default values for options
- **Validating** arguments including the type

...arguments, including the type

The new seq implementation barely scratches the surface. Click offers many niceties that will help you craft a very professional command line interface:

- Output coloring
- Prompt for omitted arguments
- Commands and sub-commands
- Argument type validation
- Callback on options and arguments
- File path validation
- Progress bar

There are many other features as well. Check out [Writing Python Command-Line Tools With Click](#) to see more concrete examples based on Click.

Python Prompt Toolkit

There are other popular Python packages that are handling the command line interface problem, like [docopt for Python](#). So, you may find the choice of the [Prompt Toolkit](#) a bit counterintuitive.

The **Python Prompt Toolkit** provides features that may make your command line application drift away from the Unix philosophy. However, it helps to bridge the gap between an arcane command line interface and a full-fledged [graphical user interface](#). In other words, it may help to make your tools and programs more user-friendly.

You can use this tool in addition to processing Python command line arguments as in the previous examples, but this gives you a path to a UI-like approach without you having to depend on a full [Python UI toolkit](#). To use `prompt_toolkit`, you need to install it with pip:

Shell

```
$ python -m pip install prompt_toolkit
```

You may find the next example a bit contrived, but the intent is to spur ideas and move you slightly away from more rigorous aspects of the command line with respect to the conventions you've seen in this tutorial.

As you've already seen the core logic of this example, the code snippet below only presents the code that significantly deviates from the previous examples:

Python

```
def error_dlg():
    message_dialog(
        title="Error",
        text="Ensure that you enter a number",
    ).run()

def seq_dlg():
    labels = ["FIRST", "INCREMENT", "LAST"]
    operands = []
    while True:
        n = input_dialog(
            title="Sequence",
            text=f"Enter argument {labels[len(operands)]}:",
        ).run()
        if n is None:
            break
        if n.isdigit():
            operands.append(int(n))
        else:
            error_dlg()
        if len(operands) == 3:
            break

    if operands:
        seq(operands)
    else:
        print("Bye")

actions = {"SEQUENCE": seq_dlg, "HELP": help, "VERSION": version}

def main():
    result = button_dialog(
        title="Sequence",
        text="Select an action:",
        buttons=[
            ("Sequence", "SEQUENCE"),
            ("Help", "HELP"),
            ("Version", "VERSION"),
        ],
    ).run()
    actions.get(result, lambda: print("Unexpected action"))()
```

The code above involves ways to interact and possibly guide users to enter the expected input, and to validate the input interactively using three dialog boxes:

1. button_dialog
2. message_dialog
3. input_dialog

The Python Prompt Toolkit exposes many other features intended to improve interaction with users. The call to the handler in `main()` is triggered by calling a function stored in a dictionary. Check out [Emulating switch/case Statements in Python](#) if you've never encountered this Python idiom before.

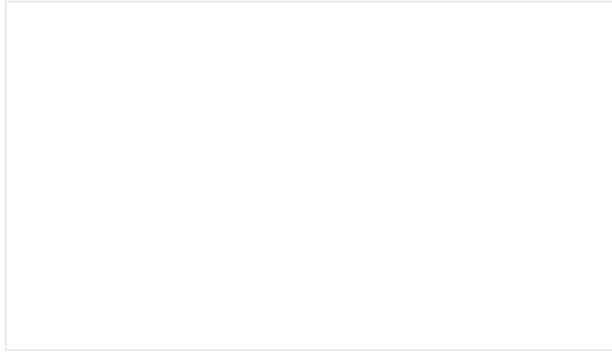
You can see the full example of the program using `prompt_toolkit` by expanding the code block below:

Complete Source Code for seq_prompt.py

Show/Hide

When you execute the code above, you're greeted with a dialog prompting you for action. Then, if you choose the action `Sequence`, another dialog box is displayed. After collecting all the necessary data, options, or arguments, the

dialog box disappears, and the result is printed at the command line, as in the previous examples:



As the command line evolves and you can see some attempts to interact with users more creatively, other packages like [PyInquirer](#) also allow you to capitalize on a very interactive approach.

To further explore the world of the **Text-Based User Interface (TUI)**, check out [Building Console User Interfaces](#) and the [Third Party section](#) in [Your Guide to the Python Print Function](#).

If you're interested in researching solutions that rely exclusively on the graphical user interface, then you may consider checking out the following resources:

- [How to Build a Python GUI Application With wxPython](#)
- [Python and PyQt: Building a GUI Desktop Calculator](#)
- [Build a Mobile Application With the Kivy Python Framework](#)

Conclusion

In this tutorial, you've navigated many different aspects of Python command line arguments. You should feel prepared to apply the following skills to your code:

- The **conventions and pseudo-standards** of Python command line arguments
- The **origins** of sys.argv in Python
- The **usage** of sys.argv to provide flexibility in running your Python programs
- The **Python standard libraries** like argparse or getopt that abstract command line processing
- The **powerful Python packages** like click and python_toolkit to further improve the usability of your programs

Whether you're running a small script or a complex text-based application, when you expose a **command line interface** you'll significantly improve the user experience of your Python software. In fact, you're probably one of those users!

Next time you use your application, you'll appreciate the documentation you supplied with the --help option or the fact that you can pass options and arguments instead of modifying the source code to supply different data.

Additional Resources

To gain further insights about Python command line arguments and their many facets, you may want to check out the following resources:

- [Comparing Python Command-Line Parsing Libraries – Argparse, Docopt, and Click](#)
- [Python, Ruby, and Golang: A Command-Line Application Comparison](#)

You may also want to try other Python libraries that target the same problems while providing you with different solutions:

- [Typer](#)
- [Plac](#)
- [Cliff](#)

- [Cement](#)
- [Python Fire](#)

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Command Line Interfaces in Python](#)

About Andre Burgaud

Andre is a seasoned software engineer passionate about technology and programming languages, in particular, Python.

[» More about Andre](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

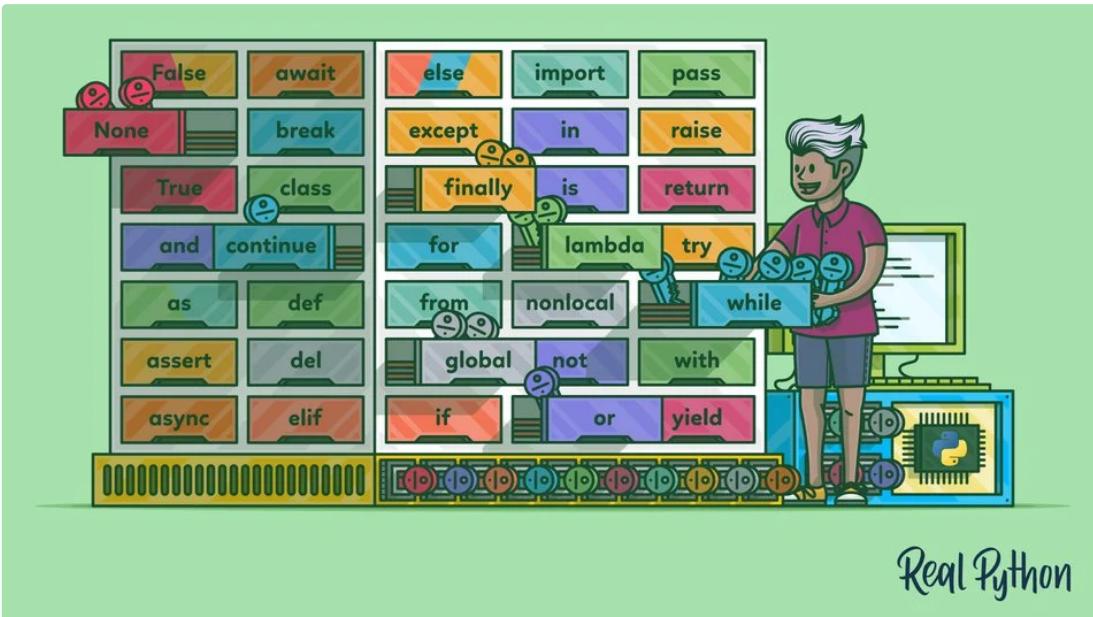
[Jaya](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [tools](#)

Recommended Video Course: [Command Line Interfaces in Python](#)



Python Keywords: An Introduction

by [Chad Hansen](#) ⌂ Jun 15, 2020 [1 Comment](#) [basics](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Python Keywords](#)
- [How to Identify Python Keywords](#)
 - [Use an IDE With Syntax Highlighting](#)
 - [Use Code in a REPL to Check Keywords](#)
 - [Look for a SyntaxError](#)
- [Python Keywords and Their Usage](#)
 - [Value Keywords: True, False, None](#)
 - [Operator Keywords: and, or, not, in, is](#)
 - [Control Flow Keywords: if, elif, else](#)
 - [Iteration Keywords: for, while, break, continue, else](#)
 - [Structure Keywords: def, class, with, as, pass, lambda](#)
 - [Returning Keywords: return, yield](#)
 - [Import Keywords: import, from, as](#)
 - [Exception-Handling Keywords: try, except, raise, finally, else, assert](#)
 - [Asynchronous Programming Keywords: async, await](#)
 - [Variable Handling Keywords: del, global, nonlocal](#)
- [Deprecated Python Keywords](#)
 - [The Former print Keyword](#)
 - [The Former exec Keyword](#)
- [Conclusion](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

Every programming language has special reserved words, or **keywords**, that have specific meanings and restrictions around how they should be used. Python is no different. Python keywords are the fundamental building blocks of any Python program.

In this article, you'll find a basic introduction to all Python keywords along with other resources that will be helpful for learning more about each keyword.

By the end of this article, you'll be able to:

- **Identify** Python keywords
- **Understand** what each keyword is used for
- **Work** with keywords programmatically using the keyword module

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Python Keywords

Python keywords are special reserved words that have specific meanings and purposes and can't be used for anything but those specific purposes. These keywords are always available—you'll never have to import them into your code.

Python keywords are different from Python's [built-in functions and types](#). The built-in functions and types are also always available, but they aren't as restrictive as the keywords in their usage.

An example of something you *can't* do with Python keywords is assign something to them. If you try, then you'll get a `SyntaxError`. You won't get a `SyntaxError` if you try to assign something to a built-in function or type, but it still isn't a good idea. For a more in-depth explanation of ways keywords can be misused, check out [Invalid Syntax in Python: Common Reasons for SyntaxError](#).

As of Python 3.8, there are [thirty-five keywords](#) in Python. Here they are with links to the relevant sections throughout the rest of this article:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

You can use these links to jump to the keywords you'd like to read about, or you can continue reading for a guided tour.

Note: Two keywords have additional uses beyond their initial use cases. The `else` keyword is also [used with loops](#) as well as [with try and except](#). The `as` keyword is also used [with the with keyword](#).

How to Identify Python Keywords

The list of Python keywords has changed over time. For example, the `await` and `async` keywords weren't added until Python 3.7. Also, both `print` and `exec` were keywords in Python 2.7 but have been turned into built-in functions in Python 3+ and no longer appear in the list of keywords.

In the sections below, you'll learn several ways to know or find out which words are keywords in Python.

Use an IDE With Syntax Highlighting

There are a lot of [good Python IDEs](#) out there. All of them will highlight keywords to differentiate them from other words in your code. This will help you quickly identify Python keywords while you're programming so you don't use them incorrectly.

Use Code in a REPL to Check Keywords

In the [Python REPL](#), there are a number of ways you can identify valid Python keywords and learn more about them.

Note: Code examples in this article use Python 3.8 unless otherwise indicated.

You can get a list of available keywords by using `help()`:

```
Python >>>
>>> help("keywords")
Here is a list of the Python keywords. Enter any keyword to get more help.

False      class      from      or
None       continue   global    pass
True       def        if        raise
and        del        import   return
as         elif       in       try
assert    else       is        while
async     except    lambda   with
await     finally   nonlocal yield
break    for        not
```

Next, as indicated in the output above, you can use `help()` again by passing in the specific keyword that you need more information about. You can do this, for example, with the `pass` keyword:

```
Python >>>
>>> help("pass")
The "pass" statement
*****
pass_stmt ::= "pass"

"pass" is a null operation – when it is executed, nothing happens. It
is useful as a placeholder when a statement is required syntactically,
but no code needs to be executed, for example:

def f(arg): pass    # a function that does nothing (yet)

class C: pass      # a class with no methods (yet)
```

Python also provides a `keyword` module for working with Python keywords in a programmatic way. The `keyword` module in Python provides two helpful members for dealing with keywords:

1. `kwlist` provides a list of all the Python keywords for the version of Python you're running.
2. `iskeyword()` provides a handy way to determine if a string is also a keyword.

To get a list of all the keywords in the version of Python you're running, and to quickly determine how many keywords are defined, use `keyword.kwlist`:

```
Python >>>
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'async', ...
>>> len(keyword.kwlist)
35
```

If you need to know more about a keyword or need to work with keywords in a programmatic way, then Python provides this documentation and tooling for you.

Look for a SyntaxError

Finally, another indicator that a word you're using is actually a keyword is if you get a `SyntaxError` while trying to assign to it, name a function with it, or do something else that isn't allowed with it. This one is a little harder to spot, but it's a way that Python will let you know you're [using a keyword incorrectly](#).

Python Keywords and Their Usage

The sections below organize the Python keywords into groups based on their usage. For example, the first group is all the keywords that are used as values, and the second group is the keywords that are used as operators. These groupings will help you better understand how keywords are used and provide a nice way to organize the long list of Python keywords.

There are a few terms used in the sections below that may be new to you. They're defined here, and you should be aware of their meaning before proceeding:

- **Truthiness** refers to the Boolean evaluation of a value. The truthiness of a value indicates whether the value is **truthy** or **falsy**.
- **Truthy** means any value that evaluates to true in the Boolean context. To determine if a value is truthy, pass it as the argument to `bool()`. If it returns `True`, then the value is truthy. Examples of truthy values are non-empty strings, any numbers that aren't `0`, non-empty lists, and many more.
- **Falsy** means any value that evaluates to false in the Boolean context. To determine if a value is falsy, pass it as the argument to `bool()`. If it returns `False`, then the value is falsy. Examples of falsy values are `"", 0, [], {}, and set()`.

For more on these terms and concepts, check out [Operators and Expressions in Python](#).

Value Keywords: True, False, None

There are three Python keywords that are used as values. These values are [singleton](#) values that can be used over and over again and always reference the exact same object. You'll most likely see and use these values a lot.

The True and False Keywords

The `True` keyword is used as the Boolean true value in Python code. The Python keyword `False` is similar to the `True` keyword, but with the opposite Boolean value of false. In other programming languages, you'll see these keywords written in lowercase (`true` and `false`), but in Python they are always written in uppercase.

The Python keywords `True` and `False` can be assigned to variables and compared to directly:

```
Python >>>
>>> x = True
>>> x is True
True

>>> y = False
>>> y is False
True
```

Most values in Python will evaluate to `True` when passed to `bool()`. There are only a few values in Python that will evaluate to `False` when passed to `bool()`: `0`, `""`, `[]`, and `{}` to name a few. Passing a value to `bool()` indicates the value's truthiness, or the equivalent Boolean value. You can compare a value's truthiness to `True` or `False` by passing the value to `bool()`:

Python

>>>

```
>>> x = "this is a truthy value"
>>> x is True
False
>>> bool(x) is True
True

>>> y = "" # This is falsy
>>> y is False
False
>>> bool(y) is False
True
```

Notice that comparing a truthy value directly to `True` or `False` using `is` doesn't work. You should directly compare a value to `True` or `False` only if you want to know whether it is *actually* the values `True` or `False`.

When writing conditional statements that are based on the truthiness of a value, you should *not* compare directly to `True` or `False`. You can rely on Python to do the truthiness check in conditionals for you:

Python

>>>

```
>>> x = "this is a truthy value"
>>> if x is True: # Don't do this
...     print("x is True")
...
>>> if x: # Do this
...     print("x is truthy")
...
x is truthy
```

In Python, you generally don't need to convert values to be explicitly `True` or `False`. Python will implicitly determine the truthiness of the value for you.

The None Keyword

The Python keyword `None` represents no value. In other programming languages, `None` is represented as `null`, `nil`, `none`, `undef`, or `undefined`.

`None` is also the default value returned by a function if it doesn't have a `return` statement:

Python

>>>

```
>>> def func():
...     print("hello")
...
>>> x = func()
hello
>>> print(x)
None
```

To go more in depth on this very important and useful Python keyword, check out [Null in Python: Understanding Python's NoneType Object](#).

Operator Keywords: and, or, not, in, is

Several Python keywords are used as operators. In other programming languages, these operators use symbols like `&`, `|`, and `!`. The Python operators for these are all keywords:

Math Operator	Other Languages	Python Keyword
AND, \wedge	<code>&&</code>	<code>and</code>
OR, \vee	<code> </code>	<code>or</code>
NOT, \neg	<code>!</code>	<code>not</code>
CONTAINS, \in		<code>in</code>
IDENTITY	<code>==</code>	<code>is</code>

Python code was designed for readability. That's why many of the operators that use symbols in other programming languages are keywords in Python.

The and Keyword

The Python keyword `and` is used to determine if both the left and right operands are truthy or falsy. If both operands are truthy, then the result will be truthy. If one is falsy, then the result will be falsy:

Python

```
<expr1> and <expr2>
```

Note that the results of an `and` statement will not necessarily be `True` or `False`. This is because of the quirky behavior of `and`. Rather than evaluating the operands to their Boolean values, `and` simply returns `<expr1>` if it is falsy or else it returns `<expr2>`. The results of an `and` statement could be passed to `bool()` to get the explicit `True` or `False` value, or they could be used in a conditional `if` statement.

If you wanted to define an expression that did the same thing as an `and` expression, but without using the `and` keyword, then you could use the Python ternary operator:

Python

```
left if not left else right
```

The above statement will produce the same result as `left and right`.

Because `and` returns the first operand if it's falsy and otherwise returns the last operand, you can also use `and` in an assignment:

Python

```
x = y and z
```

If `y` is falsy, then this would result in `x` being assigned the value of `y`. Otherwise, `x` would be assigned the value of `z`. However, this makes for confusing code. A more verbose and clear alternative would be:

Python

```
x = y if not y else z
```

This code is longer, but it more clearly indicates what you're trying to accomplish.

The or Keyword

Python's `or` keyword is used to determine if at least one of the operands is truthy. An `or` statement returns the first operand if it is truthy and otherwise returns the second operand:

Python

```
<expr1> or <expr2>
```

Just like the `and` keyword, `or` doesn't convert its operands to their Boolean values. Instead, it relies on their truthiness to determine the results.

If you wanted to write something like an `or` expression without the use of `or`, then you could do so with a ternary expression:

Python

```
left if left else right
```

This expression will produce the same result as `left or right`. To take advantage of this behavior, you'll also sometimes see `or` used in assignments. This is generally discouraged in favor of a more explicit assignment.

For a more in-depth look at `or` and how to use it, check out [How to Use the Python `or` Operator](#).

The `not` Keyword

Python's `not` keyword is used to get the opposite Boolean value of a variable:

Python

>>>

```
>>> val = "" # Truthiness value is `False`
>>> not val
True

>>> val = 5 # Truthiness value is `True`
>>> not val
False
```

The `not` keyword is used in conditional statements or other Boolean expressions to *flip* the Boolean meaning or result. Unlike `and` and `or`, `not` will determine the explicit Boolean value, `True` or `False`, and then return the opposite.

If you wanted to get the same behavior without using `not`, then you could do so with the following ternary expression:

Python

```
True if bool(<expr>) is False else False
```

This statement would return the same result as `not <expr>`.

The `in` Keyword

Python's `in` keyword is a powerful containment check, or **membership operator**. Given an element to find and a container or sequence to search, `in` will return `True` or `False` indicating whether the element was found in the container:

Python

```
<element> in <container>
```

A good example of using the `in` keyword is checking for a specific letter in a string:

Python

>>>

```
>>> name = "Chad"
>>> "c" in name
False
>>> "C" in name
True
```

The `in` keyword works with all types of containers: lists, dicts, sets, strings, and anything else that defines `__contains__()` or can be iterated over.

The `is` Keyword

Python's `is` keyword is an identity check. This is different from the `==` operator, which checks for equality.

Sometimes two things can be considered equal but not be the exact same object in memory. The `is` keyword determines whether two objects are exactly the same object:

Python

```
<obj1> is <obj2>
```

This will return `True` if `<obj1>` is the exact same object in memory as `<obj2>`, or else it will return `False`.

Most of the time you'll see `is` used to check if an object is `None`. Since `None` is a singleton, only one instance of `None` that can exist, so all `None` values are the exact same object in memory.

If these concepts are new to you, then you can get a more in-depth explanation by checking out [Python '`!=`' Is Not '`is`' not: Comparing Objects in Python](#). For a deeper dive into how `is` works, check out [Operators and Expressions in Python](#).

Control Flow Keywords: `if`, `elif`, `else`

Three Python keywords are used for control flow: `if`, `elif`, and `else`. These Python keywords allow you to use conditional logic and execute code given certain conditions. These keywords are very common—they'll be used in almost every program you see or write in Python.

The `if` Keyword

The `if` keyword is used to start a [conditional statement](#). An `if` statement allows you to write a block of code that gets executed only if the expression after `if` is truthy.

The syntax for an `if` statement starts with the keyword `if` at the beginning of the line, followed by a valid expression that will be evaluated for its truthiness value:

Python

```
if <expr>:  
    <statements>
```

The `if` statement is a crucial component of most programs. For more information about the `if` statement, check out [Conditional Statements in Python](#).

Another use of the `if` keyword is as part of Python's [ternary operator](#):

Python

```
<var> = <expr1> if <expr2> else <expr3>
```

This is a one-line version of the `if...else` statement below:

Python

```
if <expr2>:  
    <var> = <expr1>  
else:  
    <var> = <expr3>
```

If your expressions are uncomplicated statements, then using the ternary expression provides a nice way to simplify your code a bit. Once the conditions get a little complex, it's often better to rely on the standard `if` statement.

The `elif` Keyword

The `elif` statement looks and functions like the `if` statement, with two major differences:

1. Using `elif` is only valid after an `if` statement or another `elif`.
2. You can use as many `elif` statements as you need.

In other programming languages, `elif` is either `else if` (two separate words) or `elseif` (both words mashed

together). When you see `elif` in Python, think `else if`:

Python

```
if <expr1>:  
    <statements>  
elif <expr2>:  
    <statements>  
elif <expr3>:  
    <statements>
```

Python doesn't have a [switch statement](#). One way to get the same functionality that other programming languages provide with switch statements is by using `if` and `elif`. For other ways of reproducing the switch statement in Python, check out [Emulating switch/case Statements in Python](#).

The `else` Keyword

The `else` statement, in conjunction with the Python keywords `if` and `elif`, denotes a block of code that should be executed only if the other conditional blocks, `if` and `elif`, are all falsy:

Python

```
if <expr>:  
    <statements>  
else:  
    <statements>
```

Notice that the `else` statement doesn't take a conditional expression. Knowledge of the [`elif` and `else` keywords](#) and their proper usage is critical for Python programmers. Together with `if`, they make up some of the most frequently used components in any Python program.

Iteration Keywords: `for`, `while`, `break`, `continue`, `else`

Looping and iteration are hugely important programming concepts. Several Python keywords are used to create and work with loops. These, like the Python keywords used for conditionals above, will be used and seen in just about every Python program you come across. Understanding them and their proper usage will help you improve as a Python programmer.

The `for` Keyword

The most common loop in Python is the `for` loop. It's constructed by combining the Python keywords `for` and `in` explained earlier. The basic syntax for a `for` loop is as follows:

Python

```
for <element> in <container>:  
    <statements>
```

A common example is looping over the numbers one through five and printing them to the screen:

Python

>>>

```
>>> for num in range(1, 6):  
...     print(num)  
...  
1  
2  
3  
4  
5
```

In other programming languages, the syntax for a `for` loop will look a little different. You'll often need to specify the variable, the condition for continuing, and the way to increment that variable (`for (int i = 0; i < 5; i++)`).

In Python, the `for` loop is like a [for-each loop](#) in other programming languages. Given the object to iterate over, it assigns the value of each iteration to the variable:

Python

>>>

```
>>> people = ["Kevin", "Creed", "Jim"]
>>> for person in people:
...     print(f"{person} was in The Office.")
...
Kevin was in The Office.
Creed was in The Office.
Jim was in The Office.
```

In this example, you start with the list (container) of people's names. The `for` loop starts with the `for` keyword at the beginning of the line, followed by the variable to assign each element of the list to, then the `in` keyword, and finally the container (`people`).

Python's `for` loop is another major ingredient in any Python program. To learn more about `for` loops, check out [Python “for” Loops \(Definite Iteration\)](#).

The `while` Keyword

Python's `while` loop uses the keyword `while` and works like a `while` loop in other programming languages. As long as the condition that follows the `while` keyword is truthy, the block following the `while` statement will continue to be executed over and over again:

Python

```
while <expr>:
    <statements>
```

Note: For the infinite loop example below, be prepared to use `^Ctrl + C` to stop the process if you decide to try it on your own machine.

The easiest way to specify an infinite loop in Python is to use the `while` keyword with an expression that is always truthy:

Python

>>>

```
>>> while True:
...     print("working...")
...
```

For more examples of infinite loops in action, check out [Socket Programming in Python \(Guide\)](#). To learn more about `while` loops, check out [Python “while” Loops \(Indefinite Iteration\)](#).

The `break` Keyword

If you need to exit a loop early, then you can use the `break` keyword. This keyword will work in both `for` and `while` loops:

Python

```
for <element> in <container>:
    if <expr>:
        break
```

An example of using the `break` keyword would be if you were summing the integers in a list of numbers and wanted to quit when the total went above a given value:

Python

>>>

```
>>> nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sum = 0
>>> for num in nums:
...     sum += num
...     if sum > 10:
...         break
```

```
...  
>>> sum  
15
```

Both the Python keywords `break` and `continue` can be useful tools when working with loops. For a deeper discussion of their uses, check out [Python “while” Loops \(Indefinite Iteration\)](#).

The `continue` Keyword

Python also has a `continue` keyword for when you want to skip to the next loop iteration. Like in most other programming languages, the `continue` keyword allows you to stop executing the current loop iteration and move on to the next iteration:

Python

```
for <element> in <container>:  
    if <expr>:  
        continue
```

The `continue` keyword also works in `while` loops. If the `continue` keyword is reached in a loop, then the current iteration is stopped, and the next iteration of the loop is started.

The `else` Keyword Used With Loops

In addition to using the `else` keyword with conditional `if` statements, you can also use it as part of a loop. When used with a loop, the `else` keyword specifies code that should be run if the loop exits normally, meaning `break` was not called to exit the loop early.

The syntax for using `else` with a `for` loop looks like the following:

Python

```
for <element> in <container>:  
    <statements>  
else:  
    <statements>
```

This is very similar to using `else` with an `if` statement. Using `else` with a `while` loop looks similar:

Python

```
while <expr>:  
    <statements>  
else:  
    <statements>
```

The Python standard documentation has a section on [using `break` and `else` with a `for` loop](#) that you should really check out. It uses a great example to illustrate the usefulness of the `else` block.

The task it shows is looping over the numbers two through nine to find the prime numbers. One way you could do this is with a standard `for` loop with a **flag variable**:

Python

```
>>> for n in range(2, 10):  
...     prime = True  
...     for x in range(2, n):  
...         if n % x == 0:  
...             prime = False  
...             print(f"{n} is not prime")  
...             break  
...     if prime:  
...         print(f"{n} is prime!")  
...  
2 is prime!  
3 is prime!  
4 is not prime  
5 is prime!
```

>>>

```
>>> if n % 2 == 0:  
6 is not prime  
7 is prime!  
8 is not prime  
9 is not prime
```

You can use the `prime` flag to indicate how the loop was exited. If it exited normally, then the `prime` flag stays `True`. If it exited with `break`, then the `prime` flag will be set to `False`. Once outside the inner `for` loop, you can check the flag to determine if `prime` is `True` and, if so, print that the number is prime.

The `else` block provides more straightforward syntax. If you find yourself having to set a flag in a loop, then consider the next example as a way to potentially simplify your code:

Python

>>>

```
>>> for n in range(2, 10):  
...     for x in range(2, n):  
...         if n % x == 0:  
...             print(f"{n} is not prime")  
...             break  
...     else:  
...         print(f"{n} is prime!")  
...  
2 is prime!  
3 is prime!  
4 is not prime  
5 is prime!  
6 is not prime  
7 is prime!  
8 is not prime  
9 is not prime
```

The only thing that you need to do to use the `else` block in this example is to remove the `prime` flag and replace the final `if` statement with the `else` block. This ends up producing the same result as the example before, only with clearer code.

Sometimes using an `else` keyword with a loop can seem a little strange, but once you understand that it allows you to avoid using flags in your loops, it can be a powerful tool.

Structure Keywords: `def`, `class`, `with`, `as`, `pass`, `lambda`

In order to [define functions](#) and classes or use [context managers](#), you'll need to use one of the Python keywords in this section. They're an essential part of the Python language, and understanding when to use them will help you become a better Python programmer.

The `def` Keyword

Python's keyword `def` is used to define a function or method of a class. This is equivalent to `function` in JavaScript and PHP. The basic syntax for defining a function with `def` looks like this:

Python

```
def <function>(<params>):  
    <body>
```

Functions and methods can be very helpful structures in any Python program. To learn more about defining them and all their ins and outs, check out [Defining Your Own Python Function](#).

The class Keyword

To define a class in Python, you use the `class` keyword. The general syntax for defining a class with `class` is as follows:

Python

```
class MyClass(<extends>):
    <body>
```

Classes are powerful tools in object-oriented programming, and you should know about them and how to define them. To learn more, check out [Object-Oriented Programming \(OOP\) in Python 3](#).

The with Keyword

Context managers are a really helpful structure in Python. Each context manager executes specific code before and after the statements you specify. To use one, you use the `with` keyword:

Python

```
with <context manager> as <var>:
    <statements>
```

Using `with` gives you a way to define code to be executed within the context manager's [scope](#). The most basic example of this is when you're working with [file I/O](#) in Python.

If you wanted to [open a file](#), do something with that file, and then make sure that the file was closed correctly, then you would use a context manager. Consider this example in which `names.txt` contains a list of names, one per line:

Python

>>>

```
>>> with open("names.txt") as input_file:
...     for name in input_file:
...         print(name.strip())
...
Jim
Pam
Cece
Philip
```

The file I/O context manager provided by `open()` and initiated with the `with` keyword opens the file for reading, assigns the open file pointer to `input_file`, then executes whatever code you specify in the `with` block. Then, after the block is executed, the file pointer closes. Even if your code in the `with` block raises an exception, the file pointer would still close.

For a great example of using `with` and context managers, check out [Python Timer Functions: Three Ways to Monitor Your Code](#).

The as Keyword Used With with

If you want access to the results of the expression or context manager passed to `with`, you'll need to alias it using `as`. You may have also seen `as` used to alias imports and exceptions, and this is no different. The alias is available in the `with` block:

Python

```
with <expr> as <alias>:
    <statements>
```

Most of the time, you'll see these two Python keywords, `with` and `as`, used together.

The pass Keyword

Since Python doesn't have block indicators to specify the end of a block, the `pass` keyword is used to specify that the block is intentionally left blank. It's the equivalent of a **no-op**, or **no operation**. Here are a few examples of using `pass` to specify that the block is blank:

Python

```
def my_function():
    pass

class MyClass:
    pass

if True:
    pass
```

For more on `pass`, check out [Conditional Statements in Python](#).

The `lambda` Keyword

The `lambda` keyword is used to define a function that doesn't have a name and has only one statement, the results of which are returned. Functions defined with `lambda` are referred to as **lambda functions**:

Python

```
lambda <args>: <statement>
```

A basic example of a `lambda` function that computes the argument raised to the power of 10 would look like this:

Python

```
p10 = lambda x: x**10
```

This is equivalent to defining a function with `def`:

Python

```
def p10(x):
    return x**10
```

One common use for a `lambda` function is specifying a different behavior for another function. For example, imagine you wanted to sort a list of strings by their integer values. The default behavior of `sorted()` would sort the strings alphabetically. But with `sorted()`, you can specify which key the list should be sorted on.

A `lambda` function provides a nice way to do so:

Python

>>>

```
>>> ids = ["id1", "id2", "id30", "id3", "id20", "id10"]
>>> sorted(ids)
['id1', 'id10', 'id2', 'id20', 'id3', 'id30']

>>> sorted(ids, key=lambda x: int(x[2:]))
['id1', 'id2', 'id3', 'id10', 'id20', 'id30']
```

This example sorts the list based not on alphabetical order but on the numerical order of the last characters of the strings after converting them to integers. Without `lambda`, you would have had to define a function, give it a name, and then pass it to `sorted()`. `lambda` made this code cleaner.

For comparison, this is what the example above would look like without using `lambda`:

Python

>>>

```
>>> def sort_by_int(x):
...     return int(x[2:])
...
>>> ids = ["id1", "id2", "id30", "id3", "id20", "id10"]
>>> sorted(ids, key=sort_by_int)
['id1', 'id2', 'id3', 'id10', 'id20', 'id30']
```

This code produces the same result as the `lambda` example, but you need to define the function before using it.

For a lot more information about lambda, check out [How to Use Python Lambda Functions](#).

Returning Keywords: return, yield

There are two Python keywords used to specify what gets returned from functions or methods: `return` and `yield`. Understanding when and where to use `return` is vital to becoming a better Python programmer. The `yield` keyword is a more advanced feature of Python, but it can also be a useful tool to understand.

The `return` Keyword

Python's `return` keyword is valid only as part of a function defined with `def`. When Python encounters this keyword, it will exit the function at that point and return the results of whatever comes after the `return` keyword:

Python

```
def <function>():
    return <expr>
```

When given no expression, `return` will return `None` by default:

Python

```
>>> def return_none():
...     return
...
...
>>> return_none()
>>> r = return_none()
>>> print(r)
None
```

>>>

Most of the time, however, you want to return the results of an expression or a specific value:

Python

```
>>> def plus_1(num):
...     return num + 1
...
...
>>> plus_1(9)
10
>>> r = plus_1(9)
>>> print(r)
10
```

>>>

You can even use the `return` keyword multiple times in a function. This allows you to have multiple exit points in your function. A classic example of when you would want to have multiple `return` statements is the following recursive solution to calculating [factorial](#):

Python

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

In the `factorial` function above, there are two cases in which you would want to return from the function. The first is the base case, when the number is 1, and the second is the regular case, when you want to multiply the current number by the next number's factorial value.

To learn more about the `return` keyword, check out [Defining Your Own Python Function](#).

The `yield` Keyword

Python's `yield` keyword is kind of like the `return` keyword in that it specifies what gets returned from a function. However, when a function has a `yield` statement, what gets returned is a [generator](#). The generator can then be passed to Python's built-in `next()` to get the next value returned from the function.

When you call a function with `yield` statements, Python executes the function until it reaches the first `yield`.

When you call a function with `yield` statements, it **only** executes the function until it reaches the first `yield` keyword and then returns a generator. These are known as generator functions:

Python

```
def <function>():
    yield <expr>
```

The most straightforward example of this would be a generator function that returns the same set of values:

Python

>>>

```
>>> def family():
...     yield "Pam"
...     yield "Jim"
...     yield "Cece"
...     yield "Philip"
...
>>> names = family()
>>> names
<generator object family at 0x7f47a43577d8>
>>> next(names)
'Pam'
>>> next(names)
'Jim'
>>> next(names)
'Cece'
>>> next(names)
'Philip'
>>> next(names)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Once the `StopIteration` exception is raised, the generator is done returning values. In order to go through the names again, you would need to call `family()` again and get a new generator. Most of the time, a generator function will be called as part of a `for` loop, which does the `next()` calls for you.

For much more on the `yield` keyword and using generators and generator functions, check out [How to Use Generators and `yield` in Python](#) and [Python Generators 101](#).

Import Keywords: `import`, `from`, `as`

For those tools that, unlike Python keywords and built-ins, are not already available to your Python program, you'll need to import them into your program. There are many useful modules available in Python's standard library that are only an import away. There are also many other useful libraries and tools available in [PyPI](#) that, once you've installed them into your environment, you'll need to import into your programs.

The following are brief descriptions of the three Python keywords used for importing modules into your program. For more information about these keywords, check out [Python Modules and Packages – An Introduction](#) and [Python import: Advanced Techniques and Tips](#).

The `import` Keyword

Python's `import` keyword is used to import, or include, a module for use in your Python program. Basic usage syntax looks like this:

Python

```
import <module>
```

After that statement runs, the `<module>` will be available to your program.

For example, if you want to use the `Counter` class from the `collections` module in the standard library, then you

can use the following code:

```
Python >>>
>>> import collections
>>> collections.Counter()
Counter()
```

Importing collections in this way makes the whole collections module, including the Counter class, available to your program. By using the module name, you have access to all the tools available in that module. To get access to Counter, you reference it from the module: collections.Counter.

The from Keyword

The **from** keyword is used together with **import** to import something specific from a module:

```
Python >>>
from <module> import <thing>
```

This will import whatever <thing> is inside <module> to be used inside your program. These two Python keywords, **from** and **import**, are used together.

If you want to use Counter from the collections module in the standard library, then you can import it specifically:

```
Python >>>
>>> from collections import Counter
>>> Counter()
Counter()
```

Importing Counter like this makes the Counter class available, but nothing else from the collections module is available. Counter is now available without you having to reference it from the collections module.

The as Keyword

The **as** keyword is used to **alias** an imported module or tool. It's used together with the Python keywords **import** and **from** to change the name of the thing being imported:

```
Python >>>
import <module> as <alias>
from <module> import <thing> as <alias>
```

For modules that have really long names or a commonly used import alias, as can be helpful in creating the alias.

If you want to import the Counter class from the collections module but name it something different, you can alias it by using as:

```
Python >>>
>>> from collections import Counter as C
>>> C()
Counter()
```

Now Counter is available to be used in your program, but it's referenced by C instead. A more common use of as import aliases is with [NumPy](#) or [Pandas](#) packages. These are commonly imported with standard aliases:

```
Python >>>
import numpy as np
import pandas as pd
```

This is a better alternative to just importing everything from a module, and it allows you to shorten the name of the

This is a little different from just importing everything from a module, and it allows you to import the name of the module being imported.

Exception-Handling Keywords: try, except, raise, finally, else, assert

One of the most common aspects of any Python program is the raising and catching of exceptions. Because this is such a fundamental aspect of all Python code, there are several Python keywords available to help make this part of your code clear and concise.

The sections below go over these Python keywords and their basic usage. For a more in-depth tutorial on these keywords, check out [Python Exceptions: An Introduction](#).

The try Keyword

Any exception-handling block begins with Python's `try` keyword. This is the same in most other programming languages that have exception handling.

The code in the `try` block is code that might raise an exception. Several other Python keywords are associated with `try` and are used to define what should be done if different exceptions are raised or in different situations. These are `except`, `else`, and `finally`:

Python

```
try:  
    <statements>  
<except|else|finally>:  
    <statements>
```

A try block isn't valid unless it has at least one of the other Python keywords used for exception handling as part of the overall try statement.

If you wanted to calculate and return the miles per gallon of gas (mpg) given the miles driven and the gallons of gas used, then you could write a function like the following:

Python

```
def mpg(miles, gallons):  
    return miles / gallons
```

The first problem you might see is that your code could raise a `ZeroDivisionError` if the `gallons` parameter is passed in as `0`. The `try` keyword allows you to modify the code above to handle that situation appropriately:

Python

```
def mpg(miles, gallons):
    try:
        mpg = miles / gallons
    except ZeroDivisionError:
        mpg = None
    return mpg
```

Now if `gallons = 0`, then `mpg()` won't raise an exception and will return `None` instead. This might be better, or you might decide that you want to raise a different type of exception or handle this situation differently. You'll see an expanded version of this example below to illustrate the other keywords used for exception handling.

The except Keyword

Python's `except` keyword is used with `try` to define what to do when specific exceptions are raised. You can have one or more `except` blocks with a single `try`. The basic usage looks like this:

Python

```
try:  
    <statements>  
except <exception>:  
    <statements>
```

Taking the `mpg()` example from before, you could also do something specific in the event that someone passes types that won't work with the `/` operator. Having defined `mpg()` in a previous example, now try to call it with strings instead of numbers:

Python

>>>

```
>>> mpg("lots", "many")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in mpg
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

You could revise `mpg()` and use multiple `except` blocks to handle this situation, too:

Python

```
def mpg(miles, gallons):
    try:
        mpg = miles / gallons
    except ZeroDivisionError:
        mpg = None
    except TypeError as ex:
        print("you need to provide numbers")
        raise ex
    return mpg
```

Here, you modify `mpg()` to raise the `TypeError` exception only after you've printed a helpful reminder to the screen.

Notice that the `except` keyword can also be used in conjunction with the `as` keyword. This has the same effect as the other uses of `as`, giving the raised exception an alias so you can work with it in the `except` block.

Even though it's syntactically allowed, try not to use `except` statements as implicit catchalls. It's better practice to always explicitly catch *something*, even if it's just `Exception`:

Python

```
try:
    1 / 0
except: # Don't do this
    pass

try:
    1 / 0
except Exception: # This is better
    pass

try:
    1 / 0
except ZeroDivisionError: # This is best
    pass
```

If you really do want to catch a broad range of exceptions, then specify the parent `Exception`. This is more explicitly a catchall, and it won't also catch exceptions you probably don't want to catch, like `RuntimeError` or `KeyboardInterrupt`.

The `raise` Keyword

The `raise` keyword raises an exception. If you find you need to raise an exception, then you can use `raise` followed by the exception to be raised:

Python

```
raise <exception>
```

You used `raise` previously, in the `mpg()` example. When you catch the `TypeError`, you re-raise the exception after

printing a message to the screen.

The `finally` Keyword

Python's `finally` keyword is helpful for specifying code that should be run no matter what happens in the `try`, `except`, or `else` blocks. To use `finally`, use it as part of a `try` block and specify the statements that should be run no matter what:

Python

```
try:  
    <statements>  
finally:  
    <statements>
```

Using the example from before, it might be helpful to specify that, no matter what happens, you want to know what arguments the function was called with. You could modify `mpg()` to include a `finally` block that does just that:

Python

```
def mpg(miles, gallons):  
    try:  
        mpg = miles / gallons  
    except ZeroDivisionError:  
        mpg = None  
    except TypeError as ex:  
        print("you need to provide numbers")  
        raise ex  
    finally:  
        print(f"mpg({{miles}}, {{gallons}})")  
    return mpg
```

Now, no matter how `mpg()` is called or what the result is, you print the arguments supplied by the user:

Python

>>>

```
>>> mpg(10, 1)  
mpg(10, 1)  
10.0  
  
>>> mpg("lots", "many")  
you need to provide numbers  
mpg(lots, many)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 8, in mpg  
  File "<stdin>", line 3, in mpg  
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

The `finally` keyword can be a very useful part of your exception-handling code.

The `else` Keyword Used With `try` and `except`

You've learned that the `else` keyword can be used with both the `if` keyword and loops in Python, but it has one more use. It can be combined with the `try` and `except` Python keywords. You can use `else` in this way only if you also use at least one `except` block:

Python

```
try:  
    <statements>  
except <exception>:  
    <statements>  
else:  
    <statements>
```

In this context, the code in the `else` block is executed only if an exception was *not* raised in the `try` block. In other words, if the `try` block executed all the code successfully, then the `else` block code would be executed.

In the `mn() example`, imagine you want to make sure that the `mn` result is always returned as a `float` no matter

what number combination is passed in. One of the ways you could do this is to use an `else` block. If the `try` block calculation of `mpg` is successful, then you convert the result to a `float` in the `else` block before returning:

Python

```
def mpg(miles, gallons):
    try:
        mpg = miles / gallons
    except ZeroDivisionError:
        mpg = None
    except TypeError as ex:
        print("you need to provide numbers")
        raise ex
    else:
        mpg = float(mpg) if mpg is not None else mpg
    finally:
        print(f"mpg({miles}, {gallons})")
    return mpg
```

Now the results of a call to `mpg()`, if successful, will always be a `float`.

For more on using the `else` block as part of a `try` and `except` block, check out [Python Exceptions: An Introduction](#).

The assert Keyword

The `assert` keyword in Python is used to specify an `assert` statement, or an assertion about an expression. An `assert` statement will result in a no-op if the expression (`<expr>`) is truthy, and it will raise an `AssertionError` if the expression is falsy. To define an assertion, use `assert` followed by an expression:

Python

```
assert <expr>
```

Generally, `assert` statements will be used to make sure something that needs to be true is. You shouldn't rely on them, however, as they can be ignored depending on how your Python program is executed.

Asynchronous Programming Keywords: `async`, `await`

Asynchronous programming is a complex topic. There are two Python keywords defined to help make asynchronous code more readable and cleaner: `async` and `await`.

The sections below introduce the two asynchronous keywords and their basic syntax, but they won't go into depth on asynchronous programming. To learn more about asynchronous programming, check out [Async IO in Python: A Complete Walkthrough](#) and [Getting Started With Async Features in Python](#).

The `async` Keyword

The `async` keyword is used with `def` to define an asynchronous function, or [coroutine](#). The syntax is just like defining a function, with the addition of `async` at the beginning:

Python

```
async def <function>(<params>):
    <statements>
```

You can make a function asynchronous by adding the `async` keyword before the function's regular definition.

The `await` Keyword

Python's `await` keyword is used in asynchronous functions to specify a point in the function where control is given back to the event loop for other functions to run. You can use it by placing the `await` keyword in front of a call to any `async` function:

Python

```
await <some async function call>
    ...
```

```
# UK
<var> = await <some async function call>
```

When using `await`, you can either call the asynchronous function and ignore the results, or you can store the results in a variable when the function eventually returns.

Variable Handling Keywords: `del`, `global`, `nonlocal`

Three Python keywords are used to work with variables. The `del` keyword is much more commonly used than the `global` and `nonlocal` keywords. But it's still helpful to know and understand all three keywords so you can identify when and how to use them.

The `del` Keyword

`del` is used in Python to unset a variable or name. You can use it on variable names, but a more common use is to remove indexes from a [list](#) or [dictionary](#). To unset a variable, use `del` followed by the variable you want to unset:

Python

```
del <variable>
```

Let's assume you want to clean up a dictionary that you got from an API response by throwing out keys you know you won't use. You can do so with the `del` keyword:

Python

>>>

```
>>> del response["headers"]
>>> del response["errors"]
```

This will remove the "headers" and "errors" keys from the dictionary response.

The `global` Keyword

If you need to modify a variable that isn't defined in a function but is defined in the **global scope**, then you'll need to use the `global` keyword. This works by specifying in the function which variables need to be pulled into the function from the global scope:

Python

```
global <variable>
```

A basic example is incrementing a global variable with a function call. You can do that with the `global` keyword:

Python

>>>

```
>>> x = 0
>>> def inc():
...     global x
...     x += 1
...
>>> inc()
>>> x
1
>>> inc()
>>> x
2
```

This is generally not considered good practice, but it does have its uses. To learn much more on the `global` keyword, check out [Python Scope & the LEGB Rule: Resolving Names in Your Code](#).

The `nonlocal` Keyword

The `nonlocal` keyword is similar to `global` in that it allows you to modify variables from a different scope. With `global`, the scope you're pulling from is the `global scope`. With `nonlocal`, the scope you're pulling from is the `parent`.

global, the scope you're putting them in is the global scope. With nonlocal, the scope you're putting them in is the **parent scope**. The syntax is similar to global:

Python

```
nonlocal <variable>
```

This keyword isn't used very often, but it can be handy at times. For more on scoping and the nonlocal keyword, check out [Python Scope & the LEGB Rule: Resolving Names in Your Code](#).

Deprecated Python Keywords

Sometimes a Python keyword becomes a built-in function. That was the case with both print and exec. These used to be Python keywords in version 2.7, but they've since been changed to built-in functions.

The Former print Keyword

When **print** was a keyword, the syntax to print something to the screen looked like this:

Python

```
print "Hello, World"
```

Notice that it looks like a lot of the other keyword statements, with the keyword followed by the arguments.

Now print is no longer a keyword, and printing is accomplished with the built-in `print()`. To print something to the screen, you now use the following syntax:

Python

```
print("Hello, World")
```

For more on printing, check out [Your Guide to the Python `print\(\)` Function](#).

The Former exec Keyword

In Python 2.7, the **exec** keyword took Python code as a string and executed it. This was done with the following syntax:

Python

```
exec "<statements>"
```

You can get the same behavior in Python 3+, only with the built-in `exec()`. For example, if you wanted to execute "`x = 12 * 7`" in your Python code, then you could do the following:

Python

>>>

```
>>> exec("x = 12 * 7")
>>> x == 84
True
```

For more on `exec()` and its uses, check out [How to Run Your Python Scripts](#).

Conclusion

Python keywords are the fundamental building blocks of any Python program. Understanding their proper use is key to improving your skills and knowledge of Python.

Throughout this article, you've seen a few things to solidify your understanding Python keywords and to help you write more efficient and readable code.

In this article, you've learned:

- The **Python keywords** in version 3.8 and their basic usage
- Several **resources** to help deepen your understanding of many of the keywords
- How to use Python's **keywords module** to work with keywords in a programmatic way

If you understand most of these keywords and feel comfortable using them, then you might be interested to learn more about [Python's grammar](#) and how the [statements](#) that use these keywords are specified and constructed.

About **Chad Hansen**

Chad is an avid Pythonista and does web development with Django fulltime. Chad lives in Utah with his wife and six kids.

[» More about Chad](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

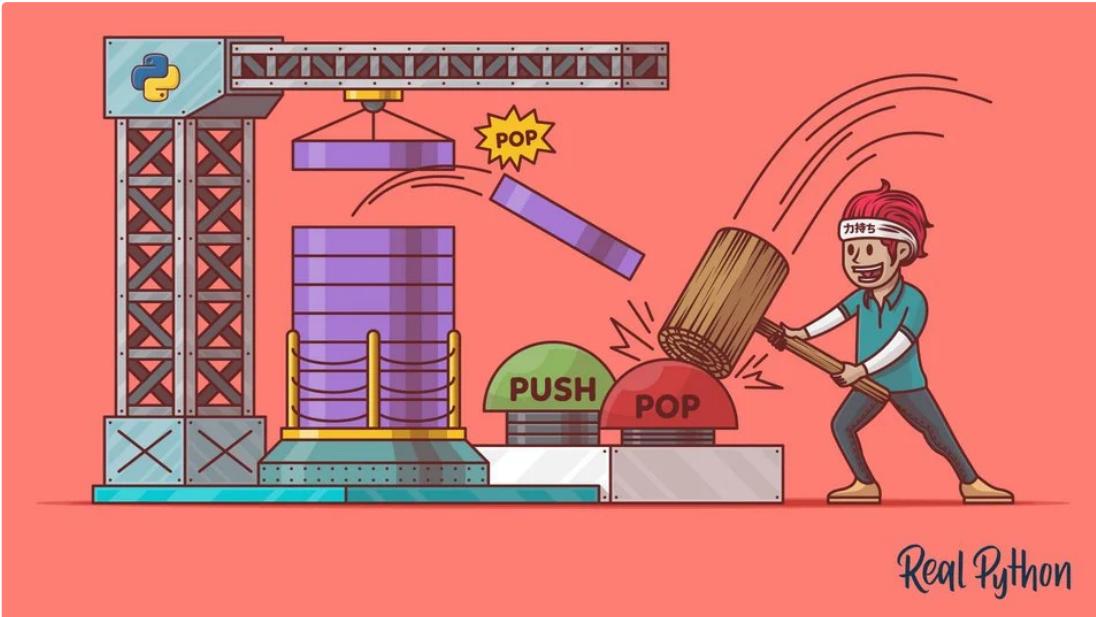
[Jim](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)



Real Python

How to Implement a Python Stack

by Jim Anderson 10 Comments basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Is a Stack?](#)
- [Implementing a Python Stack](#)
 - [Using list to Create a Python Stack](#)
 - [Using collections.deque to Create a Python Stack](#)
- [Python Stacks and Threading](#)
- [Python Stacks: Which Implementation Should You Use?](#)
- [Conclusion](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Implement a Python Stack](#)

Have you heard of stacks and wondered what they are? Do you have the general idea but are wondering how to implement a Python stack? You've come to the right place!

In this tutorial, you'll learn:

- How to recognize when a stack is a good choice for data structures
- How to decide which implementation is best for your program
- What extra considerations to make about stacks in a threading or multiprocessing environment

This tutorial is for Pythonistas who are comfortable running scripts, know what a [list](#) is and how to use it, and are wondering how to implement Python stacks.

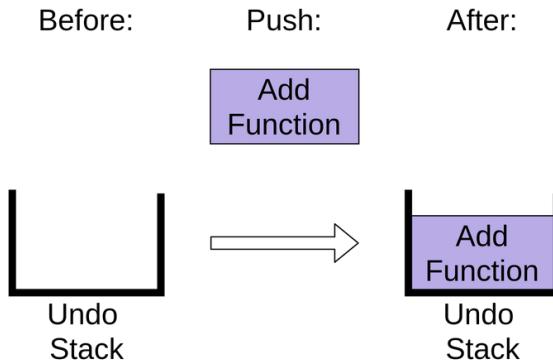
Free Bonus: [Click here to get a Python Cheat Sheet](#) and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

What Is a Stack?

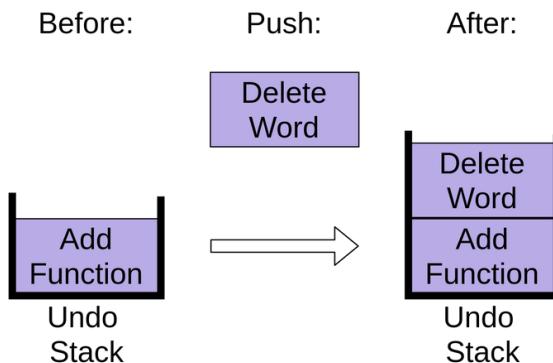
A [stack](#) is a [data structure](#) that stores items in an Last-In/First-Out manner. This is frequently referred to as LIFO. This is in contrast to a [queue](#), which stores items in a First-In/First-Out (FIFO) manner.

It's probably easiest to understand a stack if you think of a use case you're likely familiar with: the *Undo* feature in your editor.

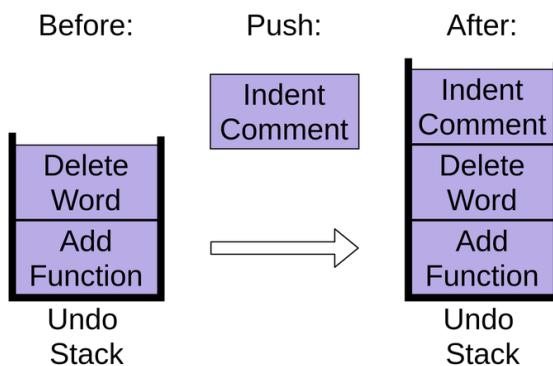
Let's imagine you're editing a Python file so we can look at some of the operations you perform. First, you add a new function. This adds a new item to the undo stack:



You can see that the stack now has an *Add Function* operation on it. After adding the function, you delete a word from a comment. This also gets added to the undo stack:

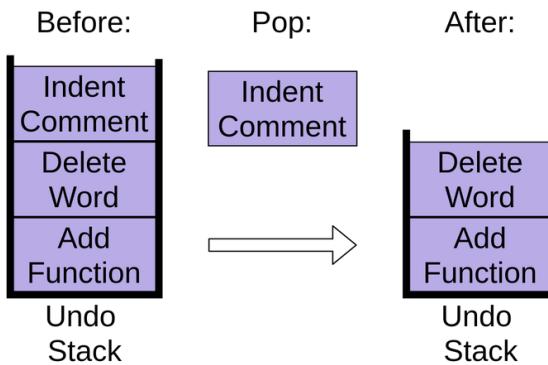


Notice how the *Delete Word* item is placed on top of the stack. Finally you indent a comment so that it's lined up properly:



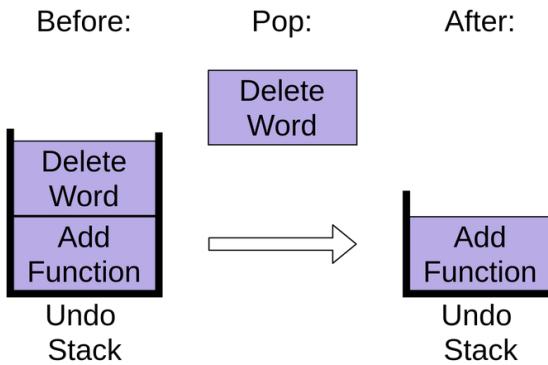
You can see that each of these commands are stored in an undo stack, with each new command being put at the top. When you're working with stacks, adding new items like this is called *push*.

Now you've decided to undo all three of those changes, so you hit the undo command. It takes the item at the top of the stack, which was indenting the comment, and removes that from the stack:



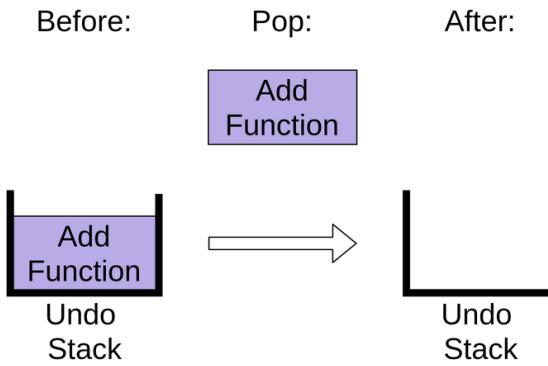
Your editor undoes the indent, and the undo stack now contains two items. This operation is the opposite of push and is commonly called pop.

When you hit undo again, the next item is popped off the stack:



This removes the *Delete Word* item, leaving only one operation on the stack.

Finally, if you hit *Undo* a third time, then the last item will be popped off the stack:



The undo stack is now empty. Hitting *Undo* again after this will have no effect because your undo stack is empty, at least in most editors. You'll see what happens when you call `.pop()` on an empty stack in the implementation descriptions below.

Implementing a Python Stack

There are a couple of options when you're implementing a Python stack. This article won't cover all of them, just the basic ones that will meet almost all of your needs. You'll focus on using data structures that are part of the Python library, rather than writing your own or using third-party packages.

You'll look at the following Python stack implementations:

- `list`
- `collections.deque`
- `queue.LifoQueue`

Using `list` to Create a Python Stack

The built-in `list` structure that you likely use frequently in your programs can be used as a stack. Instead of `.push()`, you can use `.append()` to add new elements to the top of your stack, while `.pop()` removes the elements in the LIFO order:

```
Python >>> myStack = []
>>> myStack.append('a')
>>> myStack.append('b')
>>> myStack.append('c')

>>> myStack
['a', 'b', 'c']

>>> myStack.pop()
'a'
>>> myStack.pop()
'b'
>>> myStack.pop()
'c'

>>> myStack.pop()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: pop from empty list
```

You can see in the final command that a `list` will raise an `IndexError` if you call `.pop()` on an empty stack.

`list` has the advantage of being familiar. You know how it works and likely have used it in your programs already.

Unfortunately, `list` has a few shortcomings compared to other data structures you'll look at. The biggest issue is that it can run into speed issues as it grows. The items in a `list` are stored with the goal of providing fast access to random elements in the `list`. At a high level, this means that the items are stored next to each other in memory.

If your stack grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some `.append()` calls taking much longer than other ones.

There is a less serious problem as well. If you use `.insert()` to add an element to your stack at a position other than the end, it can take much longer. This is not normally something you would do to a stack, however.

The next data structure will help you get around the reallocation problem you saw with `list`.

Using `collections.deque` to Create a Python Stack

The `collections` module contains [deque](#), which is useful for creating Python stacks. `deque` is pronounced “deck” and stands for “double-ended queue.”

You can use the same methods on `deque` as you saw above for `list`, `.append()`, and `.pop()`:

```
>>> from collections import deque
>>> myStack = deque()

>>> myStack.append('a')
>>> myStack.append('b')
>>> myStack.append('c')

>>> myStack
deque(['a', 'b', 'c'])

>>> myStack.pop()
'c'
>>> myStack.pop()
'b'
>>> myStack.pop()
'a'

>>> myStack.pop()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: pop from an empty deque
```

This looks almost identical to the `list` example above. At this point, you might be wondering why the Python core developers would create two data structures that look the same.

Why Have `deque` and `list`?

As you saw in the discussion about `list` above, it was built upon blocks of contiguous memory, meaning that the items in the list are stored right next to each other:

This works great for several operations, like indexing into the `list`. Getting `myList[3]` is fast, as Python knows exactly where to look in memory to find it. This memory layout also allows slices to work well on lists.

The contiguous memory layout is the reason that `list` might need to take more time to `.append()` some objects than others. If the block of contiguous memory is full, then it will need to get another block, which can take much longer than a normal `.append()`:

`deque`, on the other hand, is built upon a doubly linked list. In a [linked list structure](#), each entry is stored in its own memory block and has a reference to the next entry in the list.

A doubly linked list is just the same, except that each entry has references to both the previous and the next entry in the list. This allows you to easily add nodes to either end of the list.

Adding a new entry into a linked list structure only requires setting the new entry's reference to point to the current top of the stack and then pointing the top of the stack to the new entry:

This constant-time addition and removal of entries onto a stack comes with a trade-off, however. Getting `myDeque[3]` is slower than it was for a list, because Python needs to walk through each node of the list to get to the third element.

Fortunately, you rarely want to do random indexing or slicing on a stack. Most operations on a stack are either push or pop.

The constant time `.append()` and `.pop()` operations make deque an excellent choice for implementing a Python stack if your code doesn't use threading.

Python Stacks and Threading

Python stacks can be useful in multi-threaded programs as well, but if you're not interested in threading, then you can safely skip this section and jump to the summary.

The two options you've seen so far, `list` and `deque`, behave differently if your program has threads.

To start with the simpler one, you should never use `list` for any data structure that can be accessed by multiple threads. `list` is not thread-safe.

Note: If you need a refresher on thread safety and race conditions, check out [An Intro to Threading in Python](#).

`deque` is a little more complex, however. If you read the documentation for `deque`, it clearly states that both the `.append()` and `.pop()` operations are atomic, meaning that they won't be interrupted by a different thread.

So if you restrict yourself to using only `.append()` and `.pop()`, then you will be thread safe.

The concern with using `deque` in a threaded environment is that there are other methods in that class, and those are not specifically designed to be atomic, nor are they thread safe.

So, while it's possible to build a thread-safe Python stack using a `deque`, doing so exposes yourself to someone misusing it in the future and causing race conditions.

Okay, if you're threading, you can't use `list` for a stack and you probably don't want to use `deque` for a stack, so how *can* you build a Python stack for a threaded program?

The answer is in the `queue` module, [`queue.LifoQueue`](#). Remember how you learned that stacks operate on the Last-In/First-Out principle? Well, that's what the "Lifo" portion of `LifoQueue` stands for.

While the interface for `list` and `deque` were similar, `LifoQueue` uses `.put()` and `.get()` to add and remove data from the stack:

```
>>> from queue import LifoQueue
>>> myStack = LifoQueue()

>>> myStack.put('a')
>>> myStack.put('b')
>>> myStack.put('c')

>>> myStack
<queue.LifoQueue object at 0x7f408885e2b0>

>>> myStack.get()
'c'
>>> myStack.get()
'b'
>>> myStack.get()
'a'

>>> # myStack.get() <--- waits forever
>>> myStack.get_nowait()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
    File "/usr/lib/python3.7/queue.py", line 198, in get_nowait
      return self.get(block=False)
    File "/usr/lib/python3.7/queue.py", line 167, in get
      raise Empty
queue.Empty
```

Unlike deque, LifoQueue is designed to be fully thread-safe. All of its methods are safe to use in a threaded environment. It also adds optional time-outs to its operations which can frequently be a must-have feature in threaded programs.

This full thread safety comes at a cost, however. To achieve this thread-safety, LifoQueue has to do a little extra work on each operation, meaning that it will take a little longer.

Frequently, this slight slow down will not matter to your overall program speed, but if you've measured your performance and discovered that your stack operations are the bottleneck, then carefully switching to a deque might be worth doing.

I'd like to stress again that switching from LifoQueue to deque because it's faster without having measurements showing that your stack operations are a bottleneck is an example of [premature optimization](#). Don't do that.

Python Stacks: Which Implementation Should You Use?

In general, you should use a deque if you're not using threading. If you are using threading, then you should use a LifoQueue unless you've measured your performance and found that a small boost in speed for pushing and popping will make enough difference to warrant the maintenance risks.

list may be familiar, but it should be avoided because it can potentially have memory reallocation issues. The interfaces for deque and list are identical, and deque doesn't have these issues, which makes deque the best choice for your non-threaded Python stack.

Conclusion

You now know what a stack is and have seen situations where they can be used in real-life programs. You've evaluated three different options for implementing stacks and seen that deque is a great choice for non-threaded programs. If you're implementing a stack in a threading environment, then it's likely a good idea to use a LifoQueue.

You are now able to:

- Recognize when a stack would be a good data structure
- Select which implementation is right for your problem

If you still have questions, feel free to reach out in the comments sections below. Now, go write some code since you gained another tool to help you solve your programming problems!

 [Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Implement a Python Stack](#)

About Jim Anderson

Jim has been programming for a long time in a variety of languages. He has worked on embedded systems, built distributed build systems, done off-shore vendor management, and s in many, many meetings.

[» More about Jim](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

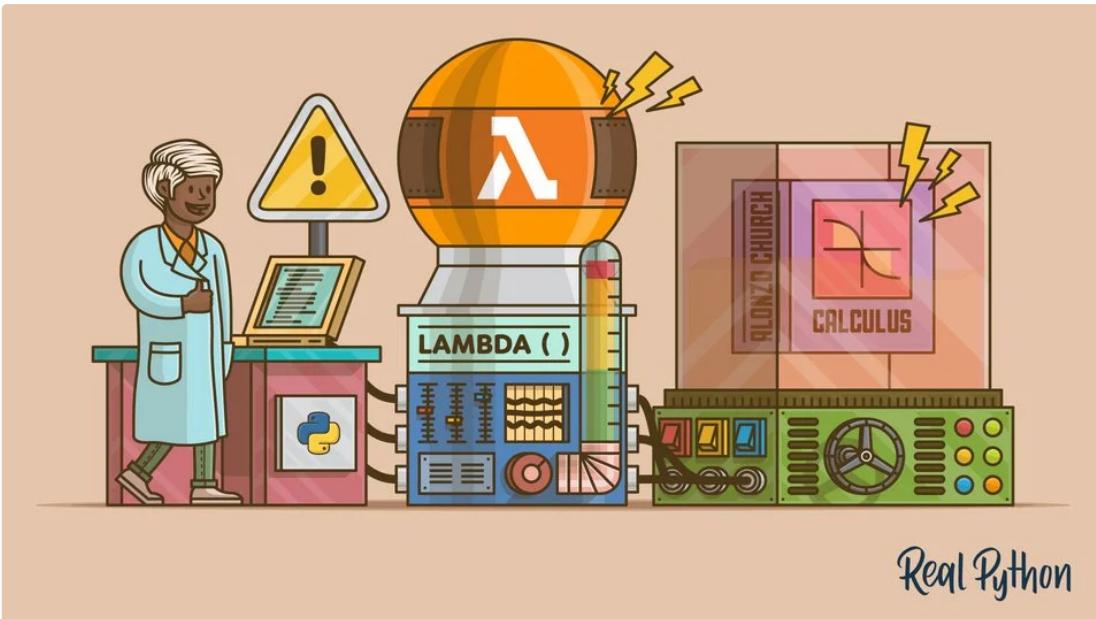
[Joanna](#)

[Mike](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [How to Implement a Python Stack](#)



Real Python

How to Use Python Lambda Functions

by Andre Burgaud 16 Comments best-practices intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Lambda Calculus](#)
 - [History](#)
 - [First Example](#)
- [Anonymous Functions](#)
- [Python Lambda and Regular Functions](#)
 - [Functions](#)
 - [Traceback](#)
 - [Syntax](#)
 - [Arguments](#)
 - [Decorators](#)
 - [Closure](#)
 - [Evaluation Time](#)
 - [Testing Lambdas](#)
- [Lambda Expression Abuses](#)
 - [Raising an Exception](#)
 - [Cryptic Style](#)
 - [Python Classes](#)
- [Appropriate Uses of Lambda Expressions](#)
 - [Classic Functional Constructs](#)
 - [Key Functions](#)
 - [UI Frameworks](#)
 - [Python Interpreter](#)
 - [timeit](#)
 - [Monkey Patching](#)
- [Alternatives to Lambdas](#)
 - [Map](#)
 - [Filter](#)
 - [Reduce](#)
- [Are Lambdas Pythonic or Not?](#)
- [Conclusion](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Use Python Lambda Functions](#)

Python and other languages like Java, C#, and even C++ have had lambda functions added to their syntax, whereas languages like LISP or the ML family of languages, Haskell, OCaml, and F#, use lambdas as a core concept.

Python lambdas are little, anonymous functions, subject to a more restrictive but more concise syntax than regular Python functions.

By the end of this article, you'll know:

- How Python lambdas came to be
- How lambdas compare with regular function objects
- How to write lambda functions
- Which functions in the Python standard library leverage lambdas
- When to use or avoid Python lambda functions

Notes: You'll see some code examples using `lambda` that seem to blatantly ignore Python style best practices. This is only intended to illustrate lambda calculus concepts or to highlight the capabilities of Python `lambda`.

Those questionable examples will be contrasted with better approaches or alternatives as you progress through the article.

This tutorial is mainly for intermediate to experienced Python programmers, but it is accessible to any curious minds with interest in programming and lambda calculus.

All the examples included in this tutorial have been tested with Python 3.7.

Take the Quiz: Test your knowledge with our interactive “Python Lambda Functions” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

Free Bonus: [Click here to get access to a chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Lambda Calculus

Lambda expressions in Python and other programming languages have their roots in lambda calculus, a model of computation invented by Alonzo Church. You'll uncover when lambda calculus was introduced and why it's a fundamental concept that ended up in the Python ecosystem.

History

[Alonzo Church](#) formalized [lambda calculus](#), a language based on pure abstraction, in the 1930s. Lambda functions are also referred to as lambda abstractions, a direct reference to the abstraction model of Alonzo Church's original creation.

Lambda calculus can encode any computation. It is [Turing complete](#), but contrary to the concept of a [Turing machine](#), it is pure and does not keep any state.

Functional languages get their origin in mathematical logic and lambda calculus, while imperative programming languages embrace the state-based model of computation invented by Alan Turing. The two models of computation, lambda calculus and [Turing machines](#), can be translated into each other. This equivalence is known

as the [Church-Turing hypothesis](#).

Functional languages directly inherit the lambda calculus philosophy, adopting a declarative approach of programming that emphasizes abstraction, data transformation, composition, and purity (no state and no side effects). Examples of functional languages include [Haskell](#), [Lisp](#), or [Erlang](#).

By contrast, the Turing Machine led to imperative programming found in languages like [Fortran](#), [C](#), or [Python](#).

The imperative style consists of programming with statements, driving the flow of the program step by step with detailed instructions. This approach promotes mutation and requires managing state.

The separation in both families presents some nuances, as some functional languages incorporate imperative features, like [OCaml](#), while functional features have been permeating the imperative family of languages in particular with the introduction of lambda functions in [Java](#), or Python.

Python is not inherently a functional language, but it adopted some functional concepts early on. In January 1994, `map()`, `filter()`, `reduce()`, and the `lambda` operator were added to the language.

First Example

Here are a few examples to give you an appetite for some Python code, functional style.

The [identity function](#), a function that returns its argument, is expressed with a standard Python function definition using the [keyword](#) `def` as follows:

Python

>>>

```
>>> def identity(x):
...     return x
```

`identity()` takes an argument `x` and returns it upon invocation.

In contrast, if you use a Python lambda construction, you get the following:

Python

>>>

```
>>> lambda x: x
```

In the example above, the expression is composed of:

- **The keyword:** `lambda`
- **A bound variable:** `x`
- **A body:** `x`

Note: In the context of this article, a **bound variable** is an argument to a lambda function.

In contrast, a **free variable** is not bound and may be referenced in the body of the expression. A free variable can be a constant or a variable defined in the enclosing [scope](#) of the function.

You can write a slightly more elaborated example, a function that adds 1 to an argument, as follows:

Python

>>>

```
>>> lambda x: x + 1
```

You can apply the function above to an argument by surrounding the function and its argument with parentheses:

Python

>>>

```
>>> (lambda x: x + 1)(2)
3
```

[Reduction](#) is a lambda calculus strategy to compute the value of the expression. In the current example, it consists of replacing the bound variable `x` with the argument 2:

Text

```
(lambda x: x + 1)(2) = lambda 2: 2 + 1  
                      = 2 + 1  
                      = 3
```

Because a lambda function is an expression, it can be named. Therefore you could write the previous code as follows:

Python

>>>

```
>>> add_one = lambda x: x + 1  
>>> add_one(2)  
3
```

The above lambda function is equivalent to writing this:

Python

```
def add_one(x):  
    return x + 1
```

These functions all take a single argument. You may have noticed that, in the definition of the lambdas, the arguments don't have parentheses around them. Multi-argument functions (functions that take more than one argument) are expressed in Python lambdas by listing arguments and separating them with a comma (,) but without surrounding them with parentheses:

Python

>>>

```
>>> full_name = lambda first, last: f'Full name: {first.title()} {last.title()}'  
>>> full_name('guido', 'van rossum')  
'Full name: Guido Van Rossum'
```

The lambda function assigned to `full_name` takes two arguments and returns a string interpolating the two parameters `first` and `last`. As expected, the definition of the lambda lists the arguments with no parentheses, whereas calling the function is done exactly like a normal Python function, with parentheses surrounding the arguments.

Anonymous Functions

The following terms may be used interchangeably depending on the programming language type and culture:

- Anonymous functions
- Lambda functions
- Lambda expressions
- Lambda abstractions
- Lambda form
- Function literals

For the rest of this article after this section, you'll mostly see the term **lambda function**.

Taken literally, an anonymous function is a function without a name. In Python, an anonymous function is created with the `lambda` keyword. More loosely, it may or not be assigned a name. Consider a two-argument anonymous function defined with `lambda` but not bound to a variable. The lambda is not given a name:

Python

>>>

```
>>> lambda x, y: x + y
```

The function above defines a lambda expression that takes two arguments and returns their sum.

Other than providing you with the feedback that Python is perfectly fine with this form, it doesn't lead to any practical use. You could invoke the function in the Python interpreter:

Python

>>>

```
>>> _(1, 2)
3
```

The example above is taking advantage of the interactive interpreter-only feature provided via the underscore (`_`). See the note below for more details.

You could not write similar code in a Python module. Consider the `_` in the interpreter as a side effect that you took advantage of. In a Python module, you would assign a name to the lambda, or you would pass the lambda to a function. You'll use those two approaches later in this article.

Note: In the interactive interpreter, the single underscore (`_`) is bound to the last expression evaluated.

In the example above, the `_` points to the lambda function. For more details about the usage of this special character in Python, check out [The Meaning of Underscores in Python](#).

Another pattern used in other languages like JavaScript is to immediately execute a Python lambda function. This is known as an **Immediately Invoked Function Expression (IIFE)**, pronounce “iffy”). Here's an example:

```
Python >>>
>>> (lambda x, y: x + y)(2, 3)
5
```

The lambda function above is defined and then immediately called with two arguments (2 and 3). It returns the value 5, which is the sum of the arguments.

Several examples in this tutorial use this format to highlight the anonymous aspect of a lambda function and avoid focusing on `lambda` in Python as a shorter way of defining a function.

Python does not encourage using immediately invoked lambda expressions. It simply results from a lambda expression being callable, unlike the body of a normal function.

Lambda functions are frequently used with [higher-order functions](#), which take one or more functions as arguments or return one or more functions.

A lambda function can be a higher-order function by taking a function (normal or lambda) as an argument like in the following contrived example:

```
Python >>>
>>> high_ord_func = lambda x, func: x + func(x)
>>> high_ord_func(2, lambda x: x * x)
6
>>> high_ord_func(2, lambda x: x + 3)
7
```

Python exposes higher-order functions as built-in functions or in the standard library. Examples include `map()`, `filter()`, `functools.reduce()`, as well as key functions like `sort()`, `sorted()`, `min()`, and `max()`. You'll use lambda functions together with Python higher-order functions in [Appropriate Uses of Lambda Expressions](#).

Python Lambda and Regular Functions

This quote from the [Python Design and History FAQ](#) seems to set the tone about the overall expectation regarding the usage of lambda functions in Python:

Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you're too lazy to define a function. ([Source](#))

Nevertheless, don't let this statement deter you from using Python's `lambda`. At first glance, you may accept that a lambda function is a function with some [syntactic sugar](#) shortening the code to define or invoke a function. The following sections highlight the commonalities and subtle differences between normal Python functions and `lambda functions`.

lambda functions.

Functions

At this point, you may wonder what fundamentally distinguishes a lambda function bound to a variable from a regular function with a single `return` line: under the surface, almost nothing. Let's verify how Python sees a function built with a single `return` statement versus a function constructed as an expression (`lambda`).

The `dis` module exposes functions to analyze Python bytecode generated by the Python compiler:

```
Python >>>
>>> import dis
>>> add = lambda x, y: x + y
>>> type(add)
<class 'function'>
>>> dis.dis(add)
 1           0 LOAD_FAST              0 (x)
 2           2 LOAD_FAST              1 (y)
 4           4 BINARY_ADD
 6           6 RETURN_VALUE
>>> add
<function <lambda> at 0x7f30c6ce9ea0>
```

You can see that `dis()` expose a readable version of the Python bytecode allowing the inspection of the low-level instructions that the Python interpreter will use while executing the program.

Now see it with a regular function object:

```
Python >>>
>>> import dis
>>> def add(x, y): return x + y
>>> type(add)
<class 'function'>
>>> dis.dis(add)
 1           0 LOAD_FAST              0 (x)
 2           2 LOAD_FAST              1 (y)
 4           4 BINARY_ADD
 6           6 RETURN_VALUE
>>> add
<function add at 0x7f30c6ce9f28>
```

The bytecode interpreted by Python is the same for both functions. But you may notice that the naming is different: the function name is `add` for a function defined with `def`, whereas the Python `lambda` function is seen as `<lambda>`.

Traceback

You saw in the previous section that, in the context of the `lambda` function, Python did not provide the name of the function, but only `<lambda>`. This can be a limitation to consider when an exception occurs, and a `traceback` shows only `<lambda>`:

```
Python >>>
>>> div_zero = lambda x: x / 0
>>> div_zero(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
ZeroDivisionError: division by zero
```

The `traceback` of an exception raised while a `lambda` function is executed only identifies the function causing the exception as `<lambda>`.

Here's the same exception raised by a normal function:

```
Python >>>
>>> def div_zero(x): return x / 0
```

```
>>> div_zero(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in div_zero
ZeroDivisionError: division by zero
```

The normal function causes a similar error but results in a more precise traceback because it gives the function name, `div_zero`.

Syntax

As you saw in the previous sections, a lambda form presents syntactic distinctions from a normal function. In particular, a lambda function has the following characteristics:

- It can only contain expressions and can't include statements in its body.
- It is written as a single line of execution.
- It does not support type annotations.
- It can be immediately invoked (IIFE).

No Statements

A lambda function can't contain any statements. In a lambda function, statements like `return`, `pass`, `assert`, or `raise` will raise a [SyntaxError](#) exception. Here's an example of adding `assert` to the body of a lambda:

```
Python >>>
>>> (lambda x: assert x == 2)(2)
  File "<input>", line 1
    (lambda x: assert x == 2)(2)
          ^
SyntaxError: invalid syntax
```

This contrived example intended to assert that parameter `x` had a value of 2. But, the interpreter identifies a `SyntaxError` while parsing the code that involves the statement `assert` in the body of the `lambda`.

Single Expression

In contrast to a normal function, a Python lambda function is a single expression. Although, in the body of a `lambda`, you can spread the expression over several lines using parentheses or a multiline string, it remains a single expression:

```
Python >>>
>>> (lambda x:
... (x % 2 and 'odd' or 'even'))(3)
'odd'
```

The example above returns the string '`odd`' when the lambda argument is odd, and '`even`' when the argument is even. It spreads across two lines because it is contained in a set of parentheses, but it remains a single expression.

Type Annotations

If you've started adopting type hinting, which is now available in Python, then you have another good reason to prefer normal functions over Python lambda functions. Check out [Python Type Checking \(Guide\)](#) to get learn more about Python type hints and type checking. In a lambda function, there is no equivalent for the following:

```
Python
def full_name(first: str, last: str) -> str:
    return f'{first.title()} {last.title()}'
```

Any type error with `full_name()` can be caught by tools like [mypy](#) or [pyre](#), whereas a `SyntaxError` with the equivalent lambda function is raised at runtime:

```
Python >>>
>>> lambda first: str, last: str: first.title() + " " + last.title() -> str
```

```
File "<stdin>", line 1
    lambda first: str, last: str: first.title() + " " + last.title() -> str
SyntaxError: invalid syntax
```

Like trying to include a statement in a lambda, adding type annotation immediately results in a `SyntaxError` at runtime.

IIFE

You've already seen several examples of [immediately invoked function execution](#):

```
Python >>>
>>> (lambda x: x * x)(3)
9
```

Outside of the Python interpreter, this feature is probably not used in practice. It's a direct consequence of a lambda function being callable as it is defined. For example, this allows you to pass the definition of a Python lambda expression to a higher-order function like `map()`, `filter()`, or `functools.reduce()`, or to a key function.

Arguments

Like a normal function object defined with `def`, Python lambda expressions support all the different ways of passing arguments. This includes:

- Positional arguments
- Named arguments (sometimes called keyword arguments)
- Variable list of arguments (often referred to as `varargs`)
- Variable list of keyword arguments
- Keyword-only arguments

The following examples illustrate options open to you in order to pass arguments to lambda expressions:

```
Python >>>
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
>>> (lambda x, y, z=3: x + y + z)(1, 2)
6
>>> (lambda x, y, z=3: x + y + z)(1, y=2)
6
>>> (lambda *args: sum(args))(1,2,3)
6
>>> (lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3)
6
>>> (lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3)
6
```

Decorators

In Python, a [decorator](#) is the implementation of a pattern that allows adding a behavior to a function or a class. It is usually expressed with the `@decorator` syntax prefixing a function. Here's a contrived example:

```
Python
def some_decorator(f):
    def wraps(*args):
        print(f"Calling function '{f.__name__}'")
        return f(args)
    return wraps

@some_decorator
def decorated_function(x):
    print(f"With argument '{x}'")
```

In the example above, `some_decorator()` is a function that adds a behavior to `decorated_function()`, so that invoking `decorated_function("Python")` results in the following output:

Shell

```
Calling function 'decorated_function'  
With argument 'Python'
```

`decorated_function()` only prints with argument 'Python', but the decorator adds an extra behavior that also prints calling function 'decorated_function'.

A decorator can be applied to a lambda. Although it's not possible to decorate a lambda with the `@decorator` syntax, a decorator is just a function, so it can call the lambda function:

Python

```
1 # Defining a decorator  
2 def trace(f):  
3     def wrap(*args, **kwargs):  
4         print(f"[TRACE] func: {f.__name__}, args: {args}, kwargs: {kwargs}")  
5         return f(*args, **kwargs)  
6  
7     return wrap  
8  
9 # Applying decorator to a function  
10 @trace  
11 def add_two(x):  
12     return x + 2  
13  
14 # Calling the decorated function  
15 add_two(3)  
16  
17 # Applying decorator to a lambda  
18 print((trace(lambda x: x ** 2))(3))
```

`add_two()`, decorated with `@trace` on line 11, is invoked with argument 3 on line 15. By contrast, on line 18, a lambda function is immediately involved and embedded in a call to `trace()`, the decorator. When you execute the code above you obtain the following:

Shell

```
[TRACE] func: add_two, args: (3,), kwargs: {}  
[TRACE] func: <lambda>, args: (3,), kwargs: {}  
9
```

See how, as you've already seen, the name of the lambda function appears as `<lambda>`, whereas `add_two` is clearly identified for the normal function.

Decorating the lambda function this way could be useful for debugging purposes, possibly to debug the behavior of

a lambda function used in the context of a higher-order function or a key function. Let's see an example with `map()`:

Python

```
list(map(trace(lambda x: x*2), range(3)))
```

The first argument of `map()` is a lambda that multiplies its argument by 2. This lambda is decorated with `trace()`. When executed, the example above outputs the following:

Shell

```
[TRACE] Calling <lambda> with args (0,) and kwargs {}
[TRACE] Calling <lambda> with args (1,) and kwargs {}
[TRACE] Calling <lambda> with args (2,) and kwargs {}
[0, 2, 4]
```

The result `[0, 2, 4]` is a list obtained from multiplying each element of `range(3)`. For now, consider `range(3)` equivalent to the list `[0, 1, 2]`.

You will be exposed to `map()` in more details in [Map](#).

A lambda can also be a decorator, but it's not recommended. If you find yourself needing to do this, consult [PEP 8, Programming Recommendations](#).

For more on Python decorators, check out [Primer on Python Decorators](#).

Closure

A [closure](#) is a function where every free variable, everything except parameters, used in that function is bound to a specific value defined in the enclosing scope of that function. In effect, closures define the environment in which they run, and so can be called from anywhere.

The concepts of lambdas and closures are not necessarily related, although lambda functions can be closures in the same way that normal functions can also be closures. Some languages have special constructs for closure or lambda (for example, Groovy with an anonymous block of code as Closure object), or a lambda expression (for example, Java Lambda expression with a limited option for closure).

Here's a closure constructed with a normal Python function:

Python

```
1 def outer_func(x):
2     y = 4
3     def inner_func(z):
4         print(f"x = {x}, y = {y}, z = {z}")
5         return x + y + z
6     return inner_func
7
8 for i in range(3):
9     closure = outer_func(i)
10    print(f"closure({i+5}) = {closure(i+5)}")
```

`outer_func()` returns `inner_func()`, a nested function that computes the sum of three arguments:

- `x` is passed as an argument to `outer_func()`.
- `y` is a variable local to `outer_func()`.
- `z` is an argument passed to `inner_func()`.

To test the behavior of `outer_func()` and `inner_func()`, `outer_func()` is invoked three times in a `for` loop that prints the following:

Shell

```
x = 0, y = 4, z = 5
closure(5) = 9
x = 1, y = 4, z = 6
```

```
closure(6) = 11
x = 2, y = 4, z = 7
closure(7) = 13
```

On line 9 of the code, `inner_func()` returned by the invocation of `outer_func()` is bound to the name `closure`. On line 5, `inner_func()` captures `x` and `y` because it has access to its embedding environment, such that upon invocation of the closure, it is able to operate on the two free variables `x` and `y`.

Similarly, a `lambda` can also be a closure. Here's the same example with a Python `lambda` function:

Python

```
1 def outer_func(x):
2     y = 4
3     return lambda z: x + y + z
4
5 for i in range(3):
6     closure = outer_func(i)
7     print(f"closure({i+5}) = {closure(i+5)}")
```

When you execute the code above, you obtain the following output:

Shell

```
closure(5) = 9
closure(6) = 11
closure(7) = 13
```

On line 6, `outer_func()` returns a `lambda` and assigns it to the variable `closure`. On line 3, the body of the `lambda` function references `x` and `y`. The variable `y` is available at definition time, whereas `x` is defined at runtime when `outer_func()` is invoked.

In this situation, both the normal function and the `lambda` behave similarly. In the next section, you'll see a situation where the behavior of a `lambda` can be deceptive due to its evaluation time (definition time vs runtime).

Evaluation Time

In some situations involving [loops](#), the behavior of a Python `lambda` function as a closure may be counterintuitive. It requires understanding when free variables are bound in the context of a `lambda`. The following examples demonstrate the difference when using a regular function vs using a Python `lambda`.

Test the scenario first using a regular function:

Python

>>>

```
1 >>> def wrap(n):
2 ...     def f():
3 ...         print(n)
4 ...     return f
5 ...
6 >>> numbers = 'one', 'two', 'three'
7 >>> funcs = []
8 >>> for n in numbers:
9 ...     funcs.append(wrap(n))
10 ...
11 >>> for f in funcs:
12 ...     f()
13 ...
14 one
15 two
16 three
```

In a normal function, `n` is evaluated at definition time, on line 9, when the function is added to the list: `funcs.append(wrap(n))`.

Now, with the implementation of the same logic with a lambda function, observe the unexpected behavior:

```
Python >>>
1 >>> numbers = 'one', 'two', 'three'
2 >>> funcs = []
3 >>> for n in numbers:
4 ...     funcs.append(lambda: print(n))
5 ...
6 >>> for f in funcs:
7 ...     f()
8 ...
9 three
10 three
11 three
```

The unexpected result occurs because the free variable `n`, as implemented, is bound at the execution time of the lambda expression. The Python lambda function on line 4 is a closure that captures `n`, a free variable bound at runtime. At runtime, while invoking the function `f` on line 7, the value of `n` is `three`.

To overcome this issue, you can assign the free variable at definition time as follows:

```
Python >>>
1 >>> numbers = 'one', 'two', 'three'
2 >>> funcs = []
3 >>> for n in numbers:
4 ...     funcs.append(lambda n=n: print(n))
5 ...
6 >>> for f in funcs:
7 ...     f()
8 ...
9 one
10 two
11 three
```

A Python lambda function behaves like a normal function in regard to arguments. Therefore, a lambda parameter can be initialized with a default value: the parameter `n` takes the outer `n` as a default value. The Python lambda function could have been written as `lambda x=n: print(x)` and have the same result.

The Python lambda function is invoked without any argument on line 7, and it uses the default value `n` set at definition time.

Testing Lambdas

Python lambdas can be tested similarly to regular functions. It's possible to use both `unittest` and `doctest`.

`unittest`

The `unittest` module handles Python lambda functions similarly to regular functions:

```
Python
import unittest

addtwo = lambda x: x + 2

class LambdaTest(unittest.TestCase):
    def test_add_two(self):
        self.assertEqual(addtwo(2), 4)

    def test_add_two_point_two(self):
        self.assertEqual(addtwo(2.2), 4.2)

    def test_add_three(self):
        # Should fail
        self.assertEqual(addtwo(3), 6)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

LambdaTest defines a test case with three test methods, each of them exercising a test scenario for addtwo() implemented as a lambda function. The execution of the Python file lambda_unittest.py that contains LambdaTest produces the following:

Shell

```
$ python lambda_unittest.py
test_add_three (__main__.LambdaTest) ... FAIL
test_add_two (__main__.LambdaTest) ... ok
test_add_two_point_two (__main__.LambdaTest) ... ok

=====
FAIL: test_add_three (__main__.LambdaTest)
-----
Traceback (most recent call last):
  File "lambda_unittest.py", line 18, in test_add_three
    self.assertEqual(addtwo(3), 6)
AssertionError: 5 != 6

-----
Ran 3 tests in 0.001s

FAILED (failures=1)
```

As expected, we have two successful test cases and one failure for test_add_three: the result is 5, but the expected result was 6. This failure is due to an intentional mistake in the test case. Changing the expected result from 6 to 5 will satisfy all the tests for LambdaTest.

doctest

The doctest module extracts interactive Python code from docstring to execute tests. Although the syntax of Python lambda functions does not support a typical docstring, it is possible to assign a string to the __doc__ element of a named lambda:

Python

```
addtwo = lambda x: x + 2
addtwo.__doc__ = """Add 2 to a number.
>>> addtwo(2)
4
>>> addtwo(2.2)
4.2
>>> addtwo(3) # Should fail
6
"""

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

The doctest in the doc comment of lambda addtwo() describes the same test cases as in the previous section.

When you execute the tests via doctest.testmod(), you get the following:

Shell

```
$ python lambda_doctest.py
Trying:
    addtwo(2)
Expecting:
    4
ok
Trying:
    addtwo(2.2)
Expecting:
    4.2
ok
Trying:
    addtwo(3) # Should fail
Expecting:
    6
*****
File "lambda_doctest.py", line 16, in __main__.addtwo
Failed example:
    addtwo(3) # Should fail
Expected:
    6
Got:
    5
1 items had no tests:
    __main__
*****
1 items had failures:
    1 of  3 in __main__.addtwo
3 tests in 2 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
```

The failed test results from the same failure explained in the execution of the unit tests in the previous section.

You can add a docstring to a Python lambda via an assignment to `__doc__` to document a lambda function. Although possible, the Python syntax better accommodates docstring for normal functions than lambda functions.

For a comprehensive overview of unit testing in Python, you may want to refer to [Getting Started With Testing in Python](#).

Lambda Expression Abuses

Several examples in this article, if written in the context of professional Python code, would qualify as abuses.

If you find yourself trying to overcome something that a lambda expression does not support, this is probably a sign that a normal function would be better suited. The docstring for a lambda expression in the previous section is a good example. Attempting to overcome the fact that a Python lambda function does not support statements is another red flag.

The next sections illustrate a few examples of lambda usages that should be avoided. Those examples might be situations where, in the context of Python lambda, the code exhibits the following pattern:

- It doesn't follow the Python style guide (PEP 8)

- It's cumbersome and difficult to read.
- It's unnecessarily clever at the cost of difficult readability.

Raising an Exception

Trying to raise an exception in a Python lambda should make you think twice. There are some clever ways to do so, but even something like the following is better to avoid:

```
Python >>>
>>> def throw(ex): raise ex
>>> (lambda: throw(Exception('Something bad happened'))())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
  File "<stdin>", line 1, in throw
Exception: Something bad happened
```

Because a statement is not syntactically correct in a Python lambda body, the workaround in the example above consists of abstracting the statement call with a dedicated function `throw()`. Using this type of workaround should be avoided. If you encounter this type of code, you should consider refactoring the code to use a regular function.

Cryptic Style

As in any programming languages, you will find Python code that can be difficult to read because of the style used. Lambda functions, due to their conciseness, can be conducive to writing code that is difficult to read.

The following lambda example contains several bad style choices:

```
Python >>>
>>> (lambda _: list(map(lambda _: _ // 2, _)))([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

The underscore (`_`) refers to a variable that you don't need to refer to explicitly. But in this example, three `_` refer to different variables. An initial upgrade to this lambda code could be to name the variables:

```
Python >>>
>>> (lambda some_list: list(map(lambda n: n // 2,
                                some_list)))([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

Admittedly, it's still difficult to read. By still taking advantage of a lambda, a regular function would go a long way to render this code more readable, spreading the logic over a few lines and function calls:

```
Python >>>
>>> def div_items(some_list):
    div_by_two = lambda n: n // 2
    return map(div_by_two, some_list)
>>> list(div_items([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

This is still not optimal but shows you a possible path to make code, and Python lambda functions in particular, more readable. In [Alternatives to Lambdas](#), you'll learn to replace `map()` and `lambda` with list comprehensions or [generator expressions](#). This will drastically improve the readability of the code.

Python Classes

You can but should not write class methods as Python lambda functions. The following example is perfectly legal Python code but exhibits unconventional Python code relying on `lambda`. For example, instead of implementing `__str__` as a regular function, it uses a `lambda`. Similarly, brand and year are [properties](#) also implemented with `lambda` functions, instead of regular functions or decorators:

Python

```
class Car:
    """Car with methods as lambda functions."""
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    brand = property(lambda self: getattr(self, '_brand'),
                    lambda self, value: setattr(self, '_brand', value))

    year = property(lambda self: getattr(self, '_year'),
                   lambda self, value: setattr(self, '_year', value))

    __str__ = lambda self: f'{self.brand} {self.year}' # 1: error E731

    honk = lambda self: print('Honk!') # 2: error E731
```

Running a tool like [flake8](#), a style guide enforcement tool, will display the following errors for `__str__` and `honk`:

Shell

```
E731 do not assign a lambda expression, use a def
```

Although `flake8` doesn't point out an issue for the usage of the Python lambda functions in the properties, they are difficult to read and prone to error because of the usage of multiple strings like `'_brand'` and `'_year'`.

Proper implementation of `__str__` would be expected to be as follows:

Python

```
def __str__(self):
    return f'{self.brand} {self.year}'
```

`brand` would be written as follows:

Python

```
@property
def brand(self):
    return self._brand

@brand.setter
def brand(self, value):
    self._brand = value
```

As a general rule, in the context of code written in Python, prefer regular functions over lambda expressions.

Nonetheless, there are cases that benefit from lambda syntax, as you will see in the next section.

Appropriate Uses of Lambda Expressions

Lambdas in Python tend to be the subject of controversies. Some of the arguments against lambdas in Python are:

- Issues with readability
- The imposition of a functional way of thinking
- Heavy syntax with the `lambda` keyword

Despite the heated debates questioning the mere existence of this feature in Python, lambda functions have properties that sometimes provide value to the Python language and to developers.

The following examples illustrate scenarios where the use of lambda functions is not only suitable but encouraged

in Python code.

Classic Functional Constructs

Lambda functions are regularly used with the built-in functions `map()` and `filter()`, as well as `functools.reduce()`, exposed in the module `functools`. The following three examples are respective illustrations of using those functions with lambda expressions as companions:

Python

>>>

```
>>> list(map(lambda x: x.upper(), ['cat', 'dog', 'cow']))
['CAT', 'DOG', 'COW']
>>> list(filter(lambda x: 'o' in x, ['cat', 'dog', 'cow']))
['dog', 'cow']
>>> from functools import reduce
>>> reduce(lambda acc, x: f'{acc} | {x}', ['cat', 'dog', 'cow'])
'cat | dog | cow'
```

You may have to read code resembling the examples above, albeit with more relevant data. For that reason, it's important to recognize those constructs. Nevertheless, those constructs have equivalent alternatives that are considered more Pythonic. In [Alternatives to Lambdas](#), you'll learn how to convert higher-order functions and their accompanying lambdas into other more idiomatic forms.

Key Functions

Key functions in Python are higher-order functions that take a parameter key as a named argument. key receives a function that can be a lambda. This function directly influences the algorithm driven by the key function itself. Here are some key functions:

- `sort()`: list method
- `sorted()`, `min()`, `max()`: built-in functions
- `nlargest()` and `nsmallest()`: in the Heap queue algorithm module `heapq`

Imagine that you want to sort a list of IDs represented as strings. Each ID is the [concatenation](#) of the string `id` and a number. Sorting this list with the built-in function `sorted()`, by default, uses a lexicographic order as the elements in the list are strings.

To influence the sorting execution, you can assign a lambda to the named argument `key`, such that the sorting will use the number associated with the ID:

Python

>>>

```
>>> ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
>>> print(sorted(ids)) # Lexicographic sort
['id1', 'id100', 'id2', 'id22', 'id3', 'id30']
>>> sorted_ids = sorted(ids, key=lambda x: int(x[2:])) # Integer sort
>>> print(sorted_ids)
['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
```

UI Frameworks

UI frameworks like [Tkinter](#), [wxPython](#), or .NET Windows Forms with [IronPython](#) take advantage of lambda functions for mapping actions in response to UI events.

The naive Tkinter program below demonstrates the usage of a lambda assigned to the command of the Reverse button:

Python

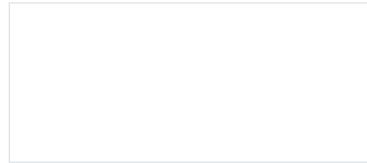
```

import tkinter as tk
import sys

window = tk.Tk()
window.grid_columnconfigure(0, weight=1)
window.title("Lambda")
window.geometry("300x100")
label = tk.Label(window, text="Lambda Calculus")
label.grid(column=0, row=0)
button = tk.Button(
    window,
    text="Reverse",
    command=lambda: label.configure(text=label.cget("text")[::-1]),
)
button.grid(column=0, row=1)
window.mainloop()

```

Clicking the button *Reverse* fires an event that triggers the lambda function, changing the label from *Lambda Calculus* to *suluclaC adbmaL**:



Both wxPython and IronPython on the .NET platform share a similar approach for handling events. Note that `lambda` is one way to handle firing events, but a function may be used for the same purpose. It ends up being self-contained and less verbose to use a `lambda` when the amount of code needed is very short.

To explore wxPython, check out [How to Build a Python GUI Application With wxPython](#).

Python Interpreter

When you're playing with Python code in the interactive interpreter, Python lambda functions are often a blessing. It's easy to craft a quick one-liner function to explore some snippets of code that will never see the light of day outside of the interpreter. The lambdas written in the interpreter, for the sake of speedy discovery, are like scrap paper that you can throw away after use.

timeit

In the same spirit as the experimentation in the Python interpreter, the module `timeit` provides functions to time small code fragments. `timeit.timeit()` in particular can be called directly, passing some Python code in a string. Here's an example:

Python	>>>
>>> from timeit import timeit >>> timeit("factorial(999)", "from math import factorial", number=10) 0.0013087529951008037	

When the statement is passed as a string, `timeit()` needs the full context. In the example above, this is provided by the second argument that sets up the environment needed by the main function to be timed. Not doing so would raise a `NameError` exception.

Another approach is to use a `lambda`:

Python	>>>
>>> from math import factorial >>> timeit(lambda: factorial(999), number=10) 0.0012704220062005334	

This solution is cleaner, more readable, and quicker to type in the interpreter. Although the execution time was slightly less for the `lambda` version, executing the functions again may show a slight advantage for the `string` version. The execution time of the setup is excluded from the overall execution time and shouldn't have any impact on the result.

Monkey Patching

For testing, it's sometimes necessary to rely on repeatable results, even if during the normal execution of a given software, the corresponding results are expected to differ, or even be totally random.

Let's say you want to test a function that, at runtime, handles random values. But, during the testing execution, you need to assert against predictable values in a repeatable manner. The following example shows how, with a `lambda` function, monkey patching can help you:

Python

```
from contextlib import contextmanager
import secrets

def gen_token():
    """Generate a random token."""
    return f'TOKEN_{secrets.token_hex(8)}'

@contextmanager
def mock_token():
    """Context manager to monkey patch the secrets.token_hex
    function during testing.
    """
    default_token_hex = secrets.token_hex
    secrets.token_hex = lambda _: 'feedfacecafebeef'
    yield
    secrets.token_hex = default_token_hex

def test_gen_key():
    """Test the random token."""
    with mock_token():
        assert gen_token() == f"TOKEN_{'feedfacecafebeef'}"

test_gen_key()
```

A context manager helps with insulating the operation of monkey patching a function from the standard library (`secrets`, in this example). The `lambda` function assigned to `secrets.token_hex()` substitutes the default behavior by returning a static value.

This allows testing any function depending on `token_hex()` in a predictable fashion. Prior to exiting from the context manager, the default behavior of `token_hex()` is reestablished to eliminate any unexpected side effects that would affect other areas of the testing that may depend on the default behavior of `token_hex()`.

Unit test frameworks like `unittest` and `pytest` take this concept to a higher level of sophistication.

With `pytest`, still using a `lambda` function, the same example becomes more elegant and concise :

Python

```
import secrets

def gen_token():
    return f'TOKEN_{secrets.token_hex(8)}'

def test_gen_key(monkeypatch):
    monkeypatch.setattr('secrets.token_hex', lambda _: 'feedfacecafebeef')
    assert gen_token() == f"TOKEN_{'feedfacecafebeef'}"
```

With the [pytest monkeypatch fixture](#), `secrets.token_hex()` is overwritten with a lambda that will return a deterministic value, `feedfacecafebeef`, allowing to validate the test. The `pytest monkeypatch` fixture allows you to control the scope of the override. In the example above, invoking `secrets.token_hex()` in subsequent tests, without using monkey patching, would execute the normal implementation of this function.

Executing the pytest test gives the following result:

Shell

```
$ pytest test_token.py -v
=====
platform linux -- Python 3.7.2, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
cachedir: .pytest_cache
rootdir: /home/andre/AB/tools/bpython, inifile:
collected 1 item

test_token.py::test_gen_key PASSED [100%]

=====
1 passed in 0.01 seconds =====
```

The test passes as we validated that the `gen_token()` was exercised, and the results were the expected ones in the context of the test.

Alternatives to Lambdas

While there are great reasons to use `lambda`, there are instances where its use is frowned upon. So what are the alternatives?

Higher-order functions like `map()`, `filter()`, and `functools.reduce()` can be converted to more elegant forms with slight twists of creativity, in particular with list comprehensions or generator expressions.

To learn more about list comprehensions, check out [Using List Comprehensions Effectively](#). To learn more about generator expressions, check out [Python Generators 101](#).

Map

The built-in function `map()` takes a function as a first argument and applies it to each of the elements of its second argument, an **iterable**. Examples of iterables are strings, lists, and tuples. For more information on iterables and iterators, check out [Iterables and Iterators](#).

`map()` returns an iterator corresponding to the transformed collection. As an example, if you wanted to transform a list of strings to a new list with each string capitalized, you could use `map()`, as follows:

```
Python >>>
>>> list(map(lambda x: x.capitalize(), ['cat', 'dog', 'cow']))
['Cat', 'Dog', 'Cow']
```

You need to invoke `list()` to convert the iterator returned by `map()` into an expanded list that can be displayed in the Python shell interpreter.

Using a list comprehension eliminates the need for defining and invoking the `lambda` function:

```
Python >>>
>>> [x.capitalize() for x in ['cat', 'dog', 'cow']]
['Cat', 'Dog', 'Cow']
```

Filter

The built-in function `filter()`, another classic functional construct, can be converted into a list comprehension. It takes a [predicate](#) as a first argument and an iterable as a second argument. It builds an iterator containing all the elements of the initial collection that satisfies the predicate function. Here's an example that filters all the even numbers in a given list of integers.

numbers in a given list of integers.

Python

>>>

```
>>> even = lambda x: x%2 == 0
>>> list(filter(even, range(11)))
[0, 2, 4, 6, 8, 10]
```

Note that `filter()` returns an iterator, hence the need to invoke the built-in type `list` that constructs a list given an iterator.

The implementation leveraging the list comprehension construct gives the following:

Python

>>>

```
>>> [x for x in range(11) if x%2 == 0]
[0, 2, 4, 6, 8, 10]
```

Reduce

Since Python 3, `reduce()` has gone from a built-in function to a `functools` module function. As `map()` and `filter()`, its first two arguments are respectively a function and an iterable. It may also take an initializer as a third argument that is used as the initial value of the resulting accumulator. For each element of the iterable, `reduce()` applies the function and accumulates the result that is returned when the iterable is exhausted.

To apply `reduce()` to a list of pairs and calculate the sum of the first item of each pair, you could write this:

Python

>>>

```
>>> import functools
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> functools.reduce(lambda acc, pair: acc + pair[0], pairs, 0)
6
```

A more idiomatic approach using a [generator expression](#), as an argument to `sum()` in the example, is the following:

Python

>>>

```
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> sum(x[0] for x in pairs)
6
```

A slightly different and possibly cleaner solution removes the need to explicitly access the first element of the pair and instead use unpacking:

Python

>>>

```
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> sum(x for x, _ in pairs)
6
```

The use of underscore (`_`) is a Python convention indicating that you can ignore the second value of the pair.

`sum()` takes a unique argument, so the generator expression does not need to be in parentheses.

Are Lambdas Pythonic or Not?

[PEP 8](#), which is the style guide for Python code, reads:

Always use a `def` statement instead of an assignment statement that binds a `lambda` expression directly to an identifier. ([Source](#))

This strongly discourages using `lambda` bound to an identifier, mainly where functions should be used and have more benefits. PEP 8 does not mention other usages of `lambda`. As you have seen in the previous sections, `lambda` functions may certainly have good uses, although they are limited.

A possible way to answer the question is that lambda functions are perfectly Pythonic if there is nothing more Pythonic available. I'm staying away from defining what "Pythonic" means, leaving you with the definition that best suits your mindset, as well as your personal or your team's coding style.

Beyond the narrow scope of Python lambda, [How to Write Beautiful Python Code With PEP 8](#) is a great resource that you may want to check out regarding code style in Python.

Conclusion

You now know how to use Python lambda functions and can:

- Write Python lambdas and use anonymous functions
- Choose wisely between lambdas or normal Python functions
- Avoid excessive use of lambdas
- Use lambdas with higher-order functions or Python key functions

If you have a penchant for mathematics, you may have some fun exploring the fascinating world of [lambda calculus](#).

Python lambdas are like salt. A pinch in your spam, ham, and eggs will enhance the flavors, but too much will spoil the dish.

 **Take the Quiz:** Test your knowledge with our interactive "Python Lambda Functions" quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

Note: The Python programming language, named after Monty Python, prefers to use `spam`, `ham`, and `eggs` as metasyntactic variables, instead of the traditional `foo`, `bar`, and `baz`.

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Use Python Lambda Functions](#)

About Andre Burgaud

Andre is a seasoned software engineer passionate about technology and programming languages, in particular, Python.

[» More about Andre](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

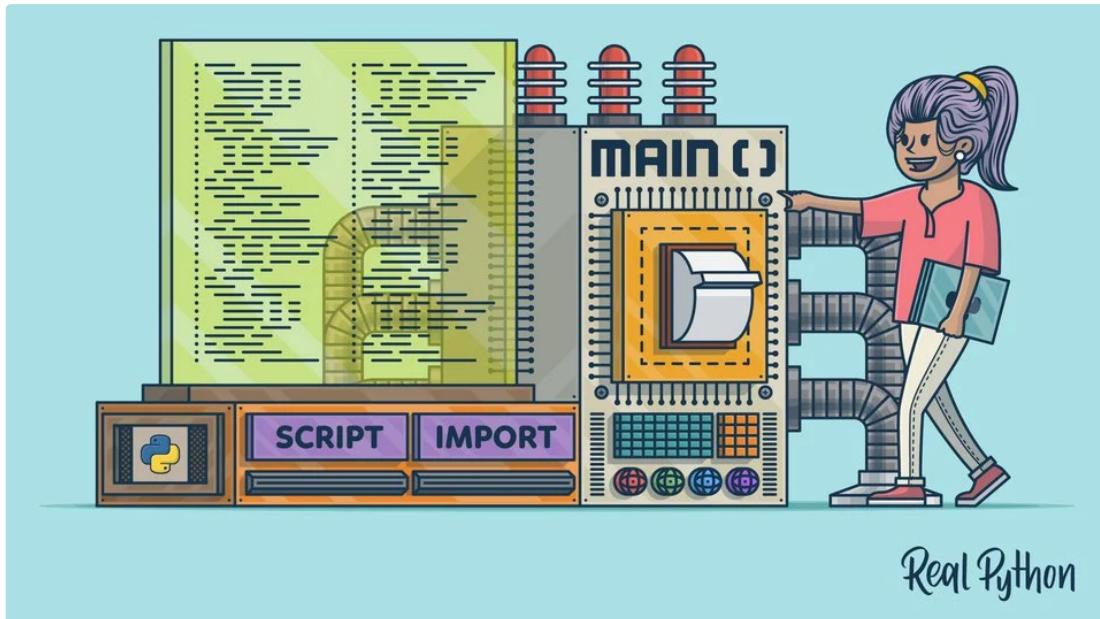
Jon

Joanna

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [python](#)

Recommended Video Course: [How to Use Python Lambda Functions](#)



Defining Main Functions in Python

by [Bryan Weber](#) 27 Comments [best-practices](#) [intermediate](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [A Basic Python main\(\)](#)
- [Execution Modes in Python](#)
 - [Executing From the Command Line](#)
 - [Importing Into a Module or the Interactive Interpreter](#)
- [Best Practices for Python Main Functions](#)
 - [Put Most Code Into a Function or Class](#)
 - [Use if __name__ == " main " to Control the Execution of Your Code](#)
 - [Create a Function Called main\(\) to Contain the Code You Want to Run](#)
 - [Call Other Functions From main\(\)](#)
 - [Summary of Python Main Function Best Practices](#)
- [Conclusion](#)

 [blackfire.io](#)
Profile & Optimize Python Apps Performance



Now available as
Public Beta
Sign-up for free and
install in minutes!

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Defining Main Functions in Python](#)

Many programming languages have a special function that is automatically executed when an operating system starts to run a program. This function is usually called `main()` and must have a specific return type and arguments according to the language standard. On the other hand, the Python interpreter executes scripts starting at the top of the file, and there is no specific function that Python automatically executes.

Nevertheless, having a defined starting point for the execution of a program is useful for understanding how a program works. Python programmers have come up with several conventions to define this starting point.

By the end of this article, you'll understand:

- What the special `__name__` variable is and how Python defines it
- Why you would want to use a `main()` in Python

- What conventions there are for defining `main()` in Python
- What the best-practices are for what code to put into your `main()`

Free Bonus: [Click here to get access to a chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

A Basic Python `main()`

In some Python scripts, you may see a function definition and a conditional statement that looks like the example below:

```
Python
def main():
    print("Hello World!")

if __name__ == "__main__":
    main()
```

In this code, there is a function called `main()` that prints the phrase `Hello World!` when the Python interpreter executes it. There is also a conditional (or `if`) statement that checks the value of `__name__` and compares it to the string `"__main__"`. When the `if` statement evaluates to True, the Python interpreter executes `main()`. You can read more about conditional statements in [Conditional Statements in Python](#).

This code pattern is quite common in Python files that you want to be **executed as a script** and **imported in another module**. To help understand how this code will execute, you should first understand how the Python interpreter sets `__name__` depending on how the code is being executed.

Execution Modes in Python

There are two primary ways that you can instruct the Python interpreter to execute or use code:

1. You can execute the Python file as a **script** using the command line.
2. You can **import** the code from one Python file into another file or into the interactive interpreter.

You can read a lot more about these approaches in [How to Run Your Python Scripts](#). No matter which way of running your code you're using, Python defines a special variable called `__name__` that contains a string whose value depends on how the code is being used.

We'll use this example file, saved as `execution_methods.py`, to explore how the behavior of the code changes depending on the context:

```
Python
print("This is my file to test Python's execution methods.")
print("The variable __name__ tells me which context this file is running in.")
print("The value of __name__ is:", repr(__name__))
```

In this file, there are three calls to `print()` defined. The first two print some introductory phrases. The third `print()` will first print the phrase `The value of __name__ is`, and then it will print the representation of the `__name__` variable using Python's built-in `repr()`.

In Python, `repr()` displays the printable representation of an object. This example uses `repr()` to emphasize that the value of `__name__` is a string. You can read more about `repr()` in the [Python documentation](#).

You'll see the words **file**, **module**, and **script** used throughout this article. Practically, there isn't much difference between them. However, there are slight differences in meaning that emphasize the purpose of a piece of code:

1. **File:** Typically, a Python file is any file that contains code. Most Python files have the extension `.py`.
2. **Script:** A Python script is a file that you intend to execute from the command line to accomplish a task.
3. **Module:** A Python module is a file that you intend to import from within another module or a script, or from the interactive interpreter. You can read more about modules in the [Python documentation](#).

This distinction is also discussed in [How to Run Your Python Scripts](#).

Executing From the Command Line

In this approach, you want to execute your Python script from the command line.

When you execute a script, you will not be able to interactively define the code that the Python interpreter is executing. The details of how you can execute Python from your command line are not that important for the purpose of this article, but you can expand the box below to read more about the differences between the command line on Windows, Linux, and macOS.

Now you should execute the `execution_methods.py` script from the command line, as shown below:

Shell

```
$ python3 execution_methods.py
This is my file to test Python's execution methods.
The variable __name__ tells me which context this file is running in.
The value of __name__ is: '__main__'
```

In this example, you can see that `__name__` has the value '`__main__`', where the quote symbols ('') tell you that the value has the string type.

Remember that, in Python, there is no difference between strings defined with single quotes ('') and double quotes (""). You can read more about defining strings in [Basic Data Types in Python](#).

You will find identical output if you include a [shebang line](#) in your script and execute it directly (`./execution_methods.py`), or use the `%run` magic in IPython or Jupyter Notebooks.

You may also see Python scripts executed from within packages by adding the `-m` argument to the command. Most often, you will see this recommended when you're using pip: `python3 -m pip install package_name`.

Adding the `-m` argument runs the code in the `__main__.py` module of a package. You can find more information about the `__main__.py` file in [How to Publish an Open-Source Python Package to PyPI](#).

In all three of these cases, `__name__` has the same value: the string '`__main__`'.

Technical detail: The Python documentation defines specifically when `__name__` will have the value '`__main__`':

A module's `__name__` is set equal to '`__main__`' when read from standard input, a script, or from an interactive prompt. ([Source](#))

`__name__` is stored in the global namespace of the module along with the `__doc__`, `__package__`, and other attributes. You can read more about these attributes in the [Python Data Model documentation](#) and, specifically for modules and packages, in the [Python Import documentation](#).

Importing Into a Module or the Interactive Interpreter

Now let's take a look at the second way that the Python interpreter will execute your code: imports. When you are developing a module or script, you will most likely want to take advantage of modules that someone else has already built, which you can do with the `import` keyword.

During the import process, Python executes the statements defined in the specified module (but only the *first* time you import a module). To demonstrate the results of importing your `execution_methods.py` file, start the interactive Python interpreter and then import your `execution_methods.py` file:

Python

>>>

```
>>> import execution_methods
This is my file to test Python's execution methods.
The variable __name__ tells me which context this file is running in.
The value of __name__ is: 'execution_methods'
```

In this code output, you can see that the Python interpreter executes the three calls to `print()`. The first two lines of output are exactly the same as when you executed the file as a script on the command line because there are no variables in either of the first two lines. However, there is a difference in the output from the third `print()`.

When the Python interpreter imports code, the value of `__name__` is set to be the same as the name of the module that is being imported. You can see this in the third line of output above. `__name__` has the value 'execution_methods', which is the name of the .py file that Python is importing from.

Note that if you import the module again without quitting Python, there will be no output.

Note: For more information on how importing works in Python, check out [Python import: Advanced Techniques and Tips](#) as well as [Absolute vs Relative Imports in Python](#).

Best Practices for Python Main Functions

Now that you can see the differences in how Python handles its different execution modes, it's useful for you to know some best practices to use. These will apply whenever you want to write code that you can run as a script *and* import in another module or an interactive session.

You will learn about four best practices to make sure that your code can serve a dual purpose:

1. Put most code into a function or class.
2. Use `__name__` to control execution of your code.
3. Create a function called `main()` to contain the code you want to run.
4. Call other functions from `main()`.

Put Most Code Into a Function or Class

Remember that the Python interpreter executes all the code in a module when it imports the module. Sometimes the code you write will have side effects that you want the user to control, such as:

- Running a computation that takes a long time
- Writing to a file on the disk
- Printing information that would clutter the user's terminal

In these cases, you want the user to control triggering the execution of this code, rather than letting the Python interpreter execute the code when it imports your module.

Therefore, the best practice is to **include most code inside a function or a class**. This is because when the Python interpreter encounters the `def` or `class` keywords, it only stores those definitions for later use and doesn't actually execute them until you tell it to.

Save the code below to a file called `best_practices.py` to demonstrate this idea:

Python

```
1 from time import sleep
2
3 print("This is my file to demonstrate best practices.")
4
5 def process_data(data):
6     print("Beginning data processing...")
7     modified_data = data + " that has been modified"
8     sleep(3)
9     print("Data processing finished.")
10    return modified_data
```

In this code, you first import `sleep()` from the [time module](#).

`sleep()` pauses the interpreter for however many seconds you give as an argument and will produce a function that takes a long time to run for this example. Next, you use `print()` to print a sentence describing the purpose of this code.

Then, you define a function called `process_data()` that does five things:

1. Prints some output to tell the user that the data processing is starting
2. Modifies the input data
3. Pauses the execution for three seconds using `sleep()`
4. Prints some output to tell the user that the processing is finished
5. Returns the modified data

Execute the Best Practices File on the Command Line

Now, what will happen when you execute this file as a script on the command line?

The Python interpreter will execute the `from time import sleep` and `print()` lines that are outside the function definition, then it will create the definition of the function called `process_data()`. Then, the script will exit without doing anything further, because the script does not have any code that executes `process_data()`.

The code block below shows the result of running this file as a script:

Shell

```
$ python3 best_practices.py  
This is my file to demonstrate best practices.
```

The output that we can see here is the result of the first `print()`. Notice that importing from `time` and defining `process_data()` produce no output. Specifically, the outputs of the calls to `print()` that are inside the definition of `process_data()` are not printed!

Import the Best Practices File in Another Module or the Interactive Interpreter

When you import this file in an interactive session (or another module), the Python interpreter will perform exactly the same steps as when it executes file as a script.

Once the Python interpreter imports the file, you can use any variables, classes, or functions defined in the module you've imported. To demonstrate this, we will use the interactive Python interpreter. Start the interactive interpreter and then type `import best_practices`:

Python

>>>

```
>>> import best_practices  
This is my file to demonstrate best practices.
```

The only output from importing the `best_practices.py` file is from the first `print()` call defined outside `process_data()`. Importing from `time` and defining `process_data()` produce no output, just like when you executed the code from the command line.

Use `if __name__ == "__main__"` to Control the Execution of Your Code

What if you want `process_data()` to execute when you run the script from the command line but not when the Python interpreter imports the file?

You can **use the `if __name__ == "__main__"` idiom to determine the execution context** and conditionally run `process_data()` only when `__name__` is equal to "`__main__`". Add the code below to the bottom of your `best_practices.py` file:

Python

```
11 | if __name__ == "__main__":  
12 |     data = "My data read from the Web"  
13 |     print(data)
```

```
14     modified_data = process_data(data)
15     print(modified_data)
```

In this code, you've added a conditional statement that checks the value of `__name__`. This conditional will evaluate to `True` when `__name__` is equal to the string "`__main__`". Remember that the special value of "`__main__`" for the `__name__` variable means the Python interpreter is executing your script and not importing it.

Inside the conditional block, you have added four lines of code (lines 12, 13, 14, and 15):

- **Lines 12 and 13:** You are creating a variable `data` that stores the data you've acquired from the Web and printing it.
- **Line 14:** You are processing the data.
- **Line 15:** You are printing the modified data.

Now, run your `best_practices.py` script from the command line to see how the output will change:

Shell

```
$ python3 best_practices.py
This is my file to demonstrate best practices.
My data read from the Web
Beginning data processing...
Data processing finished.
My data read from the Web that has been modified
```

First, the output shows the result of the `print()` call outside of `process_data()`.

After that, the value of `data` is printed. This happened because the variable `__name__` has the value "`__main__`" when the Python interpreter executes the file as a script, so the conditional statement evaluated to `True`.

Next, your script called `process_data()` and passed `data` in for modification. When `process_data()` executes, it prints some status messages to the output. Finally, the value of `modified_data` is printed.

Now you should check what happens when you import the `best_practices.py` file from the interactive interpreter (or another module). The example below demonstrates this situation:

Python

>>>

```
>>> import best_practices
This is my file to demonstrate best practices.
```

Notice that you get the same behavior as before you added the conditional statement to the end of the file! This is because the `__name__` variable had the value "`best_practices`", so Python did not execute the code inside the block, including `process_data()`, because the conditional statement evaluated to `False`.

Create a Function Called `main()` to Contain the Code You Want to Run

Now you are able to write Python code that can be run from the command line as a script and imported without unwanted side effects. Next, you are going to learn about how to write your code to make it easy for other Python programmers to follow what you mean.

Many languages, such as C, C++, Java, and several others, define a special function that must be called `main()` that the operating system automatically calls when it executes the compiled program. This function is often called the **entry point** because it is where execution enters the program.

By contrast, Python does not have a special function that serves as the entry point to a script. You can actually give the entry point function in a Python script any name you want!

Although Python does not assign any significance to a function named `main()`, the best practice is to **name the entry point function `main()` anyways**. That way, any other programmers who read your script immediately know that this function is the starting point of the code that accomplishes the primary task of the script.

In addition, `main()` should contain any code that you want to run when the Python interpreter executes the file. This is better than putting the code directly into the conditional block because a user can reuse `main()` if they import your module.

Change the best_practices.py file so that it looks like the code below:

Python

```
1 from time import sleep
2
3 print("This is my file to demonstrate best practices.")
4
5 def process_data(data):
6     print("Beginning data processing...")
7     modified_data = data + " that has been modified"
8     sleep(3)
9     print("Data processing finished.")
10    return modified_data
11
12 def main():
13     data = "My data read from the Web"
14     print(data)
15     modified_data = process_data(data)
16     print(modified_data)
17
18 if __name__ == "__main__":
19     main()
```

In this example, you added the definition of `main()` that includes the code that was previously inside the conditional block. Then, you changed the conditional block so that it executes `main()`. If you run this code as a script or import it, you will get the same output as in the previous section.

Call Other Functions From main()

Another common practice in Python is to **have `main()` execute other functions**, rather than including the task-accomplishing code in `main()`. This is especially useful when you can compose your overall task from several smaller sub-tasks that can execute independently.

For example, you may have a script that does the following:

1. Reads a data file from a source that could be a database, a file on the disk, or a web API
2. Processes the data
3. Writes the processed data to another location

If you implement each of these sub-tasks in separate functions, then it is easy for a you (or another user) to re-use a few of the steps and ignore the ones you don't want. Then you can create a default workflow in `main()`, and you can have the best of both worlds.

Whether to apply this practice to your code is a judgment call on your part. Splitting the work into several functions makes reuse easier but increases the difficulty for someone else trying to interpret your code because they have to follow several jumps in the flow of the program.

Modify your best_practices.py file so that it looks like the code below:

Python

```
1 from time import sleep
2
3 print("This is my file to demonstrate best practices.")
4
5 def process_data(data):
6     print("Beginning data processing...")
7     modified_data = data + " that has been modified"
8     sleep(3)
9     print("Data processing finished.")
10    return modified_data
11
12 def read_data_from_web():
13     print("Reading data from the Web")
14     data = "Data from the web"
15     return data
16
```

```

17 def write_data_to_database(data):
18     print("Writing data to a database")
19     print(data)
20
21 def main():
22     data = read_data_from_web()
23     modified_data = process_data(data)
24     write_data_to_database(modified_data)
25
26 if __name__ == "__main__":
27     main()

```

In this example code, the first 10 lines of the file have the same content that they had before. The second function definition on line 12 creates and returns some sample data, and the third function definition on line 17 simulates writing the modified data to a database.

On line 21, `main()` is defined. In this example, you have modified `main()` so that it calls the data reading, data processing, and data writing functions in turn.

First, the data is created from `read_data_from_web()`. This data is passed to `process_data()`, which returns the `modified_data`. Finally, `modified_data` is passed into `write_data_to_database()`.

The last two lines of the script are the conditional block that checks `__name__` and runs `main()` if the `if` statement is True.

Now, you can run the whole processing pipeline from the command line, as shown below:

Shell

```

$ python3 best_practices.py
This is my file to demonstrate best practices.
Reading data from the Web
Beginning data processing...
Data processing finished.
Writing processed data to a database
Data from the web that has been modified

```

In the output from this execution, you can see that the Python interpreter executed `main()`, which executed `read_data_from_web()`, `process_data()`, and `write_data_to_database()`. However, you can also import the `best_practices.py` file and re-use `process_data()` for a different input data source, as shown below:

Python

>>>

```

>>> import best_practices as bp
This is my file to demonstrate best practices.
>>> data = "Data from a file"
>>> modified_data = bp.process_data(data)
Beginning data processing...
Data processing finished.
>>> bp.write_data_to_database(modified_data)
Writing processed data to a database
Data from a file that has been modified

```

In this example, you imported `best_practices` and shortened the name to `bp` for this code.

The import process caused the Python interpreter to execute all of the lines of code in the `best_practices.py` file, so the output shows the line explaining the purpose of the file.

Then, you stored data from a file in `data` instead of reading the data from the Web. Then, you reused `process_data()` and `write_data_to_database()` from the `best_practices.py` file. In this case, you took advantage of reusing your code instead of defining all of the logic in `main()`.

Summary of Python Main Function Best Practices

Here are four key best practices about `main()` in Python that you just saw:

¹ Put code that takes a long time to run or has other effects on the computer in a function or class, so you can

1. If a code that takes a long time to run or has other effects on the computer in a function or class, so you can control exactly when that code is executed.

2. Use the different values of `__name__` to determine the context and change the behavior of your code with a conditional statement.
3. You should name your entry point function `main()` in order to communicate the intention of the function, even though Python does not assign any special significance to a function named `main()`.
4. If you want to reuse functionality from your code, define the logic in functions outside `main()` and call those functions within `main()`.

Conclusion

Congratulations! You now know how to create Python `main()` functions.

You learned the following:

- Knowing the value of the `__name__` variable is important to write code that serves the dual purpose of executable script and importable module.
- `__name__` takes on different values depending on how you executed your Python file. `__name__` will be equal to:
 - "`__main__`" when the file is executed from the command line or with `python -m` (to execute a package's `__main__.py` file)
 - The name of the module, if the module is being imported
- Python programmers have developed a set of good practices to use when you want to develop reusable code.

Now you're ready to go write some awesome Python `main()` function code!

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Defining Main Functions in Python](#)

About Bryan Weber

Bryan is a mechanical engineering professor and a core developer of Cantera, the open-source platform for thermodynamics, chemical kinetics, and transport.

[» More about Bryan](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Brad](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#)

Recommended Video Course: [Defining Main Functions in Python](#)



Object-Oriented Programming (OOP) in Python 3

by [David Amos](#) · Jul 06, 2020 · [95 Comments](#) · [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Is Object-Oriented Programming in Python?](#)
- [Define a Class in Python](#)
 - [Classes vs Instances](#)
 - [How to Define a Class](#)
- [Instantiate an Object in Python](#)
 - [Class and Instance Attributes](#)
 - [Instance Methods](#)
 - [Check Your Understanding](#)
- [Inherit From Other Classes in Python](#)
 - [Dog Park Example](#)
 - [Parent Classes vs Child Classes](#)
 - [Extend the Functionality of a Parent Class](#)
 - [Check Your Understanding](#)
- [Conclusion](#)



Master Real-World Python Skills
With a Community of Experts

Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons

[Watch Now »](#)

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Intro to Object-Oriented Programming \(OOP\) in Python](#)

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**. In this tutorial, you'll learn the basics of object-oriented programming in Python.

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or preprocessed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

In this tutorial, you'll learn how to:

- Create a **class**, which is like a blueprint for creating an object
- Use classes to **create new objects**
- Model systems with **class inheritance**

Note: This tutorial is adapted from the chapter “Object-Oriented Programming (OOP)” in *Python Basics: A Practical Introduction to Python 3*.

The book uses Python’s built-in **IDLE** editor to create and edit Python files and interact with the Python shell, so you will see occasional references to IDLE throughout this tutorial. However, you should have no problems running the example code from the editor and environment of your choice.

Free Bonus: [Click here to get access to a free Python OOP Cheat Sheet](#) that points you to the best tutorials, videos, and books to learn more about Object-Oriented Programming with Python.

What Is Object-Oriented Programming in Python?

Object-oriented programming is a [programming paradigm](#) that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**.

For instance, an object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

Another common programming paradigm is **procedural programming**, which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, that flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

Define a Class in Python

Primitive [data structures](#)—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

For example, let’s say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a [list](#):

Python

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where the `kirk` list is declared, will you remember that the element with index `0` is the employee’s name?

Second, it can introduce errors if not every employee has the same number of elements in the list. In the `mccoy` list above, the age is missing, so `mccoy[1]` will return “Chief Medical Officer” instead of Dr. McCoy’s age.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

Classes vs Instances

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this tutorial, you'll create a `Dog` class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how something should be defined. It doesn't actually contain any data. The `Dog` class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an **instance** is an object that is built from a class and contains real data. An instance of the `Dog` class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information. Just like many people can fill out the same form with their own unique information, many instances can be created from a single class.

How to Define a Class

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a `Dog` class:

Python

```
class Dog:  
    pass
```

The body of the `Dog` class consists of a single statement: the `pass` keyword. `pass` is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Note: Python class names are written in CapitalizedWords notation by convention. For example, a class for a specific breed of dog like the Jack Russell Terrier would be written as `JackRussellTerrier`.

The `Dog` class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all `Dog` objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.

The properties that all `Dog` objects must have are defined in a method called `__init__()`. Every time a new `Dog` object is created, `__init__()` sets the initial **state** of the object by assigning the values of the object's properties. That is, `__init__()` initializes each new instance of the class.

You can give `__init__()` any number of parameters, but the first parameter will always be a variable called `self`. When a new class instance is created, the instance is automatically passed to the `self` parameter in `__init__()` so that new **attributes** can be defined on the object.

Let's update the `Dog` class with an `__init__()` method that creates `.name` and `.age` attributes:

Python

```
class Dog:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Notice that the `__init__()` method's signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the `__init__()` method belongs to the `Dog` class.

In the body of `__init__()`, there are two statements using the `self` variable:

1. `self.name = name` creates an attribute called `name` and assigns to it the value of the `name` parameter.
2. `self.age = age` creates an attribute called `age` and assigns to it the value of the `age` parameter.

Attributes created in `__init__()` are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All `Dog` objects have a `name` and an `age`, but the values for the `name` and `age` attributes will vary depending on the `Dog` instance.

On the other hand, **class attributes** are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `__init__()`.

For example, the following `Dog` class has a class attribute called `species` with the value `"Canis familiaris"`:

Python

```
class Dog:  
    # Class attribute  
    species = "Canis familiaris"  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that we have a `Dog` class, let's create some dogs!

Instantiate an Object in Python

Open IDLE's interactive window and type the following:

Python

>>>

```
>>> class Dog:  
...     pass
```

This creates a new `Dog` class with no attributes or methods.

Creating a new object from a class is called **instantiating** an object. You can instantiate a new `Dog` object by typing the name of the class, followed by opening and closing parentheses:

Python

>>>

```
>>> Dog()  
<__main__.Dog object at 0x106702d30>
```

You now have a new `Dog` object at `0x106702d30`. This funny-looking string of letters and numbers is a **memory address** that indicates where the `Dog` object is stored in your computer's memory. Note that the address you see on your screen will be different.

Now instantiate a second `Dog` object:

Python

>>>

```
>>> Dog()  
<__main__.Dog object at 0x0004ccc90>
```

The new `Dog` instance is located at a different memory address. That's because it's an entirely new instance and is completely unique from the first `Dog` object that you instantiated.

To see this another way, type the following:

```
Python >>>
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

In this code, you create two new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.

Class and Instance Attributes

Now create a new Dog class with a class attribute called .species and two instance attributes called .name and .age:

```
Python >>>
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
```

To instantiate objects of this Dog class, you need to provide values for the name and age. If you don't, then Python raises a `TypeError`:

```
Python >>>
>>> Dog()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    Dog()
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

To pass arguments to the name and age parameters, put values into the parentheses after the class name:

```
Python >>>
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

This creates two new Dog instances—one for a nine-year-old dog named Buddy and one for a four-year-old dog named Miles.

The Dog class's `__init__()` method has three parameters, so why are only two arguments passed to it in the example?

When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of `__init__()`. This essentially removes the `self` parameter, so you only need to worry about the name and age parameters.

After you create the Dog instances, you can access their instance attributes using **dot notation**:

```
Python >>>
>>> buddy.name
'Buddy'
>>> buddy.age
9

>>> miles.name
'Miles'
>>> miles.age
4
```

You can access class attributes the same way.

Python

>>>

```
>>> buddy.species  
'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, and .age attributes, so you can use those attributes with confidence knowing that they will always return a value.

Although the attributes are guaranteed to exist, their values *can* be changed dynamically:

Python

>>>

```
>>> buddy.age = 10  
>>> buddy.age  
10  
  
>>> miles.species = "Felis silvestris"  
>>> miles.species  
'Felis silvestris'
```

In this example, you change the .age attribute of the buddy object to 10. Then you change the .species attribute of the miles object to "Felis silvestris", which is a species of cat. That makes Miles a pretty strange dog, but it is valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if it can be altered dynamically. For example, lists and [dictionaries](#) are mutable, but strings and tuples are immutable.

Instance Methods

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like `__init__()`, an instance method's first parameter is always `self`.

Open a new editor window in IDLE and type in the following Dog class:

Python

```
class Dog:  
    species = "Canis familiaris"  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    # Instance method  
    def description(self):  
        return f"{self.name} is {self.age} years old"  
  
    # Another instance method  
    def speak(self, sound):  
        return f"{self.name} says {sound}"
```

This Dog class has two instance methods:

1. `.description()` returns a string displaying the name and age of the dog.
2. `.speak()` has one parameter called `sound` and returns a string containing the dog's name and the sound the dog makes.

Save the modified Dog class to a file called dog.py and press F5 to run the program. Then open the interactive window and type the following to see your instance methods in action:

```
Python >>>
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

In the above Dog class, .description() returns a string containing information about the Dog instance miles. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class. However, .description() isn't the most [Pythonic](#) way of doing this.

When you create a list object, you can use print() to display a string that looks like the list:

```
Python >>>
>>> names = ["Fletcher", "David", "Dan"]
>>> print(names)
['Fletcher', 'David', 'Dan']
```

Let's see what happens when you print() the miles object:

```
Python >>>
>>> print(miles)
<__main__.Dog object at 0x00aeef70>
```

When you print(miles), you get a cryptic looking message telling you that miles is a Dog object at the memory address 0x00aeef70. This message isn't very helpful. You can change what gets printed by defining a special instance method called __str__().

In the editor window, change the name of the Dog class's .description() method to __str__():

```
Python
class Dog:
    # Leave other parts of Dog class as-is

    # Replace .description() with __str__()
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Save the file and press F5. Now, when you print(miles), you get a much friendlier output:

```
Python >>>
>>> miles = Dog("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

Methods like __init__() and __str__() are called **dunder methods** because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Although too advanced a topic for a beginning Python book, understanding dunder methods is an important part of mastering object-oriented programming in Python.

In the next section, you'll see how to take your knowledge one step further and create classes from other classes.

Check Your Understanding

Expand the block below to check your understanding:

You can expand the block below to see a solution:

Solution: Create a Car Class

Show/Hide

When you're ready, you can move on to the next section.

Inherit From Other Classes in Python

[Inheritance](#) is the process by which one class takes on the attributes and methods of another. Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

Note: This tutorial is adapted from the chapter “Object-Oriented Programming (OOP)” in [Python Basics: A Practical Introduction to Python 3](#). If you enjoy what you’re reading, then be sure to check out [the rest of the book](#).

Child classes can override or extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent’s attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn’t perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your mother. It’s an attribute you were born with. Let’s say you decide to color your hair purple. Assuming your mother doesn’t have purple hair, you’ve just **overridden** the hair color attribute that you inherited from your mom.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you’ll also speak English. Now imagine you decide to learn a second language, like German. In this case you’ve **extended** your attributes because you’ve added an attribute that your parents don’t have.

Dog Park Example

Pretend for a moment that you’re at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The Dog class that you wrote in the previous section can distinguish dogs by name and age but not by breed.

You could modify the Dog class in the editor window by adding a .breed attribute:

Python

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age, breed):
        self.name = name
        self.age = age
        self.breed = breed
```

The instance methods defined earlier are omitted here because they aren’t important for this discussion.

Press **F5** to save the file. Now you can model the dog park by instantiating a bunch of different dogs in the interactive window:

Python

>>>

```
>>> miles = Dog("Miles", 4, "Jack Russell Terrier")
>>> buddy = Dog("Buddy", 9, "Dachshund")
>>> jack = Dog("Jack", 3, "Bulldog")
>>> jim = Dog("Jim", 5, "Bulldog")
```

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like *woof*, but dachshunds have a higher-pitched bark that sounds more like *yap*.

Using just the `Dog` class, you must supply a string for the `sound` argument of `.speak()` every time you call it on a `Dog` instance:

```
Python >>>
>>> buddy.speak("Yap")
'Buddy says Yap'

>>> jim.speak("Woof")
'Jim says Woof'

>>> jack.speak("Woof")
'Jack says Woof'
```

Passing a string to every call to `.speak()` is repetitive and inconvenient. Moreover, the string representing the sound that each `Dog` instance makes should be determined by its `.breed` attribute, but here you have to manually pass the correct string to `.speak()` every time it's called.

You can simplify the experience of working with the `Dog` class by creating a child class for each breed of dog. This allows you to extend the functionality that each child class inherits, including specifying a default argument for `.speak()`.

Parent Classes vs Child Classes

Let's create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here's the full definition of the `Dog` class:

```
Python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses. Add the following to the `dog.py` file to create three new child classes of the `Dog` class:

```
Python
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

Press `F5` to save and run the file. With the child classes defined, you can now instantiate some dogs of specific breeds in the interactive window:

```
Python >>>
>>> miles = JackRussellTerrier("Miles", 4)
>>> buddy = Dachshund("Buddy", 9)
```

```
>>> jack = Bulldog("Jack", 3)
>>> jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

Python

```
>>> miles.species
'Canis familiaris'

>>> buddy.name
'Buddy'

>>> print(jack)
Jack is 3 years old

>>> jim.speak("Woof")
'Jim says Woof'
```

>>>

To determine which class a given object belongs to, you can use the built-in `type()`:

Python

```
>>> type(miles)
<class '__main__.JackRussellTerrier'>
```

>>>

What if you want to determine if `miles` is also an instance of the `Dog` class? You can do this with the built-in `isinstance()`:

Python

```
>>> isinstance(miles, Dog)
True
```

>>>

Notice that `isinstance()` takes two arguments, an object and a class. In the example above, `isinstance()` checks if `miles` is an instance of the `Dog` class and returns `True`.

The `miles`, `buddy`, `jack`, and `jim` objects are all `Dog` instances, but `miles` is not a `Bulldog` instance, and `jack` is not a `Dachshund` instance:

Python

```
>>> isinstance(miles, Bulldog)
False

>>> isinstance(jack, Dachshund)
False
```

>>>

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've created child classes for some different breeds of dogs, let's give each breed its own sound.

Extend the Functionality of a Parent Class

Since different breeds of dogs have slightly different barks, you want to provide a default value for the `sound` argument of their respective `.speak()` methods. To do this, you need to override `.speak()` in the class definition for each breed.

To override a method defined on the parent class, you define a method with the same name on the child class. Here's what that looks like for the `JackRussellTerrier` class:

Python

```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

Now `.speak()` is defined on the `JackRussellTerrier` class with the default argument for `sound` set to "Arf".

Update `dog.py` with the new `JackRussellTerrier` class and press `F5` to save and run the file. You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to `sound`:

Python

>>>

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

Sometimes dogs make different barks, so if Miles gets angry and growls, you can still call `.speak()` with a different sound:

Python

>>>

```
>>> miles.speak("Grrr")
'Miles says Grrr'
```

One thing to keep in mind about class inheritance is that changes to the parent class automatically propagate to child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.

For example, in the editor window, change the string returned by `.speak()` in the `Dog` class:

Python

```
class Dog:
    # Leave other attributes and methods as they are

    # Change the string returned by .speak()
    def speak(self, sound):
        return f"{self.name} barks: {sound}"
```

Save the file and press `F5`. Now, when you create a new `Bulldog` instance named `jim`, `jim.speak()` returns the new string:

Python

>>>

```
>>> jim = Bulldog("Jim", 5)
>>> jim.speak("Woof")
'Jim barks: Woof'
```

However, calling `.speak()` on a `JackRussellTerrier` instance won't show the new style of output:

Python

>>>

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

Sometimes it makes sense to completely override a method from a parent class. But in this instance, we don't want the `JackRussellTerrier` class to lose any changes that might be made to the formatting of the output string of `Dog.speak()`.

To do this, you still need to define a `.speak()` method on the child `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the `Dog` class's `.speak()` *inside* of the child class's `.speak()` using the same arguments that you passed to `JackRussellTerrier.speak()`.

You can access the parent class from inside a method of a child class by using `super()`:

Python

```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)
```

When you call `super().speak(sound)` inside `JackRussellTerrier`, Python searches the parent class, `Dog`, for a `.speak()` method and calls it with the variable `sound`.

Update `dog.py` with the new `JackRussellTerrier` class. Save the file and press `F5` so you can test it in the interactive window:

```
Python >>>
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles barks: Arf'
```

Now when you call `miles.speak()`, you'll see output reflecting the new formatting in the `Dog` class.

Note: In the above examples, the **class hierarchy** is very straightforward. The `JackRussellTerrier` class has a single parent class, `Dog`. In real-world examples, the class hierarchy can get quite complicated.

`super()` does much more than just search the parent class for a method or an attribute. It traverses the entire class hierarchy for a matching method or attribute. If you aren't careful, `super()` can have surprising results.

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Class Inheritance

Show/Hide

You can expand the block below to see a solution:

Solution: Class Inheritance

Show/Hide

Conclusion

In this tutorial, you learned about object-oriented programming (OOP) in Python. Most modern programming languages, such as [Java](#), [C#](#), and [C++](#), follow OOP principles, so the knowledge you gained here will be applicable no matter where your programming career takes you.

In this tutorial, you learned how to:

- Define a **class**, which is a sort of blueprint for an object
- Instantiate an **object** from a class
- Use **attributes** and **methods** to define the **properties** and **behaviors** of an object
- Use **inheritance** to create **child classes** from a **parent class**
- Reference a method on a parent class using `super()`
- Check if an object inherits from another class using `isinstance()`

If you enjoyed what you learned in this sample from [Python Basics: A Practical Introduction to Python 3](#), then be sure to check out [the rest of the book](#).

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Intro to Object-Oriented Programming \(OOP\) in Python](#)

About David Amos

David is a mathematician by training, a data scientist/Python developer by profession, and a coffee junkie by choice.

[» More about David](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)

Recommended Video Course: [Intro to Object-Oriented Programming \(OOP\) in Python](#)



Real Python

Absolute vs Relative Imports in Python

by [Mbithe Nzomo](#) 27 Comments best-practices intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [A Quick Recap on Imports](#)
 - [How Imports Work](#)
 - [Syntax of Import Statements](#)
 - [Styling of Import Statements](#)
- [Absolute Imports](#)
 - [Syntax and Practical Examples](#)
 - [Pros and Cons of Absolute Imports](#)
- [Relative Imports](#)
 - [Syntax and Practical Examples](#)
 - [Pros and Cons of Relative Imports](#)
- [Conclusion](#)

Your Weekly Dose of All Things Python!

[pycoders.com](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Absolute vs Relative Imports in Python](#)

If you've worked on a Python project that has more than one file, chances are you've had to use an import statement before.

Even for Pythonistas with a couple of projects under their belt, imports can be confusing! You're probably reading this because you'd like to gain a deeper understanding of imports in Python, particularly absolute and relative imports.

In this tutorial, you'll learn the differences between the two, as well as their pros and cons. Let's dive right in!

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

A Quick Recap on Imports

You need to have a good understanding of [Python modules and packages](#) to know how imports work. A Python module is a file that has a `.py` extension, and a Python package is any folder that has modules inside it (or, in Python 2, a folder that contains an `__init__.py` file).

What happens when you have code in one module that needs to access code in another module or package? You import it!

How Imports Work

But how exactly do imports work? Let's say you import a module `abc` like so:

Python

```
import abc
```

The first thing Python will do is look up the name `abc` in [sys.modules](#). This is a cache of all modules that have been previously imported.

If the name isn't found in the module cache, Python will proceed to search through a list of built-in modules. These are modules that come pre-installed with Python and can be found in the [Python Standard Library](#). If the name still isn't found in the built-in modules, Python then searches for it in a list of directories defined by [sys.path](#). This list usually includes the current directory, which is searched first.

When Python finds the module, it binds it to a name in the local scope. This means that `abc` is now defined and can be used in the current file without throwing a `NameError`.

If the name is never found, you'll get a `ModuleNotFoundError`. You can find out more about imports in the Python documentation [here](#)!

Note: Security Concerns

Be aware that Python's import system presents some significant security risks. This is largely due to its flexibility. For example, the module cache is writable, and it is possible to override core Python functionality using the import system. Importing from third-party packages can also expose your application to security threats.

Here are a couple of interesting resources to learn more about these security concerns and how to mitigate them:

- [10 common security gotchas in Python and how to avoid them](#) by Anthony Shaw (Point 5 talks about Python's import system.)
- [Episode #168: 10 Python security holes and how to plug them](#) from the TalkPython podcast (The panelists begin talking about imports at around the 27:15 mark.)

Syntax of Import Statements

Now that you know how import statements work, let's explore their syntax. You can import both packages and modules. (Note that importing a package essentially imports the package's `__init__.py` file as a module.) You can also import specific objects from a package or module.

There are generally two types of import syntax. When you use the first one, you import the resource directly, like this:

Python

```
import abc
```

`abc` can be a package or a module.

When you use the second syntax, you import the resource from another package or module. Here's an example:

Python

```
from abc import xyz
```

xyz can be a module, subpackage, or object, such as a class or function.

You can also choose to rename an imported resource, like so:

Python

```
import abc as other_name
```

This renames the imported resource abc to other_name within the script. It must now be referenced as other_name, or it will not be recognized.

Styling of Import Statements

[PEP 8](#), the official [style guide for Python](#), has a few pointers when it comes to writing import statements. Here's a summary:

1. Imports should always be written at the top of the file, after any module comments and docstrings.
2. Imports should be divided according to what is being imported. There are generally three groups:
 - standard library imports (Python's built-in modules)
 - related third party imports (modules that are installed and do not belong to the current application)
 - local application imports (modules that belong to the current application)
3. Each group of imports should be separated by a blank space.

It's also a good idea to order your imports alphabetically within each import group. This makes finding particular imports much easier, especially when there are many imports in a file.

Here's an example of how to style import statements:

Python

```
"""Illustration of good import statement styling.

Note that the imports come after the docstring.

"""

# Standard library imports
import datetime
import os

# Third party imports
from flask import Flask
from flask_restful import Api
from flask_sqlalchemy import SQLAlchemy

# Local application imports
from local_module import local_class
from local_package import local_function
```

The import statements above are divided into three distinct groups, separated by a blank space. They are also ordered alphabetically within each group.

Absolute Imports

You've gotten up to speed on how to write import statements and how to style them like a pro. Now it's time to learn a little more about absolute imports.

An absolute import specifies the resource to be imported using its full path from the project's root folder.

Syntax and Practical Examples

Let's say you have the following directory structure:

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

There's a directory, `project`, which contains two sub-directories, `package1` and `package2`. The `package1` directory has two files, `module1.py` and `module2.py`.

The `package2` directory has three files: two modules, `module3.py` and `module4.py`, and an initialization file, `__init__.py`. It also contains a directory, `subpackage1`, which in turn contains a file, `module5.py`.

Let's assume the following:

1. `package1/module2.py` contains a function, `function1`.
2. `package2/__init__.py` contains a class, `class1`.
3. `package2/subpackage1/module5.py` contains a function, `function2`.

The following are practical examples of absolute imports:

Python

```
from package1 import module1
from package1.module2 import function1
from package2 import class1
from package2.subpackage1.module5 import function2
```

Note that you must give a detailed path for each package or file, from the top-level package folder. This is somewhat similar to its file path, but we use a dot (.) instead of a slash (/).

Pros and Cons of Absolute Imports

Absolute imports are preferred because they are quite clear and straightforward. It is easy to tell exactly where the imported resource is, just by looking at the statement. Additionally, absolute imports remain valid even if the current location of the import statement changes. In fact, PEP 8 explicitly recommends absolute imports.

Sometimes, however, absolute imports can get quite verbose, depending on the complexity of the directory structure. Imagine having a statement like this:

Python

```
from package1.subpackage2.subpackage3.subpackage4.module5 import function6
```

That's ridiculous, right? Luckily, relative imports are a good alternative in such cases!

Relative Imports

A relative import specifies the resource to be imported relative to the current location—that is, the location where the import statement is. There are two types of relative imports: implicit and explicit. Implicit relative imports have been deprecated in Python 3, so I won't be covering them here.

Syntax and Practical Examples

The syntax of a relative import depends on the current location as well as the location of the module, package, or object to be imported. Here are a few examples of relative imports:

Python

```
from .some_module import some_class
from ..some_package import some_function
from . import some_class
```

You can see that there is at least one dot in each import statement above. Relative imports make use of dot notation to specify location.

A single dot means that the module or package referenced is in the same directory as the current location. Two dots mean that it is in the parent directory of the current location—that is, the directory above. Three dots mean that it is in the grandparent directory, and so on. This will probably be familiar to you if you use a Unix-like operating system!

Let's assume you have the same directory structure as before:

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

Recall the file contents:

1. package1/module2.py contains a function, function1.
2. package2/__init__.py contains a class, class1.
3. package2/subpackage1/module5.py contains a function, function2.

You can import function1 into the package1/module1.py file this way:

Python

```
# package1/module1.py

from .module2 import function1
```

You'd use only one dot here because module2.py is in the same directory as the current module, which is module1.py.

You can import class1 and function2 into the package2/module3.py file this way:

Python

```
# package2/module3.py

from . import class1
from .subpackage1.module5 import function2
```

In the first import statement, the single dot means that you are importing class1 from the current package. Remember that importing a package essentially imports the package's __init__.py file as a module.

In the second import statement, you'd use a single dot again because subpackage1 is in the same directory as the current module, which is module3.py.

Pros and Cons of Relative Imports

One clear advantage of relative imports is that they are quite succinct. Depending on the current location, they can turn the ridiculously long import statement you saw earlier to something as simple as this:

Python

```
from ..subpackage4.module5 import function6
```

Unfortunately, relative imports can be messy, particularly for shared projects where directory structure is likely to change. Relative imports are also not as readable as absolute ones, and it's not easy to tell the location of the imported resources.

Conclusion

Good job for making it to the end of this crash course on absolute and relative imports! Now you're up to speed on how imports work. You've learned the best practices for writing import statements, and you know the difference between absolute and relative imports.

With your new skills, you can confidently import packages and modules from the Python standard library, third party packages, and your own local packages. Remember that you should generally opt for absolute imports over relative ones, unless the path is complex and would make the statement too long.

Thanks for reading!

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Absolute vs Relative Imports in Python](#)

About Mbithe Nzomo

I'm an experienced software engineer with interests in artificial intelligence and machine learning. I'm currently a postgraduate student at The University of Manchester, studying Advanced Computer Science with a specialisation in AI.

[» More about Mbithe](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Adriana](#)

[David](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [python](#)

Recommended Video Course: [Absolute vs Relative Imports in Python](#)



Working With Files in Python

by [Vuyisile Ndlovu](#) [30 Comments](#) [basics](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Python's “with open\(...\)as ...” Pattern](#)
- [Getting a Directory Listing](#)
 - [Directory Listing in Legacy Python Versions](#)
 - [Directory Listing in Modern Python Versions](#)
 - [Listing All Files in a Directory](#)
 - [Listing Subdirectories](#)
- [Getting File Attributes](#)
- [Making Directories](#)
 - [Creating a Single Directory](#)
 - [Creating Multiple Directories](#)
- [Filename Pattern Matching](#)
 - [Using String Methods](#)
 - [Simple Filename Pattern Matching Using fnmatch](#)
 - [More Advanced Pattern Matching](#)
 - [Filename Pattern Matching Using glob](#)
- [Traversing Directories and Processing Files](#)
- [Making Temporary Files and Directories](#)
- [Deleting Files and Directories](#)
 - [Deleting Files in Python](#)
 - [Deleting Directories](#)
 - [Deleting Entire Directory Trees](#)
- [Copying, Moving, and Renaming Files and Directories](#)
 - [Copying Files in Python](#)
 - [Copying Directories](#)
 - [Moving Files and Directories](#)
 - [Renaming Files and Directories](#)
- [Archiving](#)
 - [Reading ZIP Files](#)
 - [Extracting ZIP Archives](#)
 - [Extracting Data From Password Protected Archives](#)
 - [Creating New ZIP Archives](#)

- [Opening TAR Archives](#)
- [Extracting Files From a TAR Archive](#)
- [Creating New TAR Archives](#)
- [Working With Compressed Archives](#)
- [An Easier Way of Creating Archives](#)
- [Reading Multiple Files](#)
- [Conclusion](#)

Find Your Dream Python Job

pythonjobshq.com



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Practical Recipes for Working With Files in Python](#)

Python has several built-in modules and functions for handling files. These functions are spread out over several modules such as `os`, `os.path`, `shutil`, and `pathlib`, to name a few. This article gathers in one place many of the functions you need to know in order to perform the most common operations on files in Python.

In this tutorial, you'll learn how to:

- Retrieve file properties
- Create directories
- Match patterns in filenames
- Traverse directory trees
- Make temporary files and directories
- Delete files and directories
- Copy, move, or rename files and directories
- Create and extract ZIP and TAR archives
- Open multiple files using the `fileinput` module

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Python's “with open(...) as ...” Pattern

Reading and writing data to files using Python is pretty straightforward. To do this, you must first open files in the appropriate mode. Here's an example of how to use Python's “`with open(...)` as ...” pattern to open a text file and read its contents:

Python

```
with open('data.txt', 'r') as f:
    data = f.read()
```

`open()` takes a filename and a mode as its arguments. `r` opens the file in read only mode. To write data to a file, pass in `w` as an argument instead:

Python

```
with open('data.txt', 'w') as f:
    data = 'some data to be written to the file'
    f.write(data)
```

In the examples above, `open()` opens files for reading or writing and returns a file handle (`f` in this case) that provides methods that can be used to read or write data to the file. Read [Working With File I/O in Python](#) for more information on how to read and write to files.

Getting a Directory Listing

Suppose your current working directory has a subdirectory called `my_directory` that has the following contents:

```
my_directory/
|
└── sub_dir/
    ├── bar.py
    └── foo.py
|
└── sub_dir_b/
    └── file4.txt
|
└── sub_dir_c/
    ├── config.py
    └── file5.txt
|
└── file1.py
└── file2.csv
└── file3.txt
```

The built-in `os` module has a number of useful functions that can be used to list directory contents and filter the results. To get a list of all the files and folders in a particular directory in the filesystem, use `os.listdir()` in legacy versions of Python or `os.scandir()` in Python 3.x. `os.scandir()` is the preferred method to use if you also want to get file and directory properties such as file size and modification date.

Directory Listing in Legacy Python Versions

In versions of Python prior to Python 3, `os.listdir()` is the method to use to get a directory listing:

```
Python >>>
>>> import os
>>> entries = os.listdir('my_directory/')
```

`os.listdir()` returns a Python list containing the names of the files and subdirectories in the directory given by the path argument:

```
Python >>>
>>> os.listdir('my_directory/')
['sub_dir_c', 'file1.py', 'sub_dir_b', 'file3.txt', 'file2.csv', 'sub_dir']
```

A directory listing like that isn't easy to read. Printing out the output of a call to `os.listdir()` using a loop helps clean things up:

```
Python >>>
>>> entries = os.listdir('my_directory/')
>>> for entry in entries:
...     print(entry)
...
...
sub_dir_c
file1.py
sub_dir_b
file3.txt
file2.csv
```

```
sub_dir
```

Directory Listing in Modern Python Versions

In modern versions of Python, an alternative to `os.listdir()` is to use `os.scandir()` and `pathlib.Path()`.

`os.scandir()` was introduced in Python 3.5 and is documented in [PEP 471](#). `os.scandir()` returns an iterator as opposed to a list when called:

```
Python
```

```
>>>
```

```
>>> import os
>>> entries = os.scandir('my_directory/')
>>> entries
<posix.ScandirIterator object at 0x7f5b047f3690>
```

The `ScandirIterator` points to all the entries in the current directory. You can loop over the contents of the iterator and print out the filenames:

```
Python
```

```
import os

with os.scandir('my_directory/') as entries:
    for entry in entries:
        print(entry.name)
```

Here, `os.scandir()` is used in conjunction with the `with` statement because it supports the context manager protocol. Using a context manager closes the iterator and frees up acquired resources automatically after the iterator has been exhausted. The result is a print out of the filenames in `my_directory/` just like you saw in the `os.listdir()` example:

```
Shell
```

```
sub_dir_c
file1.py
sub_dir_b
file3.txt
file2.csv
sub_dir
```

Another way to get a directory listing is to use the `pathlib` module:

```
Python
```

```
from pathlib import Path

entries = Path('my_directory/')
for entry in entries.iterdir():
    print(entry.name)
```

The objects returned by `Path` are either `PosixPath` or `WindowsPath` objects depending on the OS.

`pathlib.Path()` objects have an `.iterdir()` method for creating an iterator of all files and folders in a directory. Each entry yielded by `.iterdir()` contains information about the file or directory such as its name and file attributes. `pathlib` was first introduced in Python 3.4 and is a great addition to Python that provides an object oriented interface to the filesystem.

In the example above, you call `pathlib.Path()` and pass a path argument to it. Next is the call to `.iterdir()` to get a list of all files and directories in `my_directory`.

`pathlib` offers a set of classes featuring most of the common operations on paths in an easy, object-oriented way. Using `pathlib` is more efficient than using the functions in `os`. Another benefit of using `pathlib` over `os` is that it reduces the number of imports you need to make to manipulate filesystem paths. For more information, read [Python 3's pathlib Module: Taming the File System](#).

Running the code above produces the following:

Shell

```
sub_dir_c
file1.py
sub_dir_b
file3.txt
file2.csv
sub_dir
```

Using `pathlib.Path()` or `os.scandir()` instead of `os.listdir()` is the preferred way of getting a directory listing, especially when you're working with code that needs the file type and file attribute information. `pathlib.Path()` offers much of the file and path handling functionality found in `os` and `shutil`, and its methods are more efficient than some found in these modules. We will discuss how to get file properties shortly.

Here are the directory-listing functions again:

Function	Description
<code>os.listdir()</code>	Returns a list of all files and folders in a directory
<code>os.scandir()</code>	Returns an iterator of all the objects in a directory including file attribute information
<code>pathlib.Path.iterdir()</code>	Returns an iterator of all the objects in a directory including file attribute information

These functions return a list of *everything* in the directory, including subdirectories. This might not always be the behavior you want. The next section will show you how to filter the results from a directory listing.

Listing All Files in a Directory

This section will show you how to print out the names of files in a directory using `os.listdir()`, `os.scandir()`, and `pathlib.Path()`. To filter out directories and only list files from a directory listing produced by `os.listdir()`, use `os.path`:

Python

```
import os

# List all files in a directory using os.listdir
basepath = 'my_directory/'
for entry in os.listdir(basepath):
    if os.path.isfile(os.path.join(basepath, entry)):
        print(entry)
```

Here, the call to `os.listdir()` returns a list of everything in the specified path, and then that list is filtered by `os.path.isfile()` to only print out files and not directories. This produces the following output:

Shell

```
file1.py
file3.txt
file2.csv
```

An easier way to list files in a directory is to use `os.scandir()` or `pathlib.Path()`:

Python

```
import os

# List all files in a directory using scandir()
basepath = 'my_directory/'
with os.scandir(basepath) as entries:
    for entry in entries:
        if entry.is_file():
            print(entry.name)
```

```
if entry.is_file():
    print(entry.name)
```

Using `os.scandir()` has the advantage of looking cleaner and being easier to understand than using `os.listdir()`, even though it is one line of code longer. Calling `entry.is_file()` on each item in the `ScandirIterator` returns `True` if the object is a file. Printing out the names of all files in the directory gives you the following output:

Shell

```
file1.py
file3.txt
file2.csv
```

Here's how to list files in a directory using `pathlib.Path()`:

Python

```
from pathlib import Path

basepath = Path('my_directory/')
files_in_basepath = basepath.iterdir()
for item in files_in_basepath:
    if item.is_file():
        print(item.name)
```

Here, you call `.is_file()` on each entry yielded by `.iterdir()`. The output produced is the same:

Shell

```
file1.py
file3.txt
file2.csv
```

The code above can be made more concise if you combine the `for` loop and the `if` statement into a single generator expression. Dan Bader has an [excellent article on generator expressions](#) and list comprehensions.

The modified version looks like this:

Python

```
from pathlib import Path

# List all files in directory using pathlib
basepath = Path('my_directory/')
files_in_basepath = (entry for entry in basepath.iterdir() if entry.is_file())
for item in files_in_basepath:
    print(item.name)
```

This produces exactly the same output as the example before it. This section showed that filtering files or directories using `os.scandir()` and `pathlib.Path()` feels more intuitive and looks cleaner than using `os.listdir()` in conjunction with `os.path`.

Listing Subdirectories

To list subdirectories instead of files, use one of the methods below. Here's how to use `os.listdir()` and `os.path()`:

Python

```
import os
```

```
# List all subdirectories using os.listdir
basepath = 'my_directory/'
for entry in os.listdir(basepath):
    if os.path.isdir(os.path.join(basepath, entry)):
        print(entry)
```

Manipulating filesystem paths this way can quickly become cumbersome when you have multiple calls to `os.path.join()`. Running this on my computer produces the following output:

Shell

```
sub_dir_c
sub_dir_b
sub_dir
```

Here's how to use `os.scandir()`:

Python

```
import os

# List all subdirectories using scandir()
basepath = 'my_directory'
with os.scandir(basepath) as entries:
    for entry in entries:
        if entry.is_dir():
            print(entry.name)
```

As in the file listing example, here you call `.is_dir()` on each entry returned by `os.scandir()`. If the entry is a directory, `.is_dir()` returns True, and the directory's name is printed out. The output is the same as above:

Shell

```
sub_dir_c
sub_dir_b
sub_dir
```

Here's how to use `pathlib.Path()`:

Python

```
from pathlib import Path

# List all subdirectory using pathlib
basepath = Path('my_directory/')
for entry in basepath.iterdir():
    if entry.is_dir():
        print(entry.name)
```

Calling `.is_dir()` on each entry of the `basepath` iterator checks if an entry is a file or a directory. If the entry is a directory, its name is printed out to the screen, and the output produced is the same as the one from the previous example:

Shell

```
sub_dir_c
sub_dir_b
sub_dir
```

Getting File Attributes

Python makes retrieving file attributes such as file size and modified times easy. This is done through `os.stat()`, `os.scandir()`, or `pathlib.Path()`.

`os.scandir()` and `pathlib.Path()` retrieve a directory listing with file attributes combined. This can be potentially more efficient than using `os.listdir()` to list files and then getting file attribute information for each file.

The examples below show how to get the time the files in `my_directory/` were last modified. The output is in seconds:

Python

>>>

```
>>> import os
>>> with os.scandir('my_directory/') as dir_contents:
...     for entry in dir_contents:
...         info = entry.stat()
...         print(info.st_mtime)
...
1539032199.0052035
1539032469.6324475
1538998552.2402923
1540233322.4009316
1537192240.0497339
1540266380.3434134
```

`os.scandir()` returns a `ScandirIterator` object. Each entry in a `ScandirIterator` object has a `.stat()` method that retrieves information about the file or directory it points to. `.stat()` provides information such as file size and the time of last modification. In the example above, the code prints out the `st_mtime` attribute, which is the time the content of the file was last modified.

The `pathlib` module has corresponding methods for retrieving file information that give the same results:

Python

>>>

```
>>> from pathlib import Path
>>> current_dir = Path('my_directory')
>>> for path in current_dir.iterdir():
...     info = path.stat()
...     print(info.st_mtime)
...
1539032199.0052035
1539032469.6324475
1538998552.2402923
1540233322.4009316
1537192240.0497339
1540266380.3434134
```

In the example above, the code loops through the object returned by `.iterdir()` and retrieves file attributes through a `.stat()` call for each file in the directory list. The `st_mtime` attribute returns a float value that represents [seconds since the epoch](#). To convert the values returned by `st_mtime` for display purposes, you could write a helper function to convert the seconds into a `datetime` object:

Python

>>>

```
from datetime import datetime
from os import scandir

def convert_date(timestamp):
    d = datetime.utcfromtimestamp(timestamp)
    formatted_date = d.strftime('%d %b %Y')
    return formatted_date

def get_files():
    dir_entries = scandir('my_directory/')
```

```

for entry in dir_entries:
    if entry.is_file():
        info = entry.stat()
        print(f'{entry.name}\t Last Modified: {convert_date(info.st_mtime)})'

```

This will first get a list of files in `my_directory` and their attributes and then call `convert_date()` to convert each file's last modified time into a human readable form. `convert_date()` makes use of `.strftime()` to convert the time in seconds into a string.

The arguments passed to `.strftime()` are the following:

- `%d`: the day of the month
- `%b`: the month, in abbreviated form
- `%Y`: the year

Together, these directives produce output that looks like this:

Python

>>>

```

>>> get_files()
file1.py      Last modified: 04 Oct 2018
file3.txt     Last modified: 17 Sep 2018
file2.txt     Last modified: 17 Sep 2018

```

The syntax for converting dates and times into strings can be quite confusing. To read more about it, check out the [official documentation](#) on it. Another handy reference that is easy to remember is <http://strftime.org/>.

Making Directories

Sooner or later, the programs you write will have to create directories in order to store data in them. `os` and `pathlib` include functions for creating directories. We'll consider these:

Function	Description
<code>os.mkdir()</code>	Creates a single subdirectory
<code>pathlib.Path.mkdir()</code>	Creates single or multiple directories
<code>os.makedirs()</code>	Creates multiple directories, including intermediate directories

Creating a Single Directory

To create a single directory, pass a path to the directory as a parameter to `os.mkdir()`:

Python

```

import os

os.mkdir('example_directory/')

```

If a directory already exists, `os.mkdir()` raises `FileExistsError`. Alternatively, you can create a directory using `pathlib`:

Python

```

from pathlib import Path

p = Path('example_directory/')
p.mkdir()

```

If the path already exists, `mkdir()` raises a `FileExistsError`:

Python

>>>

```
>>> p.mkdir()
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
  File '/usr/lib/python3.5/pathlib.py', line 1214, in mkdir
    self._accessor.mkdir(self, mode)
  File '/usr/lib/python3.5/pathlib.py', line 371, in wrapped
    return strfunc(str(pathobj), *args)
FileExistsError: [Errno 17] File exists: '.'
[Errno 17] File exists: '..'
```

To avoid errors like this, [catch the error](#) when it happens and let your user know:

Python

```
from pathlib import Path

p = Path('example_directory')
try:
    p.mkdir()
except FileExistsError as exc:
    print(exc)
```

Alternatively, you can ignore the `FileExistsError` by passing the `exist_ok=True` argument to `.mkdir()`:

Python

```
from pathlib import Path

p = Path('example_directory')
p.mkdir(exist_ok=True)
```

This will not raise an error if the directory already exists.

Creating Multiple Directories

`os.makedirs()` is similar to `os.mkdir()`. The difference between the two is that not only can `os.makedirs()` create individual directories, it can also be used to create directory trees. In other words, it can create any necessary intermediate folders in order to ensure a full path exists.

`os.makedirs()` is similar to running `mkdir -p` in Bash. For example, to create a group of directories like `2018/10/05`, all you have to do is the following:

Python

```
import os

os.makedirs('2018/10/05')
```

This will create a nested directory structure that contains the folders 2018, 10, and 05:

```
.
|
└── 2018/
    └── 10/
        └── 05/
```

`.makedirs()` creates directories with default permissions. If you need to create directories with different permissions call `.makedirs()` and pass in the mode you would like the directories to be created in:

Python

```
ryu@ryu:
```

```
import os  
  
os.makedirs('2018/10/05', mode=0o770)
```

This creates the 2018/10/05 directory structure and gives the owner and group users read, write, and execute permissions. The default mode is 0o777, and the file permission bits of existing parent directories are not changed. For more details on file permissions, and how the mode is applied, [see the docs](#).

Run tree to confirm that the right permissions were applied:

Shell

```
$ tree -p -i .  
. .  
[drwxrwx---] 2018  
[drwxrwx---] 10  
[drwxrwx---] 05
```

This prints out a directory tree of the current directory. tree is normally used to list contents of directories in a tree-like format. Passing the -p and -i arguments to it prints out the directory names and their file permission information in a vertical list. -p prints out the file permissions, and -i makes tree produce a vertical list without indentation lines.

As you can see, all of the directories have 770 permissions. An alternative way to create directories is to use `.mkdir()` from `pathlib.Path`:

Python

```
import pathlib  
  
p = pathlib.Path('2018/10/05')  
p.mkdir(parents=True)
```

Passing `parents=True` to `Path.mkdir()` makes it create the directory 05 and any parent directories necessary to make the path valid.

By default, `os.makedirs()` and `Path.mkdir()` raise an `OSError` if the target directory already exists. This behavior can be overridden (as of Python 3.2) by passing `exist_ok=True` as a keyword argument when calling each function.

Running the code above produces a directory structure like the one below in one go:

```
.  
|  
└── 2018/  
    └── 10/  
        └── 05/
```

I prefer using `pathlib` when creating directories because I can use the same function to create single or nested directories.

Filename Pattern Matching

After getting a list of files in a directory using one of the methods above, you will most probably want to search for files that match a particular pattern.

These are the methods and functions available to you:

- `endswith()` and `startswith()` string methods
- `fnmatch.fnmatch()`
- `glob.glob()`
- `pathlib.Path.glob()`

Each of these is discussed below. The examples in this section will be performed on a directory called `some_directory` that has the following structure:

```
.  
|  
|   └── sub_dir/  
|       ├── file1.py  
|       └── file2.py  
|  
|   └── admin.py  
|   ├── data_01_backup.txt  
|   ├── data_01.txt  
|   ├── data_02_backup.txt  
|   ├── data_02.txt  
|   ├── data_03_backup.txt  
|   ├── data_03.txt  
|   └── tests.py
```

If you're following along using a Bash shell, you can create the above directory structure using the following commands:

Shell

```
$ mkdir some_directory  
$ cd some_directory/  
$ mkdir sub_dir  
$ touch sub_dir/file1.py sub_dir/file2.py  
$ touch data_{01..03}.txt data_{01..03}_backup.txt admin.py tests.py
```

This will create the `some_directory/` directory, change into it, and then create `sub_dir`. The next line creates `file1.py` and `file2.py` in `sub_dir`, and the last line creates all the other files using expansion. To learn more about shell expansion, visit [this site](#).

Using String Methods

Python has several built-in methods for [modifying and manipulating strings](#). Two of these methods, `.startswith()` and `.endswith()`, are useful when you're searching for patterns in filenames. To do this, first get a directory listing and then iterate over it:

Python

>>>

```
>>> import os  
  
>>> # Get .txt files  
>>> for f_name in os.listdir('some_directory'):  
...     if f_name.endswith('.txt'):  
...         print(f_name)
```

The code above finds all the files in `some_directory/`, iterates over them and uses `.endswith()` to print out the filenames that have the `.txt` file extension. Running this on my computer produces the following output:

Shell

```
data_01.txt  
data_03.txt  
data_03_backup.txt  
data_02_backup.txt  
data_02.txt  
data_01_backup.txt
```

Simple Filename Pattern Matching Using fnmatch

String methods are limited in their matching abilities. `fnmatch` has more advanced functions and methods for pattern matching. We will consider `fnmatch.fnmatch()`, a function that supports the use of wildcards such as `*` and `?` to match filenames. For example, in order to find all `.txt` files in a directory using `fnmatch`, you would do the

following:

```
Python >>>
>>> import os
>>> import fnmatch

>>> for file_name in os.listdir('some_directory/'):
...     if fnmatch.fnmatch(file_name, '*.txt'):
...         print(file_name)
```

This iterates over the list of files in some_directory and uses .fnmatch() to perform a wildcard search for files that have the .txt extension.

More Advanced Pattern Matching

Let's suppose you want to find .txt files that meet certain criteria. For example, you could be only interested in finding .txt files that contain the word data, a number between a set of underscores, and the word backup in their filename. Something similar to data_01_backup, data_02_backup, or data_03_backup.

Using fnmatch.fnmatch(), you could do it this way:

```
Python >>>
>>> for filename in os.listdir('.'):
...     if fnmatch.fnmatch(filename, 'data_*_backup.txt'):
...         print(filename)
```

Here, you print only the names of files that match the data_*_backup.txt pattern. The asterisk in the pattern will match any character, so running this will find all text files whose filenames start with the word data and end in backup.txt, as you can see from the output below:

```
Shell
data_03_backup.txt
data_02_backup.txt
data_01_backup.txt
```

Filename Pattern Matching Using glob

Another useful module for pattern matching is glob.

.glob() in the glob module works just like fnmatch.fnmatch(), but unlike fnmatch.fnmatch(), it treats files beginning with a period (.) as special.

UNIX and related systems translate name patterns with wildcards like ? and * into a list of files. This is called globbing.

For example, typing mv *.py python_files/ in a UNIX shell moves (mv) all files with the .py extension from the current directory to the directory python_files. The * character is a wildcard that means “any number of characters,” and *.py is the glob pattern. This shell capability is not available in the Windows Operating System. The glob module adds this capability in Python, which enables Windows programs to use this feature.

Here's an example of how to use glob to search for all Python (.py) source files in the current directory:

```
Python >>>
>>> import glob
>>> glob.glob('*.*py')
['admin.py', 'tests.py']
```

glob.glob('*.*py') searches for all files that have the .py extension in the current directory and returns them as a list. glob also supports shell-style wildcards to match patterns:

Python

>>>

```
>>> import glob  
>>> for name in glob.glob('*[0-9]*.txt'):  
...     print(name)
```

This finds all text (.txt) files that contain digits in the filename:

Shell

```
data_01.txt  
data_03.txt  
data_03_backup.txt  
data_02_backup.txt  
data_02.txt  
data_01_backup.txt
```

glob makes it easy to search for files recursively in subdirectories too:

Python

>>>

```
>>> import glob  
>>> for file in glob.iglob('**/*.py', recursive=True):  
...     print(file)
```

This example makes use of `glob.iglob()` to search for .py files in the current directory and subdirectories. Passing `recursive=True` as an argument to `.iglob()` makes it search for .py files in the current directory and any subdirectories. The difference between `glob.iglob()` and `glob.glob()` is that `.iglob()` returns an iterator instead of a list.

Running the program above produces the following:

Shell

```
admin.py  
tests.py  
sub_dir/file1.py  
sub_dir/file2.py
```

`pathlib` contains similar methods for making flexible file listings. The example below shows how you can use `.Path.glob()` to list file types that start with the letter p:

Python

>>>

```
>>> from pathlib import Path  
>>> p = Path('.').  
>>> for name in p.glob('*.*p*'):.  
...     print(name)  
  
admin.py  
scraper.py  
docs.pdf
```

Calling `p.glob('*.*p*')` returns a generator object that points to all files in the current directory that start with the letter p in their file extension.

`Path.glob()` is similar to `os.glob()` discussed above. As you can see, `pathlib` combines many of the best features of the `os`, `os.path`, and `glob` modules into one single module, which makes it a joy to use.

To recap, here is a table of the functions we have covered in this section:

Function	Description
----------	-------------

Function	Description
startswith()	Tests if a string starts with a specified pattern and returns True or False
endswith()	Tests if a string ends with a specified pattern and returns True or False
fnmatch.fnmatch(filename, pattern)	Tests whether the filename matches the pattern and returns True or False
glob.glob()	Returns a list of filenames that match a pattern
pathlib.Path.glob()	Finds patterns in path names and returns a generator object

Traversing Directories and Processing Files

A common programming task is walking a directory tree and processing files in the tree. Let's explore how the built-in Python function `os.walk()` can be used to do this. `os.walk()` is used to generate filename in a directory tree by walking the tree either top-down or bottom-up. For the purposes of this section, we'll be manipulating the following directory tree:

```
.
|
└── folder_1/
    ├── file1.py
    ├── file2.py
    └── file3.py
|
└── folder_2/
    ├── file4.py
    ├── file5.py
    └── file6.py
|
└── test1.txt
└── test2.txt
```

The following is an example that shows you how to list all files and directories in a directory tree using `os.walk()`.

`os.walk()` defaults to traversing directories in a top-down manner:

Python

```
# Walking a directory tree and printing the names of the directories and files
for dirpath, dirnames, files in os.walk('.'):
    print(f'Found directory: {dirpath}')
    for file_name in files:
        print(file_name)
```

`os.walk()` returns three values on each iteration of the loop:

1. The name of the current folder
2. A list of folders in the current folder
3. A list of files in the current folder

On each iteration, it prints out the names of the subdirectories and files it finds:

Shell

```
Found directory: .
test1.txt
test2.txt
Found directory: ./folder_1
file1.py
file3.py
file2.py
Found directory: ./folder_2
file4.py
file5.py
file6.py
```

To traverse the directory tree in a bottom-up manner, pass in a `topdown=False` keyword argument to `os.walk()`:

Python

```
for dirpath, dirnames, files in os.walk('.', topdown=False):
    print(f'Found directory: {dirpath}')
    for file_name in files:
        print(file_name)
```

Passing the `topdown=False` argument will make `os.walk()` print out the files it finds in the *subdirectories* first:

Shell

```
Found directory: ./folder_1
file1.py
file3.py
file2.py
Found directory: ./folder_2
file4.py
file5.py
file6.py
Found directory: .
test1.txt
test2.txt
```

As you can see, the program started by listing the contents of the subdirectories before listing the contents of the root directory. This is very useful in situations where you want to recursively delete files and directories. You will learn how to do this in the sections below. By default, `os.walk` does not walk down into symbolic links that resolve to directories. This behavior can be overridden by calling it with a `followlinks=True` argument.

Making Temporary Files and Directories

Python provides a handy module for creating temporary files and directories called `tempfile`.

`tempfile` can be used to open and store data temporarily in a file or directory while your program is running. `tempfile` handles the deletion of the temporary files when your program is done with them.

Here's how to create a temporary file:

Python

```
from tempfile import TemporaryFile

# Create a temporary file and write some data to it
fp = TemporaryFile('w+t')
fp.write('Hello universe!')

# Go back to the beginning and read data from file
fp.seek(0)
data = fp.read()

# Close the file, after which it will be removed
fp.close()
```

The first step is to import `TemporaryFile` from the `tempfile` module. Next, create a file like object using the `TemporaryFile()` method by calling it and passing the mode you want to open the file in. This will create and open a file that can be used as a temporary storage area.

In the example above, the mode is '`w+t`', which makes `tempfile` create a temporary text file in write mode. There is no need to give the temporary file a filename since it will be destroyed after the script is done running.

After writing to the file, you can read from it and close it when you're done processing it. Once the file is closed, it will be deleted from the filesystem. If you need to name the temporary files produced using `tempfile`, use `tempfile.NamedTemporaryFile()`.

The temporary files and directories created using `tempfile` are stored in a special system directory for storing temporary files. Python searches a standard list of directories to find one that the user can create files in.

On Windows, the directories are `C:\TEMP`, `C:\TMP`, `\TEMP`, and `\TMP`, in that order. On all other platforms, the directories are `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order. As a last resort, `tempfile` will save temporary files and directories in the current directory.

`.TemporaryFile()` is also a context manager so it can be used in conjunction with the `with` statement. Using a context manager takes care of closing and deleting the file automatically after it has been read:

Python

```
with TemporaryFile('w+t') as fp:
    fp.write('Hello universe!')
    fp.seek(0)
    fp.read()
# File is now closed and removed
```

This creates a temporary file and reads data from it. As soon as the file's contents are read, the temporary file is closed and deleted from the file system.

`tempfile` can also be used to create temporary directories. Let's look at how you can do this using `tempfile.TemporaryDirectory()`:

Python

>>>

```
>>> import tempfile
>>> with tempfile.TemporaryDirectory() as tmpdir:
...     print('Created temporary directory ', tmpdir)
...     os.path.exists(tmpdir)
...
Created temporary directory  /tmp/tmpoxbkrm6c
True

>>> # Directory contents have been removed
...
>>> tmpdir
'/tmp/tmpoxbkrm6c'
>>> os.path.exists(tmpdir)
False
```

Calling `tempfile.TemporaryDirectory()` creates a temporary directory in the file system and returns an object representing this directory. In the example above, the directory is created using a context manager, and the name of the directory is stored in `tmpdir`. The third line prints out the name of the temporary directory, and `os.path.exists(tmpdir)` confirms if the directory was actually created in the file system.

After the context manager goes out of context, the temporary directory is deleted and a call to `os.path.exists(tmpdir)` returns `False`, which means that the directory was successfully deleted.

Deleting Files and Directories

You can delete single files, directories, and entire directory trees using the methods found in the `os`, `shutil`, and `pathlib` modules. The following sections describe how to delete files and directories that you no longer need.

Deleting Files in Python

To delete a single file, use `pathlib.Path.unlink()`, `os.remove()`, or `os.unlink()`.

`os.remove()` and `os.unlink()` are semantically identical. To delete a file using `os.remove()`, do the following:

Python

```
import os

data_file = 'C:\\\\Users\\\\vuyisile\\\\Desktop\\\\Test\\\\data.txt'
os.remove(data_file)
```

Deleting a file using `os.unlink()` is similar to how you do it using `os.remove()`:

Python

```
import os

data_file = 'C:\\\\Users\\\\vuyisile\\\\Desktop\\\\Test\\\\data.txt'
os.unlink(data_file)
```

Calling `.unlink()` or `.remove()` on a file deletes the file from the filesystem. These two functions will throw an `OSError` if the path passed to them points to a directory instead of a file. To avoid this, you can either check that what you're trying to delete is actually a file and only delete it if it is, or you can use exception handling to handle the `OSError`:

Python

```
import os

data_file = 'home/data.txt'
```

```

# If the file exists, delete it
if os.path.isfile(data_file):
    os.remove(data_file)
else:
    print(f'Error: {data_file} not a valid filename')

```

`os.path.isfile()` checks whether `data_file` is actually a file. If it is, it is deleted by the call to `os.remove()`. If `data_file` points to a folder, an error message is printed to the console.

The following example shows how to use exception handling to handle errors when deleting files:

Python

```

import os

data_file = 'home/data.txt'

# Use exception handling
try:
    os.remove(data_file)
except OSError as e:
    print(f'Error: {data_file} : {e.strerror}')

```

The code above attempts to delete the file first before checking its type. If `data_file` isn't actually a file, the `OSError` that is thrown is handled in the `except` clause, and an error message is printed to the console. The error message that gets printed out is formatted using [Python f-strings](#).

Finally, you can also use `pathlib.Path.unlink()` to delete files:

Python

```

from pathlib import Path

data_file = Path('home/data.txt')

try:
    data_file.unlink()
except IsADirectoryError as e:
    print(f'Error: {data_file} : {e.strerror}')

```

This creates a `Path` object called `data_file` that points to a file. Calling `.remove()` on `data_file` will delete `home/data.txt`. If `data_file` points to a directory, an `IsADirectoryError` is raised. It is worth noting that the Python program above has the same permissions as the user running it. If the user does not have permission to delete the file, a `PermissionError` is raised.

Deleting Directories

The standard library offers the following functions for deleting directories:

- `os.rmdir()`
- `pathlib.Path.rmdir()`
- `shutil.rmtree()`

To delete a single directory or folder, use `os.rmdir()` or `pathlib.rmdir()`. These two functions only work if the directory you're trying to delete is empty. If the directory isn't empty, an `OSError` is raised. Here is how to delete a folder:

Python

```

import os

trash_dir = 'my_documents/bad_dir'

try:
    os.rmdir(trash_dir)
except OSError as e:
    print(f'Error: {trash_dir} : {e.strerror}')

```

Here, the `trash_dir` directory is deleted by passing its path to `os.rmdir()`. If the directory isn't empty, an error message is printed to the screen:

```
Python >>>
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
OSError: [Errno 39] Directory not empty: 'my_documents/bad_dir'
```

Alternatively, you can use `pathlib` to delete directories:

```
Python
from pathlib import Path

trash_dir = Path('my_documents/bad_dir')

try:
    trash_dir.rmdir()
except OSError as e:
    print(f'Error: {trash_dir} : {e.strerror}'')
```

Here, you create a `Path` object that points to the directory to be deleted. Calling `.rmdir()` on the `Path` object will delete it if it is empty.

Deleting Entire Directory Trees

To delete non-empty directories and entire directory trees, Python offers `shutil.rmtree()`:

```
Python
import shutil

trash_dir = 'my_documents/bad_dir'

try:
    shutil.rmtree(trash_dir)
except OSError as e:
    print(f'Error: {trash_dir} : {e.strerror}'')
```

Everything in `trash_dir` is deleted when `shutil.rmtree()` is called on it. There may be cases where you want to delete empty folders recursively. You can do this using one of the methods discussed above in conjunction with `os.walk()`:

```
Python
import os

for dirpath, dirnames, files in os.walk('.', topdown=False):
    try:
        os.rmdir(dirpath)
    except OSError as ex:
        pass
```

This walks down the directory tree and tries to delete each directory it finds. If the directory isn't empty, an `OSError` is raised and that directory is skipped. The table below lists the functions covered in this section:

Function	Description
<code>os.remove()</code>	Deletes a file and does not delete directories
<code>os.unlink()</code>	Is identical to <code>os.remove()</code> and deletes a single file
<code>pathlib.Path.unlink()</code>	Deletes a file and cannot delete directories

Function	Description
<code>os.rmdir()</code>	Deletes an empty directory
<code>pathlib.Path.rmdir()</code>	Deletes an empty directory
<code>shutil.rmtree()</code>	Deletes entire directory tree and can be used to delete non-empty directories

Copying, Moving, and Renaming Files and Directories

Python ships with the `shutil` module. `shutil` is short for shell utilities. It provides a number of high-level operations on files to support copying, archiving, and removal of files and directories. In this section, you'll learn how to move and copy files and directories.

Copying Files in Python

`shutil` offers a couple of functions for copying files. The most commonly used functions are `shutil.copy()` and `shutil.copy2()`. To copy a file from one location to another using `shutil.copy()`, do the following:

Python

```
import shutil

src = 'path/to/file.txt'
dst = 'path/to/dest_dir'
shutil.copy(src, dst)
```

`shutil.copy()` is comparable to the `cp` command in UNIX based systems. `shutil.copy(src, dst)` will copy the file `src` to the location specified in `dst`. If `dst` is a file, the contents of that file are replaced with the contents of `src`. If `dst` is a directory, then `src` will be copied into that directory. `shutil.copy()` only copies the file's contents and the file's permissions. Other metadata like the file's creation and modification times are not preserved.

To preserve all file metadata when copying, use `shutil.copy2()`:

Python

```
import shutil

src = 'path/to/file.txt'
dst = 'path/to/dest_dir'
shutil.copy2(src, dst)
```

Using `.copy2()` preserves details about the file such as last access time, permission bits, last modification time, and flags.

Copying Directories

While `shutil.copy()` only copies a single file, `shutil.copytree()` will copy an entire directory and everything contained in it. `shutil.copytree(src, dest)` takes two arguments: a source directory and the destination directory where files and folders will be copied to.

Here's an example of how to copy the contents of one folder to a different location:

Python

>>>

```
>>> import shutil
>>> shutil.copytree('data_1', 'data1_backup')
'data1_backup'
```

In this example, `.copytree()` copies the contents of `data_1` to a new location `data1_backup` and returns the destination directory. The destination directory must not already exist. It will be created as well as missing parent directories. `shutil.copytree()` is a good way to back up your files.

Moving Files and Directories

To move a file or directory to another location, use `shutil.move(src, dst)`.

`src` is the file or directory to be moved and `dst` is the destination:

Python

>>>

```
>>> import shutil  
>>> shutil.move('dir_1/', 'backup/')
```

`shutil.move('dir_1/', 'backup/')` moves `dir_1/` into `backup/` if `backup/` exists. If `backup/` does not exist, `dir_1/` will be renamed to `backup/`.

Renaming Files and Directories

Python includes `os.rename(src, dst)` for renaming files and directories:

Python

>>>

```
>>> os.rename('first.zip', 'first_01.zip')
```

The line above will rename `first.zip` to `first_01.zip`. If the destination path points to a directory, it will raise an `OSError`.

Another way to rename files or directories is to use `rename()` from the `pathlib` module:

Python

>>>

```
>>> from pathlib import Path  
>>> data_file = Path('data_01.txt')  
>>> data_file.rename('data.txt')
```

To rename files using `pathlib`, you first create a `pathlib.Path()` object that contains a path to the file you want to replace. The next step is to call `rename()` on the path object and pass a new filename for the file or directory you're renaming.

Archiving

Archives are a convenient way to package several files into one. The two most common archive types are ZIP and TAR. The Python programs you write can create, read, and extract data from archives. You will learn how to read and write to both archive formats in this section.

Reading ZIP Files

The `zipfile` module is a low level module that is part of the Python Standard Library. `zipfile` has functions that make it easy to open and extract ZIP files. To read the contents of a ZIP file, the first thing to do is to create a `ZipFile` object. `ZipFile` objects are similar to file objects created using `open()`. `ZipFile` is also a context manager and therefore supports the `with` statement:

Python

```
import zipfile  
  
with zipfile.ZipFile('data.zip', 'r') as zipobj:
```

Here, you create a `ZipFile` object, passing in the name of the ZIP file to open in read mode. After opening a ZIP file, information about the archive can be accessed through functions provided by the `zipfile` module. The `data.zip` archive in the example above was created from a directory named `data` that contains a total of 5 files and 1 subdirectory:

:

```
|  
|   └── sub_dir/  
|       ├── bar.py  
|       └── foo.py  
|  
└── file1.py  
└── file2.py  
└── file3.py
```

To get a list of files in the archive, call `namelist()` on the `ZipFile` object:

Python

```
import zipfile  
  
with zipfile.ZipFile('data.zip', 'r') as zipobj:  
    zipobj.namelist()
```

This produces a list:

Shell

```
['file1.py', 'file2.py', 'file3.py', 'sub_dir/', 'sub_dir/bar.py', 'sub_dir/foo.py']
```

`.namelist()` returns a list of names of the files and directories in the archive. To retrieve information about the files in the archive, use `.getinfo()`:

Python

```
import zipfile  
  
with zipfile.ZipFile('data.zip', 'r') as zipobj:  
    bar_info = zipobj.getinfo('sub_dir/bar.py')  
    bar_info.file_size
```

Here's the output:

Shell

```
15277
```

`.getinfo()` returns a `ZipInfo` object that stores information about a single member of the archive. To get information about a file in the archive, you pass its path as an argument to `.getinfo()`. Using `getinfo()`, you're able to retrieve information about archive members such as the date the files were last modified, their compressed sizes, and their full filenames. Accessing `.file_size` retrieves the file's original size in bytes.

The following example shows how to retrieve more details about archived files in a Python REPL. Assume that the `zipfile` module has been imported and `bar_info` is the same object you created in previous examples:

Python

>>>

```
>>> bar_info.date_time  
(2018, 10, 7, 23, 30, 10)  
>>> bar_info.compress_size  
2856  
>>> bar_info.filename  
'sub_dir/bar.py'
```

`bar_info` contains details about `bar.py` such as its size when compressed and its full path.

The first line shows how to retrieve a file's last modified date. The next line shows how to get the size of the file after compression. The last line shows the full path of `bar.py` in the archive.

`ZipFile` supports the context manager protocol, which is why you're able to use it with the `with` statement. Doing this automatically closes the `ZipFile` object after you're done with it. Trying to open or extract files from a closed `ZipFile` object will result in an error.

EXTRACTING ZIP ARCHIVES

The `zipfile` module allows you to extract one or more files from ZIP archives through `.extract()` and `.extractall()`.

These methods extract files to the current directory by default. They both take an optional path parameter that allows you to specify a different directory to extract files to. If the directory does not exist, it is automatically created. To extract files from the archive, do the following:

```
Python >>>
>>> import zipfile
>>> import os

>>> os.listdir('.')
['data.zip']

>>> data_zip = zipfile.ZipFile('data.zip', 'r')

>>> # Extract a single file to current directory
>>> data_zip.extract('file1.py')
'/home/terra/test/dir1/zip_extract/file1.py'

>>> os.listdir('.')
['file1.py', 'data.zip']

>>> # Extract all files into a different directory
>>> data_zip.extractall(path='extract_dir/')

>>> os.listdir('.')
['file1.py', 'extract_dir', 'data.zip']

>>> os.listdir('extract_dir')
['file1.py', 'file3.py', 'file2.py', 'sub_dir']

>>> data_zip.close()
```

The third line of code is a call to `os.listdir()`, which shows that the current directory has only one file, `data.zip`.

Next, you open `data.zip` in read mode and call `.extract()` to extract `file1.py` from it. `.extract()` returns the full file path of the extracted file. Since there's no path specified, `.extract()` extracts `file1.py` to the current directory.

The next line prints a directory listing showing that the current directory now includes the extracted file in addition to the original archive. The line after that shows how to extract the entire archive into the `zip_extract` directory. `.extractall()` creates the `extract_dir` and extracts the contents of `data.zip` into it. The last line closes the ZIP archive.

Extracting Data From Password Protected Archives

`zipfile` supports extracting password protected ZIPs. To extract password protected ZIP files, pass in the password to the `.extract()` or `.extractall()` method as an argument:

```
Python >>>
>>> import zipfile

>>> with zipfile.ZipFile('secret.zip', 'r') as pwd_zip:
...     # Extract from a password protected archive
...     pwd_zip.extractall(path='extract_dir', pwd='Quish3@o')
```

This opens the `secret.zip` archive in read mode. A password is supplied to `.extractall()`, and the archive contents are extracted to `extract_dir`. The archive is closed automatically after the extraction is complete thanks to the `with` statement.

Creating New ZIP Archives

To create a new ZIP archive, you open a `ZipFile` object in write mode (`w`) and add the files you want to archive:

```
Python >>>
```

```

>>> import zipfile

>>> file_list = ['file1.py', 'sub_dir/', 'sub_dir/bar.py', 'sub_dir/foo.py']
>>> with zipfile.ZipFile('new.zip', 'w') as new_zip:
...     for name in file_list:
...         new_zip.write(name)

```

In the example, `new_zip` is opened in write mode and each file in `file_list` is added to the archive. When the `with` statement suite is finished, `new_zip` is closed. Opening a ZIP file in write mode erases the contents of the archive and creates a new archive.

To add files to an existing archive, open a `ZipFile` object in append mode and then add the files:

Python	>>>
<pre> >>> # Open a ZipFile object in append mode >>> with zipfile.ZipFile('new.zip', 'a') as new_zip: ... new_zip.write('data.txt') ... new_zip.write('latin.txt') </pre>	

Here, you open the `new.zip` archive you created in the previous example in append mode. Opening the `ZipFile` object in append mode allows you to add new files to the ZIP file without deleting its current contents. After adding files to the ZIP file, the `with` statement goes out of context and closes the ZIP file.

Opening TAR Archives

TAR files are uncompressed file archives like ZIP. They can be compressed using gzip, bzip2, and lzma compression methods. The `TarFile` class allows reading and writing of TAR archives.

Do this to read from an archive:

Python	>>>
<pre> import tarfile with tarfile.open('example.tar', 'r') as tar_file: print(tar_file.getnames()) </pre>	

`tarfile` objects open like most file-like objects. They have an `open()` function that takes a mode that determines how the file is to be opened.

Use the '`r`', '`w`' or '`a`' modes to open an uncompressed TAR file for reading, writing, and appending, respectively. To open compressed TAR files, pass in a mode argument to `tarfile.open()` that is in the form `filemode[:compression]`. The table below lists the possible modes TAR files can be opened in:

Mode	Action
<code>r</code>	Opens archive for reading with transparent compression
<code>r:gz</code>	Opens archive for reading with gzip compression
<code>r:bz2</code>	Opens archive for reading with bzip2 compression
<code>r:xz</code>	Opens archive for reading with lzma compression
<code>w</code>	Opens archive for uncompressed writing
<code>w:gz</code>	Opens archive for gzip compressed writing
<code>w:xz</code>	Opens archive for lzma compressed writing

Mode	Action
------	--------

.open() defaults to 'r' mode. To read an uncompressed TAR file and retrieve the names of the files in it, use .getnames():

Python

>>>

```
>>> import tarfile

>>> tar = tarfile.open('example.tar', mode='r')
>>> tar.getnames()
['CONTRIBUTING.rst', 'README.md', 'app.py']
```

This returns a list with the names of the archive contents.

Note: For the purposes of showing you how to use different `tarfile` object methods, the TAR file in the examples is opened and closed manually in an interactive REPL session.

Interacting with the TAR file this way allows you to see the output of running each command. Normally, you would want to use a context manager to open file-like objects.

The metadata of each entry in the archive can be accessed using special attributes:

Python

>>>

```
>>> for entry in tar.getmembers():
...     print(entry.name)
...     print(' Modified:', time.ctime(entry.mtime))
...     print(' Size    :', entry.size, 'bytes')
...     print()
CONTRIBUTING.rst
Modified: Sat Nov  1 09:09:51 2018
Size    : 402 bytes

README.md
Modified: Sat Nov  3 07:29:40 2018
Size    : 5426 bytes

app.py
Modified: Sat Nov  3 07:29:13 2018
Size    : 6218 bytes
```

In this example, you loop through the list of files returned by `.getmembers()` and print out each file's attributes. The objects returned by `.getmembers()` have attributes that can be accessed programmatically such as the name, size, and last modified time of each of the files in the archive. After reading or writing to the archive, it must be closed to free up system resources.

Extracting Files From a TAR Archive

In this section, you'll learn how to extract files from TAR archives using the following methods:

- `.extract()`
- `.extractfile()`
- `.extractall()`

To extract a single file from a TAR archive, use `extract()`, passing in the filename:

Python

>>>

```
>>> tar.extract('README.md')
```

```
>>> tar.extract('README.md')
>>> os.listdir('.')
['README.md', 'example.tar']
```

The README.md file is extracted from the archive to the file system. Calling `os.listdir()` confirms that `README.md` file was successfully extracted into the current directory. To unpack or extract everything from the archive, use `.extractall()`:

```
Python >>>
>>> tar.extractall(path="extracted")
```

`.extractall()` has an optional path argument to specify where extracted files should go. Here, the archive is unpacked into the `extracted` directory. The following commands show that the archive was successfully extracted:

```
Shell
$ ls
example.tar  extracted  README.md

$ tree
.
├── example.tar
├── extracted
│   ├── app.py
│   ├── CONTRIBUTING.rst
│   └── README.md
└── README.md

1 directory, 5 files

$ ls extracted/
app.py  CONTRIBUTING.rst  README.md
```

To extract a file object for reading or writing, use `.extractfile()`, which takes a filename or `TarInfo` object to extract as an argument. `.extractfile()` returns a file-like object that can be read and used:

```
Python >>>
>>> f = tar.extractfile('app.py')
>>> f.read()
>>> tar.close()
```

Opened archives should always be closed after they have been read or written to. To close an archive, call `.close()` on the archive file handle or use the `with` statement when creating `tarfile` objects to automatically close the archive when you're done. This frees up system resources and writes any changes you made to the archive to the filesystem.

Creating New TAR Archives

Here's how you do it:

```
Python >>>
>>> import tarfile
```

```

>>> file_list = ['app.py', 'config.py', 'CONTRIBUTORS.md', 'tests.py']
>>> with tarfile.open('packages.tar', mode='w') as tar:
...     for file in file_list:
...         tar.add(file)

>>> # Read the contents of the newly created archive
>>> with tarfile.open('package.tar', mode='r') as t:
...     for member in t.getmembers():
...         print(member.name)
app.py
config.py
CONTRIBUTORS.md
tests.py

```

First, you make a list of files to be added to the archive so that you don't have to add each file manually.

The next line uses the `with` context manager to open a new archive called `packages.tar` in write mode. Opening an archive in write mode('w') enables you to write new files to the archive. Any existing files in the archive are deleted and a new archive is created.

After the archive is created and populated, the `with` context manager automatically closes it and saves it to the filesystem. The last three lines open the archive you just created and print out the names of the files contained in it.

To add new files to an existing archive, open the archive in append mode ('a'):

```

Python >>>
>>> with tarfile.open('package.tar', mode='a') as tar:
...     tar.add('foo.bar')

>>> with tarfile.open('package.tar', mode='r') as tar:
...     for member in tar.getmembers():
...         print(member.name)
app.py
config.py
CONTRIBUTORS.md
tests.py
foo.bar

```

Opening an archive in append mode allows you to add new files to it without deleting the ones already in it.

Working With Compressed Archives

`tarfile` can also read and write TAR archives compressed using gzip, bzip2, and lzma compression. To read or write to a compressed archive, use `tarfile.open()`, passing in the appropriate mode for the compression type.

For example, to read or write data to a TAR archive compressed using gzip, use the '`r:gz`' or '`w:gz`' modes respectively:

```

Python >>>
>>> files = ['app.py', 'config.py', 'tests.py']
>>> with tarfile.open('packages.tar.gz', mode='w:gz') as tar:
...     tar.add('app.py')
...     tar.add('config.py')
...     tar.add('tests.py')

>>> with tarfile.open('packages.tar.gz', mode='r:gz') as t:
...     for member in t.getmembers():
...         print(member.name)
app.py
config.py
tests.py

```

The '`w:gz`' mode opens the archive for gzip compressed writing and '`r:gz`' opens the archive for gzip compressed reading. Opening compressed archives in append mode is not possible. To add files to a compressed archive, you have to create a new archive.

An Easier Way of Creating Archives

The Python Standard Library also supports creating TAR and ZIP archives using the high-level methods in the `shutil` module. The archiving utilities in `shutil` allow you to create, read, and extract ZIP and TAR archives. These utilities rely on the lower level `tarfile` and `zipfile` modules.

Working With Archives Using `shutil.make_archive()`

`shutil.make_archive()` takes at least two arguments: the name of the archive and an archive format.

By default, it compresses all the files in the current directory into the archive format specified in the `format` argument. You can pass in an optional `root_dir` argument to compress files in a different directory.

`.make_archive()` supports the `zip`, `tar`, `bztar`, and `gztar` archive formats.

This is how to create a TAR archive using `shutil`:

Python

```
import shutil

# shutil.make_archive(base_name, format, root_dir)
shutil.make_archive('data/backup', 'tar', 'data/')
```

This copies everything in `data/` and creates an archive called `backup.tar` in the filesystem and returns its name. To extract the archive, call `.unpack_archive()`:

Python

```
shutil.unpack_archive('backup.tar', 'extract_dir/')
```

Calling `.unpack_archive()` and passing in an archive name and destination directory extracts the contents of `backup.tar` into `extract_dir/`. ZIP archives can be created and extracted in the same way.

Reading Multiple Files

Python supports reading data from multiple input streams or from a list of files through the `fileinput` module. This module allows you to loop over the contents of one or more text files quickly and easily. Here's the typical way `fileinput` is used:

Python

```
import fileinput
for line in fileinput.input()
    process(line)
```

`fileinput` gets its input from command line arguments passed to `sys.argv` by default.

Using `fileinput` to Loop Over Multiple Files

Let's use `fileinput` to build a crude version of the common UNIX utility `cat`. The `cat` utility reads files sequentially, writing them to standard output. When given more than one file in its command line arguments, `cat` will concatenate the text files and display the result in the terminal:

Python

```
# File: fileinput-example.py
import fileinput
import sys

files = fileinput.input()
for line in files:
    if fileinput.isfirstline():
        print(f'\n--- Reading {fileinput.filename()} ---')
    print(' -> ' + line, end='')
```

```
print()
```

Running this on two text files in my current directory produces the following output:

Shell

```
$ python3 fileinput-example.py bacon.txt cupcake.txt
--- Reading bacon.txt ---
-> Spicy jalapeno bacon ipsum dolor amet in in aute est qui enim aliquip,
-> irure cillum drumstick elit.
-> Doner jowl shank ea exercitation landjaeger incididunt ut porchetta.
-> Tenderloin bacon aliquip cupidatat chicken chuck quis anim et swine.
-> Tri-tip doner kevin cillum ham veniam cow hamburger.
-> Turkey pork loin cupidatat filet mignon capicola brisket cupim ad in.
-> Ball tip dolor do magna laboris nisi pancetta nostrud doner.

--- Reading cupcake.txt ---
-> Cupcake ipsum dolor sit amet candy I love cheesecake fruitcake.
-> Topping muffin cotton candy.
-> Gummies macaroon jujubes jelly beans marzipan.
```

`fileinput` allows you to retrieve more information about each line such as whether or not it is the first line (`.isfirstline()`), the line number (`.lineno()`), and the filename (`.filename()`). You can read more about it [here](#).

Conclusion

You now know how to use Python to perform the most common operations on files and groups of files. You've learned about the different built-in modules used to read, find, and manipulate them.

You're now equipped to use Python to:

- Get directory contents and file properties
- Create directories and directory trees
- Find patterns in filenames
- Create temporary files and directories
- Move, rename, copy, and delete files or directories
- Read and extract data from different types of archives
- Read multiple files simultaneously using `fileinput`

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Practical Recipes for Working With Files in Python](#)

About Vuyisile Ndlovu

Django developer and open source enthusiast.

[» More about Vuyisile](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[David](#)

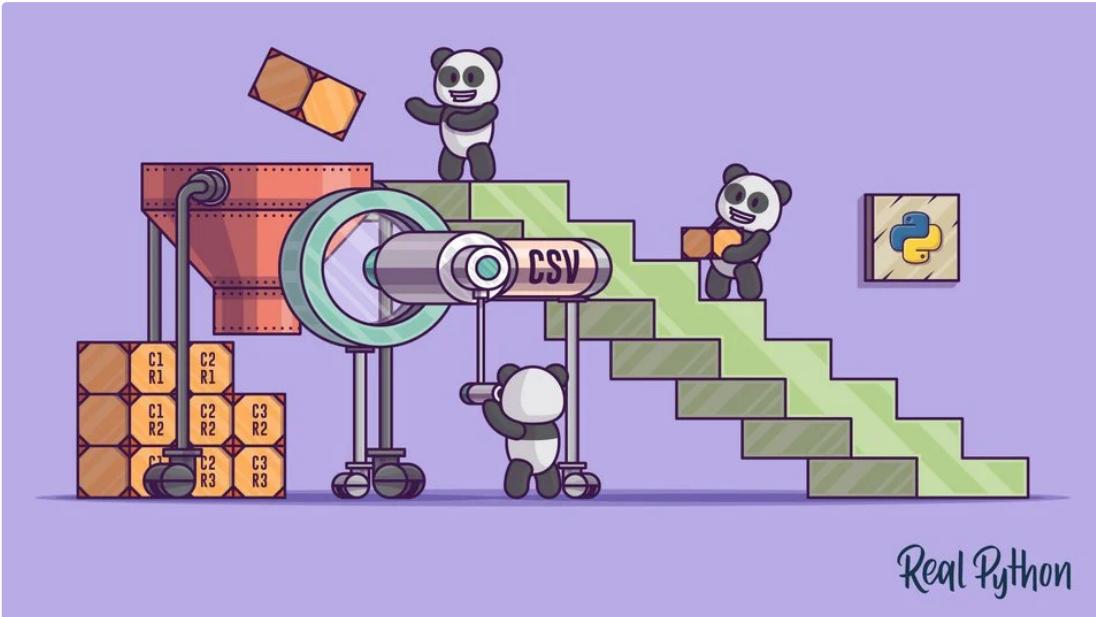
[Geir Arne](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)

Recommended Video Course: [Practical Recipes for Working With Files in Python](#)



Real Python

Reading and Writing CSV Files in Python

by [Jon Fincher](#) 72 Comments [data-science](#) [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Is a CSV File?](#)
 - [Where Do CSV Files Come From?](#)
- [Parsing CSV Files With Python's Built-in CSV Library](#)
 - [Reading CSV Files With csv](#)
 - [Reading CSV Files Into a Dictionary With csv](#)
 - [Optional Python CSV reader Parameters](#)
 - [Writing CSV Files With csv](#)
 - [Writing CSV File From a Dictionary With csv](#)
- [Parsing CSV Files With the pandas Library](#)
 - [Reading CSV Files With pandas](#)
 - [Writing CSV Files With pandas](#)
- [Conclusion](#)

[blackfire.io](#)
Profile & Optimize Python Apps Performance

Now available as
Public Beta
Sign-up for free and
install in minutes!

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Reading and Writing CSV Files](#)

Let's face it: you need to get information into and out of your programs through more than just the keyboard and console. Exchanging information through text files is a common way to share info between programs. One of the most popular formats for exchanging data is the CSV format. But how do you use it?

Let's get one thing clear: you don't have to (and you won't) build your own CSV parser from scratch. There are several perfectly acceptable libraries you can use. The Python [csv library](#) will work for most cases. If your work requires lots of data or numerical analysis, the [pandas library](#) has CSV parsing capabilities as well, which should handle the rest.

In this article, you'll learn how to read, process, and parse CSV from text files using Python. You'll see how CSV files work, learn the all-important csv library built into Python, and see how CSV parsing works using the pandas library.

So let's get started!

Free Download: [Get a sample chapter from Python Basics: A Practical Introduction to Python 3](#) to see how you can go from beginner to intermediate in Python with a complete curriculum, up-to-date for Python 3.8.

 **Take the Quiz:** Test your knowledge with our interactive “Reading and Writing CSV Files in Python” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

What Is a CSV File?

A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data. Because it's a plain text file, it can contain only actual text data—in other words, printable [ASCII](#) or [Unicode](#) characters.

The structure of a CSV file is given away by its name. Normally, CSV files use a comma to separate each specific data value. Here's what that structure looks like:

CSV

```
column 1 name, column 2 name, column 3 name
first row data 1, first row data 2, first row data 3
second row data 1, second row data 2, second row data 3
...
...
```

Notice how each piece of data is separated by a comma. Normally, the first line identifies each piece of data—in other words, the name of a data column. Every subsequent line after that is actual data and is limited only by file size constraints.

In general, the separator character is called a delimiter, and the comma is not the only one used. Other popular delimiters include the tab (\t), colon (:) and semi-colon (;) characters. Properly parsing a CSV file requires us to know which delimiter is being used.

Where Do CSV Files Come From?

CSV files are normally created by programs that handle large amounts of data. They are a convenient way to export data from spreadsheets and databases as well as import or use it in other programs. For example, you might export the results of a data mining program to a CSV file and then import that into a spreadsheet to analyze the data, generate graphs for a presentation, or prepare a report for publication.

CSV files are very easy to work with programmatically. Any language that supports text file input and string manipulation (like Python) can work with CSV files directly.

Parsing CSV Files With Python's Built-in CSV Library

The [csv library](#) provides functionality to both read from and write to CSV files. Designed to work out of the box with Excel-generated CSV files, it is easily adapted to work with a variety of CSV formats. The csv library contains objects and other code to read, write, and process data from and to CSV files.

Reading CSV Files With csv

Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python's built-in `open()` function, which returns a file object. This is then passed to the reader, which does the heavy lifting.

Here's the `employee_birthday.txt` file:

CSV

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's code to read it:

Python

```
import csv

with open('employee_birthday.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {" ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

This results in the following output:

Shell

```
Column names are name, department, birthday month
John Smith works in the Accounting department, and was born in November.
Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Each row returned by the reader is a list of String elements containing the data found by removing the delimiters. The first row returned contains the column names, which is handled in a special way.

Reading CSV Files Into a Dictionary With csv

Rather than deal with a list of individual string elements, you can read CSV data directly into a dictionary (technically, an [Ordered Dictionary](#)) as well.

Again, our input file, employee_birthday.txt is as follows:

CSV

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's the code to read it in as a dictionary this time:

Python

```
import csv

with open('employee_birthday.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {" ".join(row["name"])}')
            line_count += 1
        print(f'\t{row["name"]} works in the {row["department"]} department, and was born in {row["b"]}
```

```
line_count += 1
print(f'Processed {line_count} lines.')
```

This results in the same output as before:

Shell

```
Column names are name, department, birthday month
John Smith works in the Accounting department, and was born in November.
Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Where did the dictionary keys come from? The first line of the CSV file is assumed to contain the keys to use to build the dictionary. If you don't have these in your CSV file, you should specify your own keys by setting the `fieldnames` optional parameter to a list containing them.

Optional Python CSV reader Parameters

The reader object can handle different styles of CSV files by specifying [additional parameters](#), some of which are shown below:

- `delimiter` specifies the character used to separate each field. The default is the comma (',').
- `quotechar` specifies the character used to surround fields that contain the delimiter character. The default is a double quote (' " ').
- `escapechar` specifies the character used to escape the delimiter character, in case quotes aren't used. The default is no escape character.

These parameters deserve some more explanation. Suppose you're working with the following `employee_addresses.txt` file:

CSV

```
name,address,date joined
john smith,1132 Anywhere Lane Hoboken NJ, 07030,Jan 4
erica meyers,1234 Smith Lane Hoboken NJ, 07030,March 2
```

This CSV file contains three fields: `name`, `address`, and `date joined`, which are delimited by commas. The problem is that the data for the `address` field also contains a comma to signify the zip code.

There are three different ways to handle this situation:

- **Use a different delimiter**

That way, the comma can safely be used in the data itself. You use the `delimiter` optional parameter to specify the new delimiter.

- **Wrap the data in quotes**

The special nature of your chosen delimiter is ignored in quoted strings. Therefore, you can specify the character used for quoting with the `quotechar` optional parameter. As long as that character also doesn't appear in the data, you're fine.

- **Escape the delimiter characters in the data**

Escape characters work just as they do in format strings, nullifying the interpretation of the character being escaped (in this case, the delimiter). If an escape character is used, it must be specified using the `escapechar` optional parameter.

Writing CSV Files With csv

You can also write to a CSV file using a `writer` object and the `.write_row()` method:

Python

```
import csv

with open('employee_file.csv', mode='w') as employee_file:
```

```

employee_writer = csv.writer(employee_file, delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL)
employee_writer.writerow(['John Smith', 'Accounting', 'November'])
employee_writer.writerow(['Erica Meyers', 'IT', 'March'])

```

The quotechar optional parameter tells the writer which character to use to quote fields when writing. Whether quoting is used or not, however, is determined by the quoting optional parameter:

- If quoting is set to csv.QUOTE_MINIMAL, then .writerow() will quote fields only if they contain the delimiter or the quotechar. This is the default case.
- If quoting is set to csv.QUOTE_ALL, then .writerow() will quote all fields.
- If quoting is set to csv.QUOTE_NONNUMERIC, then .writerow() will quote all fields containing text data and convert all numeric fields to the float data type.
- If quoting is set to csv.QUOTE_NONE, then .writerow() will escape delimiters instead of quoting them. In this case, you also must provide a value for the escapechar optional parameter.

Reading the file back in plain text shows that the file is created as follows:

CSV

```

John Smith,Accounting,November
Erica Meyers,IT,March

```

Writing CSV File From a Dictionary With csv

Since you can read our data into a dictionary, it's only fair that you should be able to write it out from a dictionary as well:

Python

```

import csv

with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'emp_name': 'John Smith', 'dept': 'Accounting', 'birth_month': 'November'})
    writer.writerow({'emp_name': 'Erica Meyers', 'dept': 'IT', 'birth_month': 'March'})

```

Unlike DictReader, the fieldnames parameter is required when writing a dictionary. This makes sense, when you think about it: without a list of fieldnames, the DictWriter can't know which keys to use to retrieve values from your dictionaries. It also uses the keys in fieldnames to write out the first row as column names.

The code above generates the following output file:

CSV

```

emp_name,dept,birth_month
John Smith,Accounting,November
Erica Meyers,IT,March

```

Parsing CSV Files With the pandas Library

Of course, the Python CSV library isn't the only game in town. Reading CSV files is possible in [pandas](#) as well. It is highly recommended if you have a lot of data to analyze.

pandas is an open-source Python library that provides high performance data analysis tools and easy to use data structures. pandas is available for all Python installations, but it is a key part of the [Anaconda](#) distribution and works extremely well in [Jupyter notebooks](#) to share data, code, analysis results, visualizations, and narrative text.

Installing pandas and its dependencies in Anaconda is easily done:

Shell

```
$ conda install pandas
```

As is using [pip/pipenv](#) for other Python installations:

Shell

```
$ pip install pandas
```

We won't delve into the specifics of how pandas works or how to use it. For an in-depth treatment on using pandas to read and analyze large data sets, check out [Shantnu Tiwari's](#) superb article on [working with large Excel files in pandas](#).

Reading CSV Files With pandas

To show some of the power of pandas CSV capabilities, I've created a slightly more complicated file to read, called `hrdata.csv`. It contains data on company employees:

CSV

```
Name,Hire Date,Salary,Sick Days remaining
Graham Chapman,03/15/14,50000.00,10
John Cleese,06/01/15,65000.00,8
Eric Idle,05/12/14,45000.00,10
Terry Jones,11/01/13,70000.00,3
Terry Gilliam,08/12/14,48000.00,7
Michael Palin,05/23/13,66000.00,8
```

Reading the CSV into a pandas [DataFrame](#) is quick and straightforward:

Python

```
import pandas
df = pandas.read_csv('hrdata.csv')
print(df)
```

That's it: three lines of code, and only one of them is doing the actual work. `pandas.read_csv()` opens, analyzes, and reads the CSV file provided, and stores the data in a [DataFrame](#). Printing the [DataFrame](#) results in the following output:

Shell

	Name	Hire Date	Salary	Sick Days remaining
0	Graham Chapman	03/15/14	50000.0	10
1	John Cleese	06/01/15	65000.0	8
2	Eric Idle	05/12/14	45000.0	10
3	Terry Jones	11/01/13	70000.0	3
4	Terry Gilliam	08/12/14	48000.0	7
5	Michael Palin	05/23/13	66000.0	8

Here are a few points worth noting:

- First, pandas recognized that the first line of the CSV contained column names, and used them automatically. I call this Goodness.
- However, pandas is also using zero-based integer indices in the [DataFrame](#). That's because we didn't tell it what our index should be.
- Further, if you look at the data types of our columns , you'll see pandas has properly converted the `Salary` and `Sick Days remaining` columns to numbers, but the `Hire Date` column is still a `String`. This is easily confirmed in interactive mode:

Python

>>>

```
>>> print(type(df['Hire Date'][0]))  
<class 'str'>
```

Let's tackle these issues one at a time. To use a different column as the DataFrame index, add the `index_col` optional parameter:

Python

```
import pandas  
df = pandas.read_csv('hrdata.csv', index_col='Name')  
print(df)
```

Now the `Name` field is our DataFrame index:

Shell

	Hire Date	Salary	Sick Days	remaining
Name				
Graham Chapman	03/15/14	50000.0		10
John Cleese	06/01/15	65000.0		8
Eric Idle	05/12/14	45000.0		10
Terry Jones	11/01/13	70000.0		3
Terry Gilliam	08/12/14	48000.0		7
Michael Palin	05/23/13	66000.0		8

Next, let's fix the data type of the `Hire Date` field. You can force pandas to read data as a date with the `parse_dates` optional parameter, which is defined as a list of column names to treat as dates:

Python

```
import pandas  
df = pandas.read_csv('hrdata.csv', index_col='Name', parse_dates=['Hire Date'])  
print(df)
```

Notice the difference in the output:

Shell

	Hire Date	Salary	Sick Days	remaining
Name				
Graham Chapman	2014-03-15	50000.0		10
John Cleese	2015-06-01	65000.0		8
Eric Idle	2014-05-12	45000.0		10
Terry Jones	2013-11-01	70000.0		3
Terry Gilliam	2014-08-12	48000.0		7
Michael Palin	2013-05-23	66000.0		8

The date is now formatted properly, which is easily confirmed in interactive mode:

Python

>>>

```
>>> print(type(df['Hire Date'][0]))  
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

If your CSV files doesn't have column names in the first line, you can use the `names` optional parameter to provide a list of column names. You can also use this if you want to override the column names provided in the first line. In this case, you must also tell `pandas.read_csv()` to ignore existing column names using the `header=0` optional parameter:

Python

```
import pandas  
df = pandas.read_csv('hrdata.csv',
```

```

        index_col='Employee',
        parse_dates=['Hired'],
        header=0,
        names=['Employee', 'Hired', 'Salary', 'Sick Days'])
print(df)

```

Notice that, since the column names changed, the columns specified in the `index_col` and `parse_dates` optional parameters must also be changed. This now results in the following output:

Shell

	Hired	Salary	Sick Days
Employee			
Graham Chapman	2014-03-15	50000.0	10
John Cleese	2015-06-01	65000.0	8
Eric Idle	2014-05-12	45000.0	10
Terry Jones	2013-11-01	70000.0	3
Terry Gilliam	2014-08-12	48000.0	7
Michael Palin	2013-05-23	66000.0	8

Writing CSV Files With pandas

Of course, if you can't get your data out of pandas again, it doesn't do you much good. Writing a DataFrame to a CSV file is just as easy as reading one in. Let's write the data with the new column names to a new CSV file:

Python

```

import pandas
df = pandas.read_csv('hrdata.csv',
                     index_col='Employee',
                     parse_dates=['Hired'],
                     header=0,
                     names=['Employee', 'Hired', 'Salary', 'Sick Days'])
df.to_csv('hrdata_modified.csv')

```

The only difference between this code and the reading code above is that the `print(df)` call was replaced with `df.to_csv()`, providing the file name. The new CSV file looks like this:

Shell

```

Employee,Hired,Salary,Sick Days
Graham Chapman,2014-03-15,50000.0,10
John Cleese,2015-06-01,65000.0,8
Eric Idle,2014-05-12,45000.0,10
Terry Jones,2013-11-01,70000.0,3
Terry Gilliam,2014-08-12,48000.0,7
Michael Palin,2013-05-23,66000.0,8

```

Conclusion

If you understand the basics of reading CSV files, then you won't ever be caught flat footed when you need to deal with importing data. Most CSV reading, processing, and writing tasks can be easily handled by the basic csv Python library. If you have a lot of data to read and process, the pandas library provides quick and easy CSV handling capabilities as well.

 **Take the Quiz:** Test your knowledge with our interactive “Reading and Writing CSV Files in Python” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

Are there other ways to parse text files? Of course! Libraries like [ANTLR](#), [PLY](#), and [PlyPlus](#) can all handle heavy-duty parsing, and if simple string manipulation won't work, there are always regular expressions.

But those are topics for other articles...

Free Download: Get a sample chapter from Python Basics: A Practical Introduction to Python 3 to see how you can go from beginner to intermediate in Python with a complete curriculum, up-to-date for Python 3.8.

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Reading and Writing CSV Files](#)

About Jon Fincher

Jon taught Python and Java in two high schools in Washington State. Previously, he was a Program Manager at Microsoft.

[» More about Jon](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

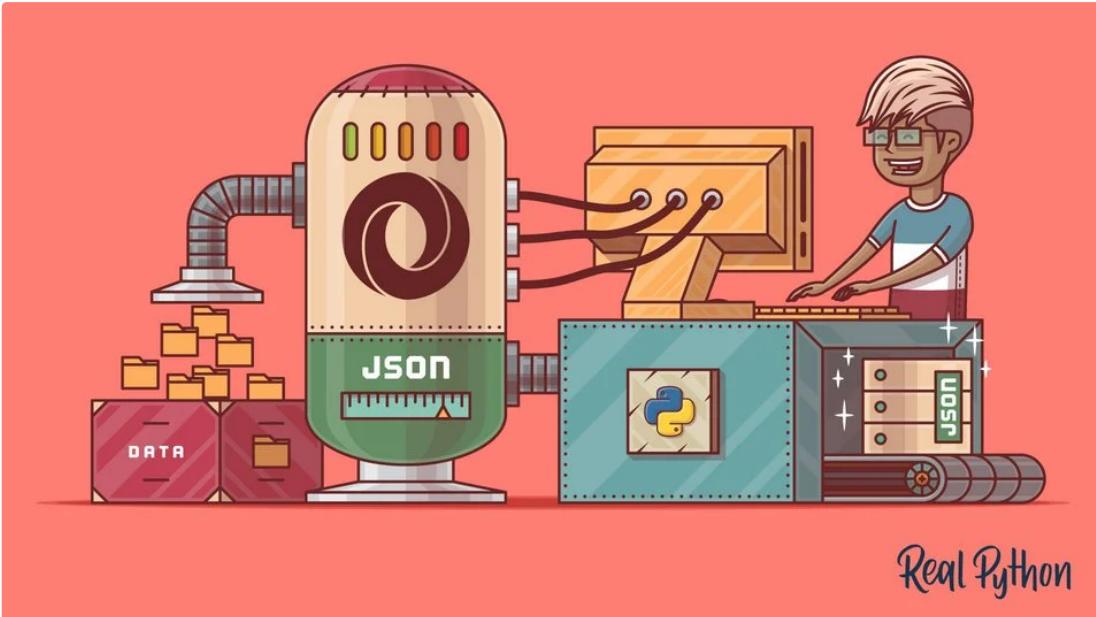
[Joanna](#)

[Jason](#)

Keep Learning

Related Tutorial Categories: [data-science](#) [intermediate](#) [python](#)

Recommended Video Course: [Reading and Writing CSV Files](#)



Real Python

Working With JSON Data in Python

by [Lucas Lofaro](#) 39 Comments [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [A \(Very\) Brief History of JSON](#)
- [Look, it's JSON!](#)
- [Python Supports JSON Natively!](#)
 - [A Little Vocabulary](#)
 - [Serializing JSON](#)
 - [A Simple Serialization Example](#)
 - [Some Useful Keyword Arguments](#)
 - [Deserializing JSON](#)
 - [A Simple Deserialization Example](#)
- [A Real World Example \(sort of\)](#)
- [Encoding and Decoding Custom Python Objects](#)
 - [Simplifying Data Structures](#)
 - [Encoding Custom Types](#)
 - [Decoding Custom Types](#)
- [All done!](#)

Python Tricks The Book

A Buffet of Awesome Python Features

[Get Your Free Sample Chapter](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Working With JSON Data in Python](#)

Since its inception, [JSON](#) has quickly become the de facto standard for information exchange. Chances are you're here because you need to transport some data from here to there. Perhaps you're gathering information through an [API](#) or storing your data in a [document database](#). One way or another, you're up to your neck in JSON, and you've got to Python your way out.

Luckily, this is a pretty common task, and—as with most common tasks—Python makes it almost disgustingly easy. Have no fear, fellow Pythoneers and Pythonistas. This one's gonna be a breeze!

So, we use JSON to store and exchange data? Yup, you got it! It's nothing more than a standardized format the community uses to pass data around. Keep in mind, JSON isn't the only format available for this kind of work, but [XML](#) and [YAML](#) are probably the only other ones worth mentioning in the same breath.

[Free PDF Download: Python 3 Cheat Sheet](#)

[Remove ads](#)

A (Very) Brief History of JSON

Not so surprisingly, **JavaScript Object Notation** was inspired by a subset of the [JavaScript programming language](#) dealing with object literal syntax. They've got a [nifty website](#) that explains the whole thing. Don't worry though: JSON has long since become language agnostic and exists as [its own standard](#), so we can thankfully avoid JavaScript for the sake of this discussion.

Ultimately, the community at large adopted JSON because it's easy for both humans and machines to create and understand.

Look, it's JSON!

Get ready. I'm about to show you some real life JSON—just like you'd see out there in the wild. It's okay: JSON is supposed to be readable by anyone who's used a C-style language, and Python is a C-style language...so that's you!

JSON

```
{  
    "firstName": "Jane",  
    "lastName": "Doe",  
    "hobbies": ["running", "sky diving", "singing"],  
    "age": 35,  
    "children": [  
        {  
            "firstName": "Alice",  
            "age": 6  
        },  
        {  
            "firstName": "Bob",  
            "age": 8  
        }  
    ]  
}
```

As you can see, JSON supports primitive types, like strings and numbers, as well as nested lists and objects.

Wait, that looks like a Python dictionary! I know, right? It's pretty much universal object notation at this point, but I don't think UON rolls off the tongue quite as nicely. Feel free to discuss alternatives in the comments.

Whew! You survived your first encounter with some wild JSON. Now you just need to learn how to tame it.

Python Supports JSON Natively!

Python comes with a built-in package called [json](#) for encoding and decoding JSON data.

Just throw this little guy up at the top of your file:

Python

```
import json
```

A Little Vocabulary

The process of encoding JSON is usually called **serialization**. This term refers to the transformation of data into a *series of bytes* (hence *serial*) to be stored or transmitted across a network. You may also hear the term **marshaling**, but that's a [whole other discussion](#). Naturally, **deserialization** is the reciprocal process of decoding data that has been stored or delivered in the JSON standard.

Yikes! That sounds pretty technical. Definitely. But in reality, all we're talking about here is *reading* and *writing*. Think of it like this: *encoding* is for *writing* data to disk, while *decoding* is for *reading* data into memory.

Serializing JSON

What happens after a computer processes lots of information? It needs to take a data dump. Accordingly, the `json` library exposes the `dump()` method for writing data to files. There is also a `dumps()` method (pronounced as "dump-s") for writing to a Python string.

Simple Python objects are translated to JSON according to a fairly intuitive conversion.

Python	JSON
<code>dict</code>	<code>object</code>
<code>list, tuple</code>	<code>array</code>
<code>str</code>	<code>string</code>
<code>int, long, float</code>	<code>number</code>
<code>True</code>	<code>true</code>
<code>False</code>	<code>false</code>
<code>None</code>	<code>null</code>

[Remove ads](#)

A Simple Serialization Example

Imagine you're working with a Python object in memory that looks a little something like this:

```
Python
data = {
    "president": {
        "name": "Zaphod Beeblebrox",
        "species": "Betelgeusian"
    }
}
```

It is critical that you save this information to disk, so your mission is to write it to a file.

Using Python's context manager, you can create a file called `data_file.json` and open it in write mode. (JSON files conveniently end in a `.json` extension.)

```
Python
with open("data_file.json", "w") as write_file:
    json.dump(data, write_file)
```

Note that `dump()` takes two positional arguments: (1) the data object to be serialized, and (2) the file-like object to which the bytes will be written.

Or, if you were so inclined as to continue using this serialized JSON data in your program, you could write it to a native Python `str` object.

Python

```
json_string = json.dumps(data)
```

Notice that the file-like object is absent since you aren't actually writing to disk. Other than that, `dumps()` is just like `dump()`.

Hooray! You've birthed some baby JSON, and you're ready to release it out into the wild to grow big and strong.

Some Useful Keyword Arguments

Remember, JSON is meant to be easily readable by humans, but readable syntax isn't enough if it's all squished together. Plus you've probably got a different programming style than me, and it might be easier for you to read code when it's formatted to your liking.

NOTE: Both the `dump()` and `dumps()` methods use the same keyword arguments.

The first option most people want to change is whitespace. You can use the `indent` keyword argument to specify the indentation size for nested structures. Check out the difference for yourself by using `data`, which we defined above, and running the following commands in a console:

Python

>>>

```
>>> json.dumps(data)
>>> json.dumps(data, indent=4)
```

Another formatting option is the `separators` keyword argument. By default, this is a 2-tuple of the separator strings `(", ", " : ")`, but a common alternative for compact JSON is `(", ", " :")`. Take a look at the sample JSON again to see where these separators come into play.

There are others, like `sort_keys`, but I have no idea what that one does. You can find a whole list in the [docs](#) if you're curious.

Deserializing JSON

Great, looks like you've captured yourself some wild JSON! Now it's time to whip it into shape. In the `json` library, you'll find `load()` and `loads()` for turning JSON encoded data into Python objects.

Just like serialization, there is a simple conversion table for deserialization, though you can probably guess what it looks like already.

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True

false	False
JSON	Python
null	None

Technically, this conversion isn't a perfect inverse to the serialization table. That basically means that if you encode an object now and then decode it again later, you may not get exactly the same object back. I imagine it's a bit like teleportation: break my molecules down over here and put them back together over there. Am I still the same person?

In reality, it's probably more like getting one friend to translate something into Japanese and another friend to translate it back into English. Regardless, the simplest example would be encoding a tuple and getting back a list after decoding, like so:

```
Python >>>
>>> blackjack_hand = (8, "Q")
>>> encoded_hand = json.dumps(blackjack_hand)
>>> decoded_hand = json.loads(encoded_hand)

>>> blackjack_hand == decoded_hand
False
>>> type(blackjack_hand)
<class 'tuple'>
>>> type(decoded_hand)
<class 'list'>
>>> blackjack_hand == tuple(decoded_hand)
True
```

[Remove ads](#)

A Simple Deserialization Example

This time, imagine you've got some data stored on disk that you'd like to manipulate in memory. You'll still use the context manager, but this time you'll open up the existing `data_file.json` in read mode.

```
Python
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)
```

Things are pretty straightforward here, but keep in mind that the result of this method could return any of the allowed data types from the conversion table. This is only important if you're loading in data you haven't seen before. In most cases, the root object will be a dict or a list.

If you've pulled JSON data in from another program or have otherwise obtained a string of JSON formatted data in Python, you can easily deserialize that with `loads()`, which naturally loads from a string:

```
Python
json_string = """
{
    "researcher": {
        "name": "Ford Prefect",
        "species": "Betelgeusian".
```

```

        "relatives": [
            {
                "name": "Zaphod Beeblebrox",
                "species": "Betelgeusian"
            }
        ]
    }
"""

data = json.loads(json_string)

```

Voilà! You've tamed the wild JSON, and now it's under your control. But what you do with that power is up to you. You could feed it, nurture it, and even teach it tricks. It's not that I don't trust you...but keep it on a leash, okay?

A Real World Example (sort of)

For your introductory example, you'll use [JSONPlaceholder](#), a great source of fake JSON data for practice purposes.

First create a script file called scratch.py, or whatever you want. I can't really stop you.

You'll need to make an API request to the JSONPlaceholder service, so just use the [requests](#) package to do the heavy lifting. Add these imports at the top of your file:

Python

```
import json
import requests
```

Now, you're going to be working with a list of TODOs cuz like...you know, it's a rite of passage or whatever.

Go ahead and make a request to the JSONPlaceholder API for the /todos endpoint. If you're unfamiliar with requests, there's actually a handy json() method that will do all of the work for you, but you can practice using the json library to deserialize the text attribute of the response object. It should look something like this:

Python

```
response = requests.get("https://jsonplaceholder.typicode.com/todos")
todos = json.loads(response.text)
```

You don't believe this works? Fine, run the file in interactive mode and test it for yourself. While you're at it, check the type of todos. If you're feeling adventurous, take a peek at the first 10 or so items in the list.

Python

>>>

```
>>> todos == response.json()
True
>>> type(todos)
<class 'list'>
>>> todos[:10]
...
```

See, I wouldn't lie to you, but I'm glad you're a skeptic.

What's interactive mode? Ah, I thought you'd never ask! You know how you're always jumping back and forth between the your editor and the terminal? Well, us sneaky Pythoneers use the -i interactive flag when we run the script. This is a great little trick for testing code because it runs the script and then opens up an interactive command prompt with access to all the data from the script!

All right, time for some action. You can see the structure of the data by visiting the [endpoint](#) in a browser, but here's a sample TODO:

JSON

r

```

        "userId": 1,
        "id": 1,
        "title": "delectus aut autem",
        "completed": false
    }

```

There are multiple users, each with a unique `userId`, and each task has a Boolean `completed` property. Can you determine which users have completed the most tasks?

Python

```

# Map of userId to number of complete TODOs for that user
todos_by_user = {}

# Increment complete TODOs count for each user.
for todo in todos:
    if todo["completed"]:
        try:
            # Increment the existing user's count.
            todos_by_user[todo["userId"]] += 1
        except KeyError:
            # This user has not been seen. Set their count to 1.
            todos_by_user[todo["userId"]] = 1

# Create a sorted list of (userId, num_complete) pairs.
top_users = sorted(todos_by_user.items(),
                    key=lambda x: x[1], reverse=True)

# Get the maximum number of complete TODOs.
max_complete = top_users[0][1]

# Create a list of all users who have completed
# the maximum number of TODOs.
users = []
for user, num_complete in top_users:
    if num_complete < max_complete:
        break
    users.append(str(user))

max_users = " and ".join(users)

```

Yeah, yeah, your implementation is better, but the point is, you can now manipulate the JSON data as a normal Python object!

I don't know about you, but when I run the script interactively again, I get the following results:

Python

>>>

```

>>> s = "s" if len(users) > 1 else ""
>>> print(f"user{s} {max_users} completed {max_complete} TODOs")
users 5 and 10 completed 12 TODOs

```

That's cool and all, but you're here to learn about JSON. For your final task, you'll create a JSON file that contains the `completed` TODOs for each of the users who completed the maximum number of TODOs.

All you need to do is filter todos and write the resulting list to a file. For the sake of originality, you can call the output file `filtered_data_file.json`. There are many ways you could go about this, but here's one:

Python

```

# Define a function to filter out completed TODOs
# of users with max completed TODOs.
def keep(todo):
    is_complete = todo["completed"]
    has_max_count = str(todo["userId"]) in users
    return is_complete and has_max_count

# Write filtered TODOs to file.
with open("filtered_data_file.json", "w") as data_file:
    filtered.todos = list(filter(keep, todos))
    json.dump(filtered.todos, data_file, indent=2)

```

Perfect, you've gotten rid of all the data you don't need and saved the good stuff to a brand new file! Run the script again and check out `filtered_data_file.json` to verify everything worked. It'll be in the same directory as `scratch.py` when you run it.

Now that you've made it this far, I bet you're feeling like some pretty hot stuff, right? Don't get cocky: humility is a virtue. I am inclined to agree with you though. So far, it's been smooth sailing, but you might want to batten down the hatches for this last leg of the journey.

[Remove ads](#)

Encoding and Decoding Custom Python Objects

What happens when we try to serialize the `Elf` class from that Dungeons & Dragons app you're working on?

Python

```
class Elf:
    def __init__(self, level, ability_scores=None):
        self.level = level
        self.ability_scores = {
            "str": 11, "dex": 12, "con": 10,
            "int": 16, "wis": 14, "cha": 13
        } if ability_scores is None else ability_scores
        self.hp = 10 + self.ability_scores["con"]
```

Not so surprisingly, Python complains that `Elf` isn't *serializable* (which you'd know if you've ever tried to tell an Elf otherwise):

Python

>>>

```
>>> elf = Elf(level=4)
>>> json.dumps(elf)
TypeError: Object of type 'Elf' is not JSON serializable
```

Although the `json` module can handle most built-in Python types, it doesn't understand how to encode customized data types by default. It's like trying to fit a square peg in a round hole—you need a buzzsaw and parental supervision.

Simplifying Data Structures

Now, the question is how to deal with more complex data structures. Well, you could try to encode and decode the JSON by hand, but there's a slightly more clever solution that'll save you some work. Instead of going straight from the custom data type to JSON, you can throw in an intermediary step.

All you need to do is represent your data in terms of the built-in types `json` already understands. Essentially, you translate the more complex object into a simpler representation, which the `json` module then translates into JSON. It's like the transitive property in mathematics: if $A = B$ and $B = C$, then $A = C$.

To get the hang of this, you'll need a complex object to play with. You could use any custom class you like, but Python has a built-in type called `complex` for representing complex numbers, and it isn't serializable by default. So, for the sake of these examples, your complex object is going to be a `complex` object. Confused yet?

Python

>>>

```
>>> z = 3 + 8j
>>> type(z)
<class 'complex'>
>>> json.dumps(z)
TypeError: Object of type 'complex' is not JSON serializable
```

Where do complex numbers come from? You see, when a real number and an imaginary number love each other very much, they add together to produce a number which is (justifiably) called [complex](#).

A good question to ask yourself when working with custom types is **What is the minimum amount of information necessary to recreate this object?** In the case of complex numbers, you only need to know the real and imaginary parts, both of which you can access as attributes on the complex object:

Python

```
>>> z.real
3.0
>>> z.imag
8.0
```

>>>

Passing the same numbers into a complex constructor is enough to satisfy the `__eq__` comparison operator:

Python

```
>>> complex(3, 8) == z
True
```

>>>

Breaking custom data types down into their essential components is critical to both the serialization and deserialization processes.

Encoding Custom Types

To translate a custom object into JSON, all you need to do is provide an encoding function to the `dump()` method's `default` parameter. The `json` module will call this function on any objects that aren't natively serializable. Here's a simple decoding function you can use for practice:

Python

```
def encode_complex(z):
    if isinstance(z, complex):
        return (z.real, z.imag)
    else:
        type_name = z.__class__.__name__
        raise TypeError(f"Object of type '{type_name}' is not JSON serializable")
```

Notice that you're expected to raise a `TypeError` if you don't get the kind of object you were expecting. This way, you avoid accidentally serializing any Elves. Now you can try encoding complex objects for yourself!

Python

```
>>> json.dumps(9 + 5j, default=encode_complex)
'[9.0, 5.0]'
>>> json.dumps(elf, default=encode_complex)
TypeError: Object of type 'Elf' is not JSON serializable
```

>>>

Why did we encode the complex number as a tuple? Great question! That certainly wasn't the only choice, nor is it necessarily the best choice. In fact, this wouldn't be a very good representation if you ever wanted to decode the object later, as you'll see shortly.

The other common approach is to subclass the standard `JSONEncoder` and override its `default()` method:

Python

```
class ComplexEncoder(json.JSONEncoder):
    def default(self, z):
        if isinstance(z, complex):
```

```

    if isinstance(z, Complex):
        return (z.real, z.imag)
    else:
        return super().default(z)

```

Instead of raising the `TypeError` yourself, you can simply let the base class handle it. You can use this either directly in the `dump()` method via the `cls` parameter or by creating an instance of the encoder and calling its `encode()` method:

```

Python >>>
>>> json.dumps(2 + 5j, cls=ComplexEncoder)
'[2.0, 5.0]'

>>> encoder = ComplexEncoder()
>>> encoder.encode(3 + 6j)
'[3.0, 6.0]'

```

[Remove ads](#)

Decoding Custom Types

While the real and imaginary parts of a complex number are absolutely necessary, they are actually not quite sufficient to recreate the object. This is what happens when you try encoding a complex number with the `ComplexEncoder` and then decoding the result:

```

Python >>>
>>> complex_json = json.dumps(4 + 17j, cls=ComplexEncoder)
>>> json.loads(complex_json)
[4.0, 17.0]

```

All you get back is a list, and you'd have to pass the values into a complex constructor if you wanted that complex object again. Recall our discussion about [teleportation](#). What's missing is *metadata*, or information about the type of data you're encoding.

I suppose the question you really ought ask yourself is **What is the minimum amount of information that is both necessary and sufficient to recreate this object?**

The `json` module expects all custom types to be expressed as objects in the JSON standard. For variety, you can create a JSON file this time called `complex_data.json` and add the following object representing a complex number:

```

JSON
{
    "__complex__": true,
    "real": 42,
    "imag": 36
}

```

See the clever bit? That `"__complex__"` key is the metadata we just talked about. It doesn't really matter what the associated value is. To get this little hack to work, all you need to do is verify that the key exists:

```

Python
def decode_complex(dct):
    if "__complex__" in dct:

```

```
    return complex(dct["real"], dct["imag"])
return dct
```

If `"__complex__"` isn't in the dictionary, you can just return the object and let the default decoder deal with it.

Every time the `load()` method attempts to parse an object, you are given the opportunity to intercede before the default decoder has its way with the data. You can do this by passing your decoding function to the `object_hook` parameter.

Now play the same kind of game as before:

```
Python >>>
>>> with open("complex_data.json") as complex_data:
...     data = complex_data.read()
...     z = json.loads(data, object_hook=decode_complex)
...
>>> type(z)
<class 'complex'>
```

While `object_hook` might feel like the counterpart to the `dump()` method's `default` parameter, the analogy really begins and ends there.

This doesn't just work with one object either. Try putting this list of complex numbers into `complex_data.json` and running the script again:

```
JSON
[{"__complex__":true,"real":42,"imag":36}, {"__complex__":true,"real":64,"imag":11}]
```

If all goes well, you'll get a list of `complex` objects:

```
Python >>>
>>> with open("complex_data.json") as complex_data:
...     data = complex_data.read()
...     numbers = json.loads(data, object_hook=decode_complex)
...
>>> numbers
[(42+36j), (64+11j)]
```

You could also try subclassing `JSONDecoder` and overriding `object_hook`, but it's better to stick with the lightweight solution whenever possible.

All done!

Congratulations, you can now wield the mighty power of JSON for any and all of your nefarious Python needs.

While the examples you've worked with here are certainly contrived and overly simplistic, they illustrate a workflow you can apply to more general tasks:

1. [Import](#) the `json` package.
2. Read the data with `load()` or `loads()`.
3. Process the data.
4. Write the altered data with `dump()` or `dumps()`.

What you do with your data once it's been loaded into memory will depend on your use case. Generally, your goal will be gathering data from a source, extracting useful information, and passing that information along or keeping a record of it.

Today you took a journey: you captured and tamed some wild JSON, and you made it back in time for supper! As an added bonus, learning the `json` package will make learning [pickle](#) and [marshal](#) a snap.

Good luck with all of your future Pythonic endeavors!

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Working With JSON Data in Python](#)

About Lucas Lofaro

Lucas is a wandering Pythoneer with a curious mind and a desire to spread knowledge to those who seek it.

[» More about Lucas](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

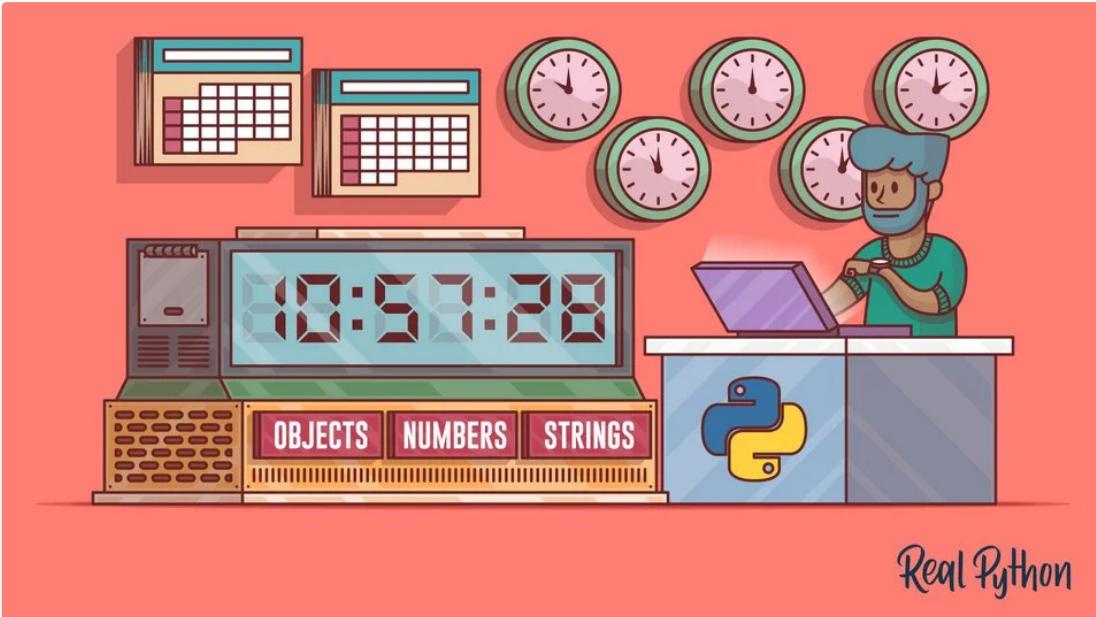
[Dan](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)

Recommended Video Course: [Working With JSON Data in Python](#)



Real Python

A Beginner's Guide to the Python time Module

by Alex Ronquillo 12 Comments Intermediate

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Dealing With Python Time Using Seconds](#)
 - [The Epoch](#)
 - [Python Time in Seconds as a Floating Point Number](#)
 - [Python Time in Seconds as a String Representing Local Time](#)
- [Understanding Time Zones](#)
 - [UTC and Time Zones](#)
 - [Daylight Savings Time](#)
- [Dealing With Python Time Using Data Structures](#)
 - [Python Time as a Tuple](#)
 - [Python Time as an Object](#)
- [Converting Python Time in Seconds to an Object](#)
 - [Coordinated Universal Time \(UTC\)](#)
 - [Local Time](#)
- [Converting a Local Time Object to Seconds](#)
- [Converting a Python Time Object to a String](#)
 - [asctime\(\)](#)
 - [strftime\(\)](#)
- [Converting a Python Time String to an Object](#)
- [Suspending Execution](#)
- [Measuring Performance](#)
- [Conclusion](#)
- [Further Reading](#)



Profile & Optimize Python Apps Performance



Now available as
Public Beta
Sign-up for free and
install in minutes!



This tutorial has a related video course created by the Real Python team. Watch it together with

the written tutorial to deepen your understanding: [Mastering Python's Built-in time Module](#)

The Python `time` module provides many ways of representing time in code, such as objects, numbers, and strings. It also provides functionality other than representing time, like waiting during code execution and measuring the efficiency of your code.

This article will walk you through the most commonly used functions and objects in `time`.

By the end of this article, you'll be able to:

- **Understand** core concepts at the heart of working with dates and times, such as epochs, time zones, and daylight savings time
- **Represent** time in code using floats, tuples, and `struct_time`
- **Convert** between different time representations
- **Suspend** thread execution
- **Measure** code performance using `perf_counter()`

You'll start by learning how you can use a floating point number to represent time.

Free Bonus: [Click here to get our free Python Cheat Sheet](#) that shows you the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Dealing With Python Time Using Seconds

One of the ways you can manage the concept of Python time in your application is by using a floating point number that represents the number of seconds that have passed since the beginning of an era—that is, since a certain starting point.

Let's dive deeper into what that means, why it's useful, and how you can use it to implement logic, based on Python time, in your application.

The Epoch

You learned in the previous section that you can manage Python time with a floating point number representing elapsed time since the beginning of an era.

[Merriam-Webster](#) defines an era as:

- A fixed point in time from which a series of years is reckoned
- A system of chronological notation computed from a given date as basis

The important concept to grasp here is that, when dealing with Python time, you're considering a period of time identified by a starting point. In computing, you call this starting point the **epoch**.

The epoch, then, is the starting point against which you can measure the passage of time.

For example, if you define the epoch to be midnight on January 1, 1970 UTC—the epoch as defined on Windows and most UNIX systems—then you can represent midnight on January 2, 1970 UTC as 86400 seconds since the epoch.

This is because there are 60 seconds in a minute, 60 minutes in an hour, and 24 hours in a day. January 2, 1970 UTC is only one day after the epoch, so you can apply basic math to arrive at that result:

```
Python >>> 60 * 60 * 24  
86400
```

It is also important to note that you can still represent time before the epoch. The number of seconds would just be negative.

For example, you would represent midnight on December 31, 1969 UTC (using an epoch of January 1, 1970) as -86400 seconds.

While January 1, 1970 UTC is a common epoch, it is not the only epoch used in computing. In fact, different operating systems, filesystems, and APIs sometimes use different epochs.

As you saw before, UNIX systems define the epoch as January 1, 1970. The Win32 API, on the other hand, defines the epoch as [January 1, 1601](#).

You can use `time.gmtime()` to determine your system's epoch:

```
Python >>>
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3, tm_yday=3)
```

You'll learn about `gmtime()` and `struct_time` throughout the course of this article. For now, just know that you can use `time` to discover the epoch using this function.

Now that you understand more about how to measure time in seconds using an epoch, let's take a look at Python's `time` module to see what functions it offers that help you do so.

Python Time in Seconds as a Floating Point Number

First, `time.time()` returns the number of seconds that have passed since the epoch. The return value is a floating point number to account for fractional seconds:

```
Python >>>
>>> from time import time
>>> time()
1551143536.9323719
```

The number you get on your machine may be very different because the reference point considered to be the epoch may be very different.

Further Reading: Python 3.7 introduced `time_ns()`, which returns an integer value representing the same elapsed time since the epoch, but in nanoseconds rather than seconds.

Measuring time in seconds is useful for a number of reasons:

- You can use a float to calculate the difference between two points in time.
- A float is easily [serializable](#), meaning that it can be stored for data transfer and come out intact on the other side.

Sometimes, however, you may want to see the current time represented as a string. To do so, you can pass the number of seconds you get from `time()` into `time.ctime()`.

Python Time in Seconds as a String Representing Local Time

As you saw before, you may want to convert the Python time, represented as the number of elapsed seconds since the epoch, to a [string](#). You can do so using `ctime()`:

```
Python >>>
>>> from time import time, ctime
>>> t = time()
>>> ctime(t)
'Mon Feb 25 19:11:59 2019'
```

Here, you've recorded the current time in seconds into the variable `t`, then passed `t` as an argument to `ctime()`, which returns a string representation of that same time.

Technical Detail: The argument, representing seconds since the epoch, is optional according to the `ctime()` definition. If you don't pass an argument, then `ctime()` uses the return value of `time()` by default. So, you could simplify the example above:

Python

>>>

```
>>> from time import ctime  
>>> ctime()  
'Mon Feb 25 19:11:59 2019'
```

The string representation of time, also known as a **timestamp**, returned by `ctime()` is formatted with the following structure:

1. **Day of the week:** Mon (Monday)
2. **Month of the year:** Feb (February)
3. **Day of the month:** 25
4. **Hours, minutes, and seconds using the 24-hour clock notation:** 19:11:59
5. **Year:** 2019

The previous example displays the timestamp of a particular moment captured from a computer in the South Central region of the United States. But, let's say you live in Sydney, Australia, and you executed the same command at the same instant.

Instead of the above output, you'd see the following:

Python

>>>

```
>>> from time import time, ctime  
>>> t = time()  
>>> ctime(t)  
'Tue Feb 26 12:11:59 2019'
```

Notice that the day of week, day of month, and hour portions of the timestamp are different than the first example.

These outputs are different because the timestamp returned by `ctime()` depends on your geographical location.

Note: While the concept of time zones is relative to your physical location, you can modify this in your computer's settings without actually relocating.

The representation of time dependent on your physical location is called **local time** and makes use of a concept called **time zones**.

Note: Since local time is related to your locale, timestamps often account for locale-specific details such as the order of the elements in the string and translations of the day and month abbreviations. `ctime()` ignores these details.

Let's dig a little deeper into the notion of time zones so that you can better understand Python time representations.

Understanding Time Zones

A time zone is a region of the world that conforms to a standardized time. Time zones are defined by their offset from Coordinated Universal Time (UTC) and, potentially, the inclusion of daylight savings time (which we'll cover in more detail later in this article).

Fun Fact: If you're a native English speaker, you might be wondering why the abbreviation for "Coordinated Universal Time" is UTC rather than the more obvious CUT. However, if you're a native French speaker, you would call it "Temps Universel Coordonné," which suggests a different abbreviation: TUC.

Ultimately, the International Telecommunication Union and the International Astronomical Union compromised on UTC as the official abbreviation so that, regardless of language, the abbreviation would be the same.

UTC and Time Zones

UTC is the time standard against which all the world's timekeeping is synchronized (or coordinated). It is not, itself, a time zone but rather a transcendent standard that defines what time zones are.

UTC time is precisely measured using [astronomical time](#), referring to the Earth's rotation, and [atomic clocks](#).

Time zones are then defined by their offset from UTC. For example, in North and South America, the Central Time Zone (CT) is behind UTC by five or six hours and, therefore, uses the notation UTC-5:00 or UTC-6:00.

Sydney, Australia, on the other hand, belongs to the Australian Eastern Time Zone (AET), which is ten or eleven hours ahead of UTC (UTC+10:00 or UTC+11:00).

This difference (UTC-6:00 to UTC+10:00) is the reason for the variance you observed in the two outputs from `ctime()` in the previous examples:

- **Central Time (CT):** 'Mon Feb 25 19:11:59 2019'
- **Australian Eastern Time (AET):** 'Tue Feb 26 12:11:59 2019'

These times are exactly sixteen hours apart, which is consistent with the time zone offsets mentioned above.

You may be wondering why CT can be either five or six hours behind UTC or why AET can be ten or eleven hours ahead. The reason for this is that some areas around the world, including parts of these time zones, observe daylight savings time.

Daylight Savings Time

Summer months generally experience more [daylight hours](#) than winter months. Because of this, some areas observe daylight savings time (DST) during the spring and summer to make better use of those daylight hours.

For places that observe DST, their clocks will jump ahead one hour at the beginning of spring (effectively losing an hour). Then, in the fall, the clocks will be reset to standard time.

The letters S and D represent standard time and daylight savings time in time zone notation:

- Central Standard Time (CST)
- Australian Eastern Daylight Time (AEDT)

When you represent times as timestamps in local time, it is always important to consider whether DST is applicable or not.

`ctime()` accounts for daylight savings time. So, the output difference listed previously would be more accurate as the following:

- **Central Standard Time (CST):** 'Mon Feb 25 19:11:59 2019'
- **Australian Eastern Daylight Time (AEDT):** 'Tue Feb 26 12:11:59 2019'

Dealing With Python Time Using Data Structures

Now that you have a firm grasp on many fundamental concepts of time including epochs, time zones, and UTC, let's take a look at more ways to represent time using the Python `time` module.

Python Time as a Tuple

Instead of using a number to represent Python time, you can use another primitive data structure: a [tuple](#).

The tuple allows you to manage time a little more easily by abstracting some of the data and making it more readable.

When you represent time as a tuple, each element in your tuple corresponds to a specific element of time:

1. Year
2. Month as an integer, ranging between 1 (January) and 12 (December)
3. Day of the month

4. Hour as an integer, ranging between 0 (12 A.M.) and 23 (11 P.M.)
5. Minute
6. Second
7. Day of the week as an integer, ranging between 0 (Monday) and 6 (Sunday)
8. Day of the year
9. Daylight savings time as an integer with the following values:
 - 1 is daylight savings time.
 - 0 is standard time.
 - -1 is unknown.

Using the methods you've already learned, you can represent the same Python time in two different ways:

```
Python >>>
>>> from time import time, ctime
>>> t = time()
>>> t
1551186415.360564
>>> ctime(t)
'Tue Feb 26 07:06:55 2019'

>>> time_tuple = (2019, 2, 26, 7, 6, 55, 1, 57, 0)
```

In this case, both `t` and `time_tuple` represent the same time, but the tuple provides a more readable interface for working with time components.

Technical Detail: Actually, if you look at the Python time represented by `time_tuple` in seconds (which you'll see how to do later in this article), you'll see that it resolves to `1551186415.0` rather than `1551186415.360564`.

This is because the tuple doesn't have a way to represent fractional seconds.

While the tuple provides a more manageable interface for working with Python time, there is an even better object: `struct_time`.

Python Time as an Object

The problem with the tuple construct is that it still looks like a bunch of numbers, even though it's better organized than a single, seconds-based number.

`struct_time` provides a solution to this by utilizing [NamedTuple](#), from Python's `collections` module, to associate the tuple's sequence of numbers with useful identifiers:

```
Python >>>
>>> from time import struct_time
>>> time_tuple = (2019, 2, 26, 7, 6, 55, 1, 57, 0)
>>> time_obj = struct_time(time_tuple)
>>> time_obj
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=26, tm_hour=7, tm_min=6, tm_sec=55, tm_wday=1, tm_y
```

Technical Detail: If you're coming from another language, the terms `struct` and `object` might be in opposition to one another.

In Python, there is no data type called `struct`. Instead, everything is an `object`.

However, the name `struct_time` is derived from the [C-based time library](#) where the data type is actually a `struct`.

In fact, Python's `time` module, which is [implemented in C](#), uses this `struct` directly by including the header file `times.h`.

Now, you can access specific elements of `time_obj` using the attribute's name rather than an index:

Python

>>>

```
>>> day_of_year = time_obj.tm_yday
>>> day_of_year
57
>>> day_of_month = time_obj.tm_mday
>>> day_of_month
26
```

Beyond the readability and usability of `struct_time`, it is also important to know because it is the return type of many of the functions in the Python `time` module.

Converting Python Time in Seconds to an Object

Now that you've seen the three primary ways of working with Python time, you'll learn how to convert between the different time data types.

Converting between time data types is dependent on whether the time is in UTC or local time.

Coordinated Universal Time (UTC)

The epoch uses UTC for its definition rather than a time zone. Therefore, the seconds elapsed since the epoch is not variable depending on your geographical location.

However, the same cannot be said of `struct_time`. The object representation of Python time may or may not take your time zone into account.

There are two ways to convert a float representing seconds to a `struct_time`:

1. UTC
2. Local time

To convert a Python time float to a UTC-based `struct_time`, the Python `time` module provides a function called `gmtime()`.

You've seen `gmtime()` used once before in this article:

Python

>>>

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3, tm_yda
```

You used this call to discover your system's epoch. Now, you have a better foundation for understanding what's actually happening here.

`gmtime()` converts the number of elapsed seconds since the epoch to a `struct_time` in UTC. In this case, you've passed `0` as the number of seconds, meaning you're trying to find the epoch, itself, in UTC.

Note: Notice the attribute `tm_isdst` is set to `0`. This attribute represents whether the time zone is using daylight savings time. UTC never subscribes to DST, so that flag will always be `0` when using `gmtime()`.

As you saw before, `struct_time` cannot represent fractional seconds, so `gmtime()` ignores the fractional seconds in the argument:

Python

>>>

```
>>> import time
>>> time.gmtime(1.99)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=1, tm_wday=3, tm_yda
```

Notice that even though the number of seconds you passed was very close to 2, the no fractional seconds were

Notice that even though the number of seconds you passed was very close to 2, the .99 fractional seconds were simply ignored, as shown by `tm_sec=1`.

The `secs` parameter for `gmtime()` is optional, meaning you can call `gmtime()` with no arguments. Doing so will provide the current time in UTC:

Python

>>>

```
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=28, tm_hour=12, tm_min=57, tm_sec=24, tm_wday=3, tm
```

Interestingly, there is no inverse for this function within `time`. Instead, you'll have to look in Python's `calendar` module for a function named `timegm()`:

Python

>>>

```
>>> import calendar
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2019, tm_mon=2, tm_mday=28, tm_hour=13, tm_min=23, tm_sec=12, tm_wday=3, tm
>>> calendar.timegm(time.gmtime())
1551360204
```

`timegm()` takes a tuple (or `struct_time`, since it is a subclass of `tuple`) and returns the corresponding number of seconds since the epoch.

Historical Context: If you're interested in why `timegm()` is not in `time`, you can view the discussion in [Python Issue 6280](#).

In short, it was originally added to `calendar` because `time` closely follows C's time library (defined in `time.h`), which contains no matching function. The above-mentioned issue proposed the idea of moving or copying `timegm()` into `time`.

However, with advances to the `datetime` library, inconsistencies in the patched implementation of `time.timegm()`, and a question of how to then handle `calendar.timegm()`, the maintainers declined the patch, encouraging the use of `datetime` instead.

Working with UTC is valuable in programming because it's a standard. You don't have to worry about DST, time zone, or locale information.

That said, there are plenty of cases when you'd want to use local time. Next, you'll see how to convert from seconds to local time so that you can do just that.

Local Time

In your application, you may need to work with local time rather than UTC. Python's `time` module provides a function for getting local time from the number of seconds elapsed since the epoch called `localtime()`.

The signature of `localtime()` is similar to `gmtime()` in that it takes an optional `secs` argument, which it uses to build a `struct_time` using your local time zone:

Python

>>>

```
>>> import time
>>> time.time()
1551448206.86196
>>> time.localtime(1551448206.86196)
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=7, tm_min=50, tm_sec=6, tm_wday=4, tm_yd
```

Notice that `tm_isdst=0`. Since DST matters with local time, `tm_isdst` will change between 0 and 1 depending on whether or not DST is applicable for the given time. Since `tm_isdst=0`, DST is not applicable for March 1, 2019.

In the United States in 2019, daylight savings time begins on March 10. So, to test if the DST flag will change correctly, you need to add 9 days' worth of seconds to the `secs` argument.

To compute this, you take the number of seconds in a day (86,400) and multiply that by 9 days:

Python

>>>

```
>>> new_secs = 1551448206.86196 + (86400 * 9)
>>> time.localtime(new_secs)
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=10, tm_hour=8, tm_min=50, tm_sec=6, tm_wday=6, tm_y
```

Now, you'll see that the `struct_time` shows the date to be March 10, 2019 with `tm_isdst=1`. Also, notice that `tm_hour` has also jumped ahead, to 8 instead of 7 in the previous example, because of daylight savings time.

Since Python 3.3, `struct_time` has also included two attributes that are useful in determining the time zone of the `struct_time`:

1. `tm_zone`
2. `tm_gmtoff`

At first, these attributes were platform dependent, but they have been available on all platforms since Python 3.6.

First, `tm_zone` stores the local time zone:

Python

>>>

```
>>> import time
>>> current_local = time.localtime()
>>> current_local.tm_zone
'CST'
```

Here, you can see that `localtime()` returns a `struct_time` with the time zone set to CST (Central Standard Time).

As you saw before, you can also tell the time zone based on two pieces of information, the UTC offset and DST (if applicable):

Python

>>>

```
>>> import time
>>> current_local = time.localtime()
>>> current_local.tm_gmtoff
-21600
>>> current_local.tm_isdst
0
```

In this case, you can see that `current_local` is 21600 seconds behind GMT, which stands for Greenwich Mean Time. GMT is the time zone with no UTC offset: $UTC \pm 00:00$.

21600 seconds divided by seconds per hour (3,600) means that `current_local` time is $GMT -06:00$ (or $UTC -06:00$).

You can use the GMT offset plus the DST status to deduce that `current_local` is $UTC -06:00$ at standard time, which corresponds to the Central standard time zone.

Like `gmtime()`, you can ignore the `secs` argument when calling `localtime()`, and it will return the current local time in a `struct_time`:

Python

>>>

```
>>> import time
>>> time.localtime()
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=8, tm_min=34, tm_sec=28, tm_wday=4, tm_y
```

Unlike `gmtime()`, the inverse function of `localtime()` does exist in the Python `time` module. Let's take a look at how that works.

Converting a Local Time Object to Seconds

You've already seen how to convert a UTC time object to seconds using `calendar.timegm()`. To convert local time to seconds, you'll use `mkttime()`.

`mkttime()` requires you to pass a parameter called `t` that takes the form of either a normal 9-tuple or a `struct_time` object representing local time:

```
Python >>>
>>> import time
>>> time_tuple = (2019, 3, 10, 8, 50, 6, 6, 69, 1)
>>> time.mktime(time_tuple)
1552225806.0

>>> time_struct = time.struct_time(time_tuple)
>>> time.mktime(time_struct)
1552225806.0
```

It's important to keep in mind that `t` must be a tuple representing local time, not UTC:

```
Python >>>
>>> from time import gmtime, mktime
>>> # 1
>>> current_utc = time.gmtime()
>>> current_utc
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=14, tm_min=51, tm_sec=19, tm_wday=4, tm_isdst=0)

>>> # 2
>>> current_utc_secs = mktime(current_utc)
>>> current_utc_secs
1551473479.0

>>> # 3
>>> time.gmtime(current_utc_secs)
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=20, tm_min=51, tm_sec=19, tm_wday=4, tm_isdst=0)
```

Note: For this example, assume that the local time is March 1, 2019 08:51:19 CST.

This example shows why it's important to use `mkttime()` with local time, rather than UTC:

1. `gmtime()` with no argument returns a `struct_time` using UTC. `current_utc` shows March 1, 2019 14:51:19 UTC. This is accurate because CST is UTC-06:00, so UTC should be 6 hours ahead of local time.
2. `mkttime()` tries to return the number of seconds, expecting local time, but you passed `current_utc` instead. So, instead of understanding that `current_utc` is UTC time, it assumes you meant March 1, 2019 14:51:19 CST.
3. `gmtime()` is then used to convert those seconds back into UTC, which results in an inconsistency. The time is now March 1, 2019 20:51:19 UTC. The reason for this discrepancy is the fact that `mkttime()` expected local time. So, the conversion back to UTC adds *another* 6 hours to local time.

Working with time zones is notoriously difficult, so it's important to set yourself up for success by understanding the differences between UTC and local time and the Python time functions that deal with each.

Converting a Python Time Object to a String

While working with tuples is fun and all, sometimes it's best to work with strings.

String representations of time, also known as timestamps, help make times more readable and can be especially useful for building intuitive user interfaces.

There are two Python `time` functions that you use for converting a `time.struct_time` object to a string:

1. `asctime()`
2. `strftime()`

You'll begin by learning about `asctime()`.

asctime()

You use `asctime()` for converting a time tuple or `struct_time` to a timestamp:

```
Python >>>
>>> import time
>>> time.asctime(time.gmtime())
'Fri Mar  1 18:42:08 2019'
>>> time.asctime(time.localtime())
'Fri Mar  1 12:42:15 2019'
```

Both `gmtime()` and `localtime()` return `struct_time` instances, for UTC and local time respectively.

You can use `asctime()` to convert either `struct_time` to a timestamp. `asctime()` works similarly to `ctime()`, which you learned about earlier in this article, except instead of passing a floating point number, you pass a tuple. Even the timestamp format is the same between the two functions.

As with `ctime()`, the parameter for `asctime()` is optional. If you do not pass a time object to `asctime()`, then it will use the current local time:

```
Python >>>
>>> import time
>>> time.asctime()
'Fri Mar  1 12:56:07 2019'
```

As with `ctime()`, it also ignores locale information.

One of the biggest drawbacks of `asctime()` is its format inflexibility. `strftime()` solves this problem by allowing you to format your timestamps.

strftime()

You may find yourself in a position where the string format from `ctime()` and `asctime()` isn't satisfactory for your application. Instead, you may want to format your strings in a way that's more meaningful to your users.

One example of this is if you would like to display your time in a string that takes locale information into account.

To format strings, given a `struct_time` or Python time tuple, you use `strftime()`, which stands for “string **format** time.”

`strftime()` takes two arguments:

1. `format` specifies the order and form of the time elements in your string.
2. `t` is an optional time tuple.

To format a string, you use **directives**. Directives are character sequences that begin with a % that specify a particular time element, such as:

- `%d`: Day of the month
- `%m`: Month of the year
- `%Y`: Year

For example, you can output the date in your local time using the [ISO 8601](#) standard like this:

Python

>>>

```
>>> import time  
>>> time.strftime('%Y-%m-%d', time.localtime())  
'2019-03-01'
```

Further Reading: While representing dates using Python time is completely valid and acceptable, you should also consider using Python's `datetime` module, which provides shortcuts and a more robust framework for working with dates and times together.

For example, you can simplify outputting a date in the ISO 8601 format using `datetime`:

Python

>>>

```
>>> from datetime import date  
>>> date(year=2019, month=3, day=1).isoformat()  
'2019-03-01'
```

To learn more about using the Python `datetime` module, check out [Using Python datetime to Work With Dates and Times](#)

As you saw before, a great benefit of using `strftime()` over `asctime()` is its ability to render timestamps that make use of locale-specific information.

For example, if you want to represent the date and time in a locale-sensitive way, you can't use `asctime()`:

Python

>>>

```
>>> from time import asctime  
>>> asctime()  
'Sat Mar  2 15:21:14 2019'  
  
>>> import locale  
>>> locale.setlocale(locale.LC_TIME, 'zh_HK') # Chinese - Hong Kong  
'zh_HK'  
>>> asctime()  
'Sat Mar  2 15:58:49 2019'
```

Notice that even after programmatically changing your locale, `asctime()` still returns the date and time in the same format as before.

Technical Detail: `LC_TIME` is the locale category for date and time formatting. The `locale` argument '`zh_HK`' may be different, depending on your system.

When you use `strftime()`, however, you'll see that it accounts for locale:

Python

>>>

```
>>> from time import strftime, localtime  
>>> strftime('%c', localtime())  
'Sat Mar  2 15:23:20 2019'  
  
>>> import locale  
>>> locale.setlocale(locale.LC_TIME, 'zh_HK') # Chinese - Hong Kong  
'zh_HK'  
>>> strftime('%c', localtime())  
'六 3/ 2 15:58:12 2019' 2019'
```

Here, you have successfully utilized the locale information because you used `strftime()`.

Note: `%c` is the directive for locale-appropriate date and time.

If the time tuple is not passed to the parameter `t`, then `strftime()` will use the result of `localtime()` by default. So, you could simplify the examples above by removing the optional second argument:

```
Python >>>
>>> from time import strftime
>>> strftime('The current local datetime is: %c')
'The current local datetime is: Fri Mar  1 23:18:32 2019'
```

Here, you've used the default time instead of passing your own as an argument. Also, notice that the `format` argument can consist of text other than formatting directives.

Further Reading: Check out this thorough [list of directives available to strftime\(\)](#).

The Python `time` module also includes the inverse operation of converting a timestamp back into a `struct_time` object.

Converting a Python Time String to an Object

When you're working with date and time related strings, it can be very valuable to convert the timestamp to a time object.

To convert a time string to a `struct_time`, you use `strptime()`, which stands for "string **p**arse **t**ime":

```
Python >>>
>>> from time import strptime
>>> strptime('2019-03-01', '%Y-%m-%d')
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=4, tm_yday=60, tm_isdst=-1)
```

The first argument to `strptime()` must be the timestamp you wish to convert. The second argument is the `format` that the timestamp is in.

The `format` parameter is optional and defaults to '`%a %b %d %H:%M:%S %Y`'. Therefore, if you have a timestamp in that format, you don't need to pass it as an argument:

```
Python >>>
>>> strptime('Fri Mar 01 23:38:40 2019')
time.struct_time(tm_year=2019, tm_mon=3, tm_mday=1, tm_hour=23, tm_min=38, tm_sec=40, tm_wday=4, tm_yday=60, tm_isdst=-1)
```

Since a `struct_time` has 9 key date and time components, `strptime()` must provide reasonable defaults for values for those components it can't parse from string.

In the previous examples, `tm_isdst=-1`. This means that `strptime()` can't determine by the timestamp whether it represents daylight savings time or not.

Now you know how to work with Python times and dates using the `time` module in a variety of ways. However, there are other uses for `time` outside of simply creating time objects, getting Python time strings, and using seconds elapsed since the epoch.

Suspending Execution

One really useful Python time function is `sleep()`, which suspends the thread's execution for a specified amount of time.

For example, you can suspend your program's execution for 10 seconds like this:

```
Python >>>
>>> from time import sleep, strftime
>>> strftime('%c')
'Fri Mar  1 23:49:26 2019'
>>> sleep(10)
>>> strftime('%c')
'Fri Mar  1 23:49:36 2019'
```

Your program will print the first formatted datetime string, then pause for 10 seconds, and finally print the second formatted datetime string.

You can also pass fractional seconds to `sleep()`:

Python

>>>

```
>>> from time import sleep  
>>> sleep(0.5)
```

`sleep()` is useful for testing or making your program wait for any reason, but you must be careful not to halt your production code unless you have good reason to do so.

Before Python 3.5, a signal sent to your process could interrupt `sleep()`. However, in 3.5 and later, `sleep()` will always suspend execution for at least the amount of specified time, even if the process receives a signal.

`sleep()` is just one Python time function that can help you test your programs and make them more robust.

Measuring Performance

You can use `time` to measure the performance of your program.

The way you do this is to use `perf_counter()` which, as the name suggests, provides a performance counter with a high resolution to measure short distances of time.

To use `perf_counter()`, you place a counter before your code begins execution as well as after your code's execution completes:

Python

>>>

```
>>> from time import perf_counter  
>>> def longrunning_function():  
...     for i in range(1, 11):  
...         time.sleep(i / i ** 2)  
...  
>>> start = perf_counter()  
>>> longrunning_function()  
>>> end = perf_counter()  
>>> execution_time = (end - start)  
>>> execution_time  
8.201258441999926
```

First, `start` captures the moment before you call the function. `end` captures the moment after the function returns. The function's total execution time took (`end - start`) seconds.

Technical Detail: Python 3.7 introduced `perf_counter_ns()`, which works the same as `perf_counter()`, but uses nanoseconds instead of seconds.

`perf_counter()` (or `perf_counter_ns()`) is the most precise way to measure the performance of your code using one execution. However, if you're trying to accurately gauge the performance of a code snippet, I recommend using the [Python `timeit` module](#).

`timeit` specializes in running code many times to get a more accurate performance analysis and helps you to avoid oversimplifying your time measurement as well as other common pitfalls.

Conclusion

Congratulations! You now have a great foundation for working with dates and times in Python.

Now, you're able to:

- Use a floating point number, representing seconds elapsed since the epoch, to deal with time
- Manage time using tuples and `struct_time` objects
- Convert between `seconds`, `tuples`, and `timestamp` strings

- Suspend the execution of a Python thread
- Measure performance using `perf_counter()`

On top of all that, you've learned some fundamental concepts surrounding date and time, such as:

- Epochs
- UTC
- Time zones
- Daylight savings time

Now, it's time for you to apply your newfound knowledge of Python time in your real world applications!

Further Reading

If you want to continue learning more about using dates and times in Python, take a look at the following modules:

- [datetime](#): A more robust date and time module in Python's standard library
- [timeit](#): A module for measuring the performance of code snippets
- [astropy](#): Higher precision datetimes used in astronomy

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Mastering Python's Built-in time Module](#)

About Alex Ronquillo

Alex Ronquillo is a Software Engineer at thelab. He's an avid Pythonista who is also passionate about writing and game development.

[» More about Alex](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

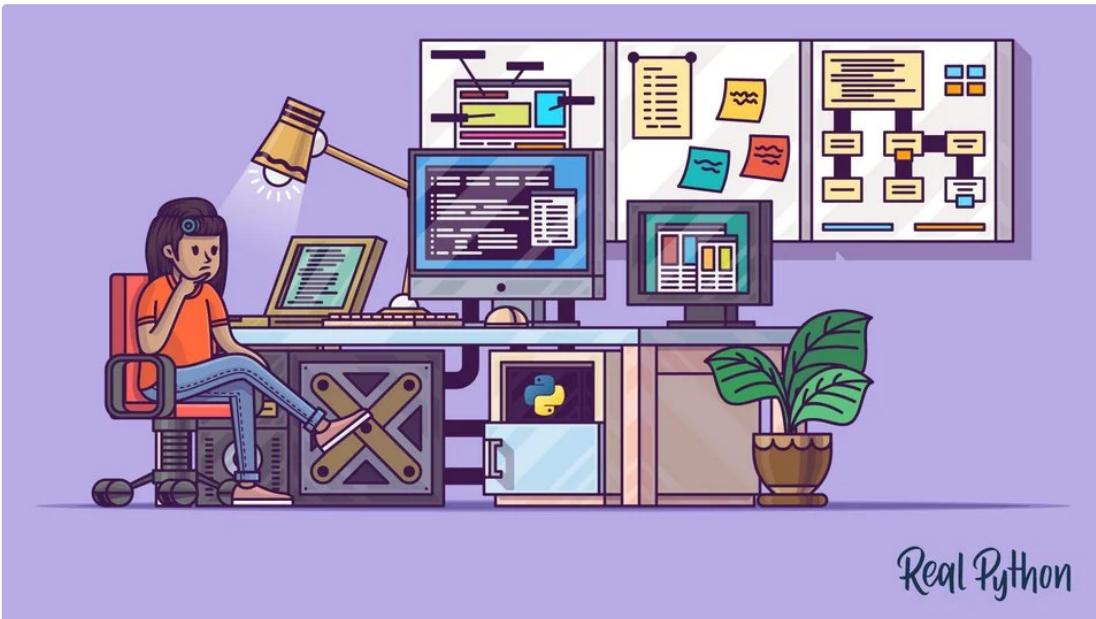
[Brad](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#)

Recommended Video Course: [Mastering Python's Built-in time Module](#)



Implementing an Interface in Python

by William Murphy · Feb 10, 2020 · 28 Comments · advanced · python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Python Interface Overview](#)
- [Informal Interfaces](#)
 - [Using Metaclasses](#)
 - [Using Virtual Base Classes](#)
- [Formal Interfaces](#)
 - [Using abc.ABCMeta](#)
 - [Using __subclasshook__\(\)](#)
 - [Using abc to Register a Virtual Subclass](#)
 - [Using Subclass Detection With Registration](#)
 - [Using Abstract Method Declaration](#)
- [Interfaces in Other Languages](#)
 - [Java](#)
 - [C++](#)
 - [Go](#)
- [Conclusion](#)

 **blackfire.io**
Profile & Optimize Python Apps Performance



Now available as
Public Beta
Sign-up for free and
install in minutes!

Interfaces play an important role in software engineering. As an application grows, updates and changes to the code base become more difficult to manage. More often than not, you wind up having classes that look very similar but are unrelated, which can lead to some confusion. In this tutorial, you'll see how you can use a **Python interface** to help determine what class you should use to tackle the current problem.

In this tutorial, you'll be able to:

- **Understand** how interfaces work and the caveats of Python interface creation
- **Comprehend** how useful interfaces are in a dynamic language like Python
- **Implement** an informal Python interface
- **Use** `abc.ABCMeta` and `@abc.abstractmethod` to implement a formal Python interface

Interfaces in Python are handled differently than in most other languages, and they can vary in their design complexity. By the end of this tutorial, you'll have a better understanding of some aspects of Python's data model, as well as how interfaces in Python compare to those in languages like Java, C++, and Go.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Python Interface Overview

At a high level, an interface acts as a **blueprint** for designing classes. Like classes, interfaces define methods. Unlike classes, these methods are abstract. An **abstract method** is one that the interface simply defines. It doesn't implement the methods. This is done by classes, which then **implement** the interface and give concrete meaning to the interface's abstract methods.

Python's approach to interface design is somewhat different when compared to languages like [Java](#), [Go](#), and [C++](#). These languages all have an `interface` keyword, while Python does not. Python further deviates from other languages in one other aspect. It doesn't require the class that's implementing the interface to define all of the interface's abstract methods.

Informal Interfaces

In certain circumstances, you may not need the strict rules of a formal Python interface. Python's dynamic nature allows you to implement an **informal interface**. An informal Python interface is a class that defines methods that can be overridden, but there's no strict enforcement.

In the following example, you'll take the perspective of a data engineer who needs to extract text from various different unstructured file types, like PDFs and emails. You'll create an informal interface that defines the methods that will be in both the `PdfParser` and `EmailParser` concrete classes:

Python

```
class InformalParserInterface:
    def load_data_source(self, path: str, file_name: str) -> str:
        """Load in the file for extracting text."""
        pass

    def extract_text(self, full_file_name: str) -> dict:
        """Extract text from the currently loaded file."""
        pass
```

`InformalParserInterface` defines the two methods `.load_data_source()` and `.extract_text()`. These methods are defined but not implemented. The implementation will occur once you create **concrete classes** that inherit from `InformalParserInterface`.

As you can see, `InformalParserInterface` looks identical to a standard Python class. You rely on [duck typing](#) to inform users that this is an interface and should be used accordingly.

Note: Haven't heard of **duck typing**? This term says that if you have an object that looks like a duck, walks like a duck, and quacks like a duck, then it must be a duck! To learn more, check out [Duck Typing](#).

With duck typing in mind, you define two classes that implement the `InformalParserInterface`. To use your interface, you must create a concrete class. A **concrete class** is a subclass of the interface that provides an implementation of the interface's methods. You'll create two concrete classes to implement your interface. The first is `PdfParser`, which you'll use to parse the text from [PDF](#) files:

Python

```
class PdfParser(InformalParserInterface):
    """Extract text from a PDF"""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides InformalParserInterface.load_data_source()"""
        pass
```

```
def extract_text(self, full_file_path: str) -> dict:  
    """Overrides InformalParserInterface.extract_text()"""  
    pass
```

The concrete implementation of `InformalParserInterface` now allows you to extract text from [PDF](#) files.

The second concrete class is `EmlParser`, which you'll use to parse the text from emails:

Python

```
class EmlParser(InformalParserInterface):  
    """Extract text from an email"""\n    def load_data_source(self, path: str, file_name: str) -> str:  
        """Overrides InformalParserInterface.load_data_source()"""  
        pass  
  
    def extract_text_from_email(self, full_file_path: str) -> dict:  
        """A method defined only in EmlParser.  
        Does not override InformalParserInterface.extract_text()  
        """  
        pass
```

The concrete implementation of `InformalParserInterface` now allows you to extract text from email files.

So far, you've defined two **concrete implementations** of the `InformalPythonInterface`. However, note that `EmlParser` fails to properly define `.extract_text()`. If you were to check whether `EmlParser` implements `InformalParserInterface`, then you'd get the following result:

Python

>>>

```
>>> # Check if both PdfParser and EmlParser implement InformalParserInterface  
>>> issubclass(PdfParser, InformalParserInterface)  
True  
  
>>> issubclass(EmlParser, InformalParserInterface)  
True
```

This would return `True`, which poses a bit of a problem since it violates the definition of an interface!

Now check the **method resolution order (MRO)** of `PdfParser` and `EmlParser`. This tells you the superclasses of the class in question, as well as the order in which they're searched for executing a method. You can view a class's MRO by using the dunder method `cls.__mro__`:

Python

>>>

```
>>> PdfParser.__mro__  
(__main__.PdfParser, __main__.InformalParserInterface, object)  
  
>>> EmlParser.__mro__  
(__main__.EmlParser, __main__.InformalParserInterface, object)
```

Such informal interfaces are fine for small projects where only a few developers are working on the source code. However, as projects get larger and teams grow, this could lead to developers spending countless hours looking for hard-to-find logic errors in the codebase!

Using Metaclasses

Ideally, you would want `issubclass(EmlParser, InformalParserInterface)` to return `False` when the implementing class doesn't define all of the interface's abstract methods. To do this, you'll create a [metaclass](#) called `ParserMeta`. You'll be overriding two [dunder](#) methods:

1. `__instancecheck__()`
2. `__subclasscheck__()`

In the code block below, you create a class called `UpdatedInformalParserInterface` that builds from the `ParserMeta` metaclass:

Python

```
class ParserMeta(type):
    """A Parser metaclass that will be used for parser class creation.
    """
    def __instancecheck__(cls, instance):
        return cls.__subclasscheck__(type(instance))

    def __subclasscheck__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text))

class UpdatedInformalParserInterface(metaclass=ParserMeta):
    """This interface is used for concrete classes to inherit from.
    There is no need to define the ParserMeta methods as any class
    as they are implicitly made available via .__subclasscheck__().
    """
    pass
```

Now that `ParserMeta` and `UpdatedInformalParserInterface` have been created, you can create your concrete implementations.

First, create a new class for parsing PDFs called `PdfParserNew`:

Python

```
class PdfParserNew:
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides UpdatedInformalParserInterface.load_data_source()"""
        pass

    def extract_text(self, full_file_path: str) -> dict:
        """Overrides UpdatedInformalParserInterface.extract_text()"""
        pass
```

Here, `PdfParserNew` overrides `.load_data_source()` and `.extract_text()`, so `issubclass(PdfParserNew, UpdatedInformalParserInterface)` should return `True`.

In this next code block, you have a new implementation of the email parser called `EmlParserNew`:

Python

```
class EmlParserNew:
    """Extract text from an email."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides UpdatedInformalParserInterface.load_data_source()"""
        pass

    def extract_text_from_email(self, full_file_path: str) -> dict:
        """A method defined only in EmlParser.
        Does not override UpdatedInformalParserInterface.extract_text()
        """
        pass
```

Here, you have a metaclass that's used to create `UpdatedInformalParserInterface`. By using a metaclass, you don't need to explicitly define the subclasses. Instead, the subclass must **define the required methods**. If it doesn't, then `issubclass(EmlParserNew, UpdatedInformalParserInterface)` will return `False`.

Running `issubclass()` on your concrete classes will produce the following:

Python

>>>

```
>>> issubclass(PdfParserNew, UpdatedInformalParserInterface)
True

>>> issubclass(EmlParserNew, UpdatedInformalParserInterface)
False
```

As expected, `EmlParserNew` is not a subclass of `UpdatedInformalParserInterface` since `.extract_text()` wasn't defined in `EmlParserNew`.

Now, let's have a look at the MRO:

```
Python >>> PdfParserNew.__mro__
(<class '__main__.PdfParserNew'>, <class 'object'>)
```

As you can see, `UpdatedInformalParserInterface` is a superclass of `PdfParserNew`, but it doesn't appear in the MRO. This unusual behavior is caused by the fact that `UpdatedInformalParserInterface` is a **virtual base class** of `PdfParserNew`.

Using Virtual Base Classes

In the previous example, `issubclass(EmlParserNew, UpdatedInformalParserInterface)` returned `True`, even though `UpdatedInformalParserInterface` did not appear in the `EmlParserNew` MRO. That's because `UpdatedInformalParserInterface` is a **virtual base class** of `EmlParserNew`.

The key difference between these and standard subclasses is that virtual base classes use the `__subclasscheck__()` dunder method to implicitly check if a class is a virtual subclass of the superclass. Additionally, virtual base classes don't appear in the subclass MRO.

Take a look at this code block:

```
Python
class PersonMeta(type):
    """A person metaclass"""
    def __instancecheck__(cls, instance):
        return cls.__subclasscheck__(type(instance))

    def __subclasscheck__(cls, subclass):
        return (hasattr(subclass, 'name') and
                callable(subclass.name) and
                hasattr(subclass, 'age') and
                callable(subclass.age))

class PersonSuper:
    """A person superclass"""
    def name(self) -> str:
        pass

    def age(self) -> int:
        pass

class Person(metaclass=PersonMeta):
    """Person interface built from PersonMeta metaclass."""
    pass
```

Here, you have the setup for creating your virtual base classes:

1. The metaclass `PersonMeta`
2. The base class `PersonSuper`
3. The Python interface `Person`

Now that the setup for creating **virtual base classes** is done you'll define two concrete classes, `Employee` and `Friend`. The `Employee` class inherits from `PersonSuper`, while `Friend` implicitly inherits from `Person`:

```
Python
# Inheriting subclasses
class Employee(PersonSuper):
    """Inherits from PersonSuper
    PersonSuper will appear in Employee.__mro__
    """
    pass

class Friend:
```

```

"""Built implicitly from Person
Friend is a virtual subclass of Person since
both required methods exist.
Person not in Friend.__mro__
"""

def name(self):
    pass

def age(self):
    pass

```

Although `Friend` does not explicitly inherit from `Person`, it implements `.name()` and `.age()`, so `Person` becomes a **virtual base class** of `Friend`. When you run `issubclass(Friend, Person)` it should return `True`, meaning that `Friend` is a subclass of `Person`.

The following [UML](#) diagram shows what happens when you call `issubclass()` on the `Friend` class:

Taking a look at `PersonMeta`, you'll notice that there's another dunder method called `__instancecheck__()`. This method is used to check if instances of `Friend` are created from the `Person` interface. Your code will call `__instancecheck__()` when you use `isinstance(Friend, Person)`.

Formal Interfaces

Informal interfaces can be useful for projects with a small code base and a limited number of programmers. However, informal interfaces would be the wrong approach for larger applications. In order to create a **formal Python interface**, you'll need a few more tools from Python's `abc` module.

Using abc . ABCMeta

To enforce the subclass instantiation of abstract methods, you'll utilize Python's builtin `ABCMeta` from the [abc](#) module. Going back to your `UpdatedInformalParserInterface` interface, you created your own metaclass, `ParserMeta`, with the overridden dunder methods `__instancecheck__()` and `__subclasscheck__()`.

Rather than create your own metaclass, you'll use `abc.ABCMeta` as the metaclass. Then, you'll overwrite `__subclasshook__()` in place of `__instancecheck__()` and `__subclasscheck__()`, as it creates a more reliable implementation of these dunder methods.

Using `__subclasshook__()`

Here's the implementation of `FormalParserInterface` using `abc.ABCMeta` as your metaclass:

Python

```

import abc

class FormalParserInterface(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text))

class PdfParserNew:
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:

```

```

    """Overrides FormalParserInterface.load_data_source()
    pass

    def extract_text(self, full_file_path: str) -> dict:
        """Overrides FormalParserInterface.extract_text()
        pass

    class EmlParserNew:
        """Extract text from an email."""
        def load_data_source(self, path: str, file_name: str) -> str:
            """Overrides FormalParserInterface.load_data_source()
            pass

        def extract_text_from_email(self, full_file_path: str) -> dict:
            """A method defined only in EmlParser.
            Does not override FormalParserInterface.extract_text()
            """
            pass

```

If you run `issubclass()` on `PdfParserNew` and `EmlParserNew`, then `issubclass()` will return `True` and `False`, respectively.

Using abc to Register a Virtual Subclass

Once you've imported the `abc` module, you can directly **register a virtual subclass** by using the `.register()` metamethod. In the next example, you register the interface `Double` as a virtual base class of the built-in `__float__` class:

Python

```

class Double(metaclass=abc.ABCMeta):
    """Double precision floating point number."""
    pass

Double.register(float)

```

You can check out the effect of using `.register()`:

Python

```

>>> issubclass(float, Double)
True

>>> isinstance(1.2345, Double)
True

```

>>>

By using the `.register()` meta method, you've successfully registered `Double` as a virtual subclass of `float`.

Once you've registered `Double`, you can use it as class [decorator](#) to set the decorated class as a virtual subclass:

Python

```

@Double.register
class Double64:
    """A 64-bit double-precision floating-point number."""
    pass

print(issubclass(Double64, Double)) # True

```

The decorator `register` method helps you to create a hierarchy of custom virtual class inheritance.

Using Subclass Detection With Registration

You must be careful when you're combining `__subklasshook__()` with `.register()`, as `__subklasshook__()`

takes precedence over virtual subclass registration. To ensure that the registered virtual subclasses are taken into consideration, you must add `NotImplemented` to the `__subclasshook__()` dunder method. The `FormalParserInterface` would be updated to the following:

Python

```
class FormalParserInterface(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text) or
                NotImplemented)

class PdfParserNew:
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass

    def extract_text(self, full_file_path: str) -> dict:
        """Overrides FormalParserInterface.extract_text()"""
        pass

    @FormalParserInterface.register
    class EmlParserNew:
        """Extract text from an email."""
        def load_data_source(self, path: str, file_name: str) -> str:
            """Overrides FormalParserInterface.load_data_source()"""
            pass

        def extract_text_from_email(self, full_file_path: str) -> dict:
            """A method defined only in EmlParser.
            Does not override FormalParserInterface.extract_text()
            """
            pass

    print(issubclass(PdfParserNew, FormalParserInterface)) # True
    print(issubclass(EmlParserNew, FormalParserInterface)) # True
```

Since you've used registration, you can see that `EmlParserNew` is considered a virtual subclass of your `FormalParserInterface` interface. This is not what you wanted since `EmlParserNew` doesn't override `.extract_text()`. **Please use caution with virtual subclass registration!**

Using Abstract Method Declaration

An **abstract method** is a method that's declared by the Python interface, but it may not have a useful implementation. The abstract method must be overridden by the concrete class that implements the interface in question.

To create abstract methods in Python, you add the `@abc.abstractmethod` decorator to the interface's methods. In the next example, you update the `FormalParserInterface` to include the abstract methods `.load_data_source()` and `.extract_text()`:

Python

```
class FormalParserInterface(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text) or
                NotImplemented)

    @abc.abstractmethod
    def load_data_source(self, path: str, file_name: str):
        """Load in the data set"""
        raise NotImplementedError

    @abc.abstractmethod
```

```

def extract_text(self, full_file_path: str):
    """Extract text from the data set"""
    raise NotImplementedError

class PdfParserNew(FormalParserInterface):
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass

    def extract_text(self, full_file_path: str) -> dict:
        """Overrides FormalParserInterface.extract_text()"""
        pass

class EmlParserNew(FormalParserInterface):
    """Extract text from an email."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass

    def extract_text_from_email(self, full_file_path: str) -> dict:
        """A method defined only in EmlParser.
        Does not override FormalParserInterface.extract_text()
        """
        pass

```

In the above example, you've finally created a formal interface that will raise errors when the abstract methods aren't overridden. The `PdfParserNew` instance, `pdf_parser`, won't raise any errors, as `PdfParserNew` is correctly overriding the `FormalParserInterface` abstract methods. However, `EmlParserNew` will raise an error:

```

Python >>>
>>> pdf_parser = PdfParserNew()
>>> eml_parser = EmlParserNew()
Traceback (most recent call last):
  File "real_python_interfaces.py", line 53, in <module>
    eml_interface = EmlParserNew()
TypeError: Can't instantiate abstract class EmlParserNew with abstract methods extract_text

```

As you can see, the `traceback` message tells you that you haven't overridden all the abstract methods. This is the behavior you expect when building a formal Python interface.

Interfaces in Other Languages

Interfaces appear in many programming languages, and their implementation varies greatly from language to language. In the next few sections, you'll compare interfaces in Python to Java, C++, and Go.

Java

Unlike Python, `Java` contains an `interface` keyword. Keeping with the file parser example, you declare an interface in Java like so:

```

Java
public interface FileParserInterface {
    // Static fields, and abstract methods go here ...
    public void loadDataSource();
    public void extractText();
}

```

Now you'll create two concrete classes, `PdfParser` and `EmlParser`, to implement the `FileParserInterface`. To do so, you must use the `implements` keyword in the class definition, like so:

Java

```
public class EmlParser implements FileParserInterface {  
    public void loadDataSource() {  
        // Code to load the data set  
    }  
  
    public void extractText() {  
        // Code to extract the text  
    }  
}
```

Continuing with your file parsing example, a fully-functional Java interface would look something like this:

Java

```
import java.util.*;  
import java.io.*;  
  
public class FileParser {  
    public static void main(String[] args) throws IOException {  
        // The main entry point  
    }  
  
    public interface FileParserInterface {  
        HashMap<String, ArrayList<String>> file_contents = null;  
  
        public void loadDataSource();  
        public void extractText();  
    }  
  
    public class PdfParser implements FileParserInterface {  
        public void loadDataSource() {  
            // Code to load the data set  
        }  
  
        public void extractText() {  
            // Code to extract the text  
        }  
    }  
  
    public class EmlParser implements FileParserInterface {  
        public void loadDataSource() {  
            // Code to load the data set  
        }  
  
        public void extractText() {  
            // Code to extract the text  
        }  
    }  
}
```

As you can see, a Python interface gives you much more flexibility during creation than a Java interface does.

C++

Like Python, C++ uses abstract base classes to create interfaces. When defining an interface in C++, you use the keyword `virtual` to describe a method that should be overwritten in the concrete class:

C++

```
class FileParserInterface {  
  
public:  
    virtual void loadDataSource(std::string path, std::string file_name);  
    virtual void extractText(std::string full_file_name);  
};
```

When you want to implement the interface, you'll give the concrete class name, followed by a colon (:), and then the name of the interface. The following example demonstrates C++ interface implementation:

C++

```
class PdfParser : FileParserInterface {  
public:  
    void loadDataSource(std::string path, std::string file_name);  
    void extractText(std::string full_file_name);  
};  
  
class EmlParser : FileParserInterface {  
public:  
    void loadDataSource(std::string path, std::string file_name);  
    void extractText(std::string full_file_name);  
};
```

A Python interface and a C++ interface have some similarities in that they both make use of abstract base classes to simulate interfaces.

Go

Although Go's syntax is reminiscent of Python, the Go programming language contains an `interface` keyword, like Java. Let's create the `fileParserInterface` in Go:

Go

```
type fileParserInterface interface {  
    loadDataSet(path string, filename string)  
    extractText(full_file_path string)  
}
```

A big difference between Python and Go is that Go doesn't have classes. Rather, Go is similar to C in that it uses the `struct` keyword to create structures. A **structure** is similar to a class in that a structure contains data and methods. However, unlike a class, all of the data and methods are publicly accessed. The concrete structs in Go will be used to implement the `fileParserInterface`.

Here's an example of how Go uses interfaces:

Go

```
package main

type fileParserInterface interface {
    loadDataSet(path string, filename string)
    extractText(full_file_path string)
}

type pdfParser struct {
    // Data goes here ...
}

type emlParser struct {
    // Data goes here ...
}

func (p pdfParser) loadDataSet() {
    // Method definition ...
}

func (p pdfParser) extractText() {
    // Method definition ...
}

func (e emlParser) loadDataSet() {
    // Method definition ...
}

func (e emlParser) extractText() {
    // Method definition ...
}

func main() {
    // Main entrypoint
}
```

Unlike a Python interface, a Go interface is created using structs and the `interface` keyword.

Conclusion

Python offers great flexibility when you're creating interfaces. An informal Python interface is useful for small projects where you're less likely to get confused as to what the return types of the methods are. As a project grows, the need for a **formal Python interface** becomes more important as it becomes more difficult to infer return types. This ensures that the concrete class, which implements the interface, overwrites the abstract methods.

Now you can:

- Understand **how interfaces work** and the caveats of creating a Python interface
- Understand the **usefulness** of interfaces in a dynamic language like Python
- Implement **formal and informal** interfaces in Python
- **Compare Python interfaces** to those in languages like Java, C++, and Go

Now that you've become familiar with how to create a Python interface, add a Python interface to your next project to see its usefulness in action!

About William Murphy

William has been working with Python for over 6 years, working in roles such as data scientist, machine learning engineer, data engineer, and dev ops engineer. He is currently a senior software engineering consultant at ModernDay Productions.

[» More about William](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

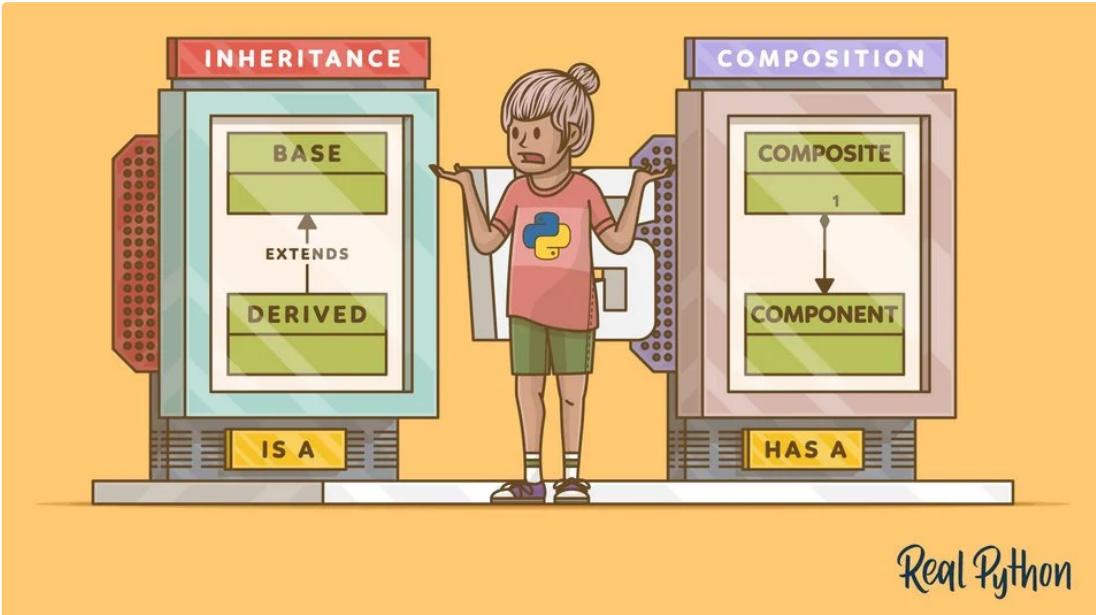
[Jaya](#)

[Joanna](#)

[Mike](#)

Keep Learning

Related Tutorial Categories: [advanced](#) [python](#)



Real Python

Inheritance and Composition: A Python OOP Guide

by Isaac Rodriguez 19 Comments best-practices intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Are Inheritance and Composition?](#)
 - [What's Inheritance?](#)
 - [What's Composition?](#)
- [An Overview of Inheritance in Python](#)
 - [The Object Super Class](#)
 - [Exceptions Are an Exception](#)
 - [Creating Class Hierarchies](#)
 - [Abstract Base Classes in Python](#)
 - [Implementation Inheritance vs Interface Inheritance](#)
 - [The Class Explosion Problem](#)
 - [Inheriting Multiple Classes](#)
- [Composition in Python](#)
 - [Flexible Designs With Composition](#)
 - [Customizing Behavior With Composition](#)
- [Choosing Between Inheritance and Composition in Python](#)
 - [Inheritance to Model “Is A” Relationship](#)
 - [Mixing Features With Mixin Classes](#)
 - [Composition to Model “Has A” Relationship](#)
 - [Composition to Change Run-Time Behavior](#)
 - [Choosing Between Inheritance and Composition in Python](#)
- [Conclusion](#)
- [Recommended Reading](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Inheritance and Composition: A Python OOP Guide](#)

In this article, you'll explore **inheritance** and **composition** in Python. [Inheritance](#) and [composition](#) are two important concepts in object oriented programming that model the relationship between two classes. They are the building blocks of [object oriented design](#), and they help programmers to write reusable code.

By the end of this article, you'll know how to:

- Use inheritance in Python
- Model class hierarchies using inheritance
- Use multiple inheritance in Python and understand its drawbacks
- Use composition to create complex objects
- Reuse existing code by applying composition
- Change application behavior at run-time through composition

Free Bonus: [Click here to get access to a free Python OOP Cheat Sheet](#) that points you to the best tutorials, videos, and books to learn more about Object-Oriented Programming with Python.

What Are Inheritance and Composition?

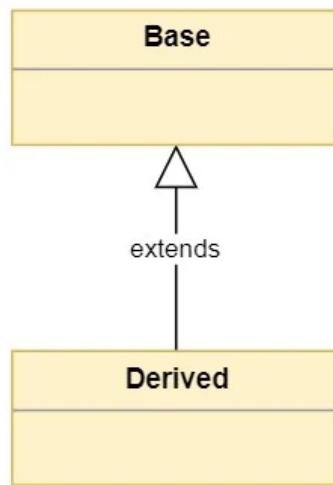
Inheritance and **composition** are two major concepts in object oriented programming that model the relationship between two classes. They drive the design of an application and determine how the application should evolve as new features are added or requirements change.

Both of them enable code reuse, but they do it in different ways.

What's Inheritance?

Inheritance models what is called an **is a** relationship. This means that when you have a Derived class that inherits from a Base class, you created a relationship where **Derived is a specialized version of Base**.

Inheritance is represented using the [Unified Modeling Language](#) or UML in the following way:



Classes are represented as boxes with the class name on top. The inheritance relationship is represented by an arrow from the derived class pointing to the base class. The word **extends** is usually added to the arrow.

Note: In an inheritance relationship:

- Classes that inherit from another are called derived classes, subclasses, or subtypes.
- Classes from which other classes are derived are called base classes or super classes.
- A derived class is said to derive, inherit, or extend a base class.

Let's say you have a base class `Animal` and you derive from it to create a `Horse` class. The inheritance relationship states that a `Horse` **is an** `Animal`. This means that `Horse` inherits the [interface](#) and implementation of `Animal`, and `Horse` objects can be used to replace `Animal` objects in the application.

This is known as the [Liskov substitution principle](#). The principle states that “in a computer program, if `s` is a subtype of `T`, then objects of type `T` may be replaced with objects of type `s` without altering any of the desired properties of the program”.

You'll see in this article why you should always follow the Liskov substitution principle when creating your class hierarchies, and the problems you'll run into if you don't.

What's Composition?

Composition is a concept that models a **has a** relationship. It enables creating complex types by combining objects of other types. This means that a class `Composite` can contain an object of another class `Component`. This relationship means that a `Composite` **has a** `Component`.

UML represents composition as follows:



Composition is represented through a line with a diamond at the composite class pointing to the component class. The composite side can express the cardinality of the relationship. The cardinality indicates the number or valid range of Component instances the Composite class will contain.

In the diagram above, the `1` represents that the Composite class contains one object of type Component. Cardinality can be expressed in the following ways:

- **A number** indicates the number of Component instances that are contained in the Composite.
- **The `*` symbol** indicates that the Composite class can contain a variable number of Component instances.
- **A range `1..4`** indicates that the Composite class can contain a range of Component instances. The range is indicated with the minimum and maximum number of instances, or minimum and many instances like in `1..*`.

Note: Classes that contain objects of other classes are usually referred to as composites, where classes that are used to create more complex types are referred to as components.

For example, your `Horse` class can be composed by another object of type `Tail`. Composition allows you to express that relationship by saying a `Horse` **has a** `Tail`.

Composition enables you to reuse code by adding objects to other objects, as opposed to inheriting the interface and implementation of other classes. Both `Horse` and `Dog` classes can leverage the functionality of `Tail` through composition without deriving one class from the other.

An Overview of Inheritance in Python

Everything in Python is an object. Modules are objects, class definitions and functions are objects, and of course, objects created from classes are objects too.

Inheritance is a required feature of every object oriented programming language. This means that Python supports inheritance, and as you'll see later, it's one of the few languages that supports multiple inheritance.

When you write Python code using classes, you are using inheritance even if you don't know you're using it. Let's take a look at what that means.

The Object Super Class

The easiest way to see inheritance in Python is to jump into the [Python interactive shell](#) and write a little bit of code. You'll start by writing the simplest class possible:

```
Python >>>
>>> class MyClass:
...     pass
...
```

You declared a class `MyClass` that doesn't do much, but it will illustrate the most basic inheritance concepts. Now that you have the class declared, you can use the `dir()` function to list its members:

```
Python >>>
>>> c = MyClass()
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__']
```

`dir()` returns a list of all the members in the specified object. You have not declared any members in `MyClass`, so where is the list coming from? You can find out using the interactive interpreter:

```
Python >>>
>>> o = object()
>>> dir(o)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__']
```

As you can see, the two lists are nearly identical. There are some additional members in `MyClass` like `__dict__` and `__weakref__`, but every single member of the `object` class is also present in `MyClass`.

This is because every class you create in Python implicitly derives from `object`. You could be more explicit and write `class MyClass(object):`, but it's redundant and unnecessary.

Note: In Python 2, you have to explicitly derive from `object` for reasons beyond the scope of this article, but you can read about it in the [New-style and classic classes](#) section of the Python 2 documentation.

Exceptions Are an Exception

Every class that you create in Python will implicitly derive from `object`. The exception to this rule are classes used to indicate errors by raising an [exception](#).

You can see the problem using the Python interactive interpreter:

Python

>>>

```
>>> class MyError:  
...     pass  
...  
>>> raise MyError()  
  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: exceptions must derive from BaseException
```

You created a new class to indicate a type of error. Then you tried to use it to raise an exception. An exception is raised but the output states that the exception is of type `TypeError` not `MyError` and that all exceptions must derive from `BaseException`.

`BaseException` is a base class provided for all error types. To create a new error type, you must derive your class from `BaseException` or one of its derived classes. The convention in Python is to derive your custom error types from `Exception`, which in turn derives from `BaseException`.

The correct way to define your error type is the following:

Python

>>>

```
>>> class MyError(Exception):  
...     pass  
...  
>>> raise MyError()  
  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
__main__.MyError
```

As you can see, when you raise `MyError`, the output correctly states the type of error raised.

Creating Class Hierarchies

Inheritance is the mechanism you'll use to create hierarchies of related classes. These related classes will share a common interface that will be defined in the base classes. Derived classes can specialize the interface by providing a particular implementation where applies.

In this section, you'll start modeling an HR system. The example will demonstrate the use of inheritance and how derived classes can provide a concrete implementation of the base class interface.

The HR system needs to process payroll for the company's employees, but there are different types of employees depending on how their payroll is calculated.

You start by implementing a `PayrollSystem` class that processes payroll:

Python

```
# In hr.py  
  
class PayrollSystem:  
    def calculate_payroll(self, employees):  
        print('Calculating Payroll')  
        print('=====')  
        for employee in employees:  
            print(f'Payroll for: {employee.id} - {employee.name}')  
            print(f'- Check amount: {employee.calculate_payroll()}')  
            print()
```

The `PayrollSystem` implements a `.calculate_payroll()` method that takes a collection of employees and prints their `id`, `name`, and check amount using the `.calculate_payroll()` method exposed on each `employee` object.

Now, you implement a base class `Employee` that handles the common interface for every employee type:

Python

```
# In hr.py

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name
```

Employee is the base class for all employee types. It is constructed with an `id` and a `name`. What you are saying is that every Employee must have an `id` assigned as well as a `name`.

The HR system requires that every Employee processed must provide a `.calculate_payroll()` interface that returns the weekly salary for the employee. The implementation of that interface differs depending on the type of Employee.

For example, administrative workers have a fixed salary, so every week they get paid the same amount:

Python

```
# In hr.py

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary
```

You create a derived class `SalaryEmployee` that inherits `Employee`. The class is initialized with the `id` and `name` required by the base class, and you use `super()` to initialize the members of the base class. You can read all about `super()` in [Supercharge Your Classes With Python `super\(\)`](#).

`SalaryEmployee` also requires a `weekly_salary` initialization parameter that represents the amount the employee makes per week.

The class provides the required `.calculate_payroll()` method used by the HR system. The implementation just returns the amount stored in `weekly_salary`.

The company also employs manufacturing workers that are paid by the hour, so you add an `HourlyEmployee` to the HR system:

Python

```
# In hr.py

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate
```

The `HourlyEmployee` class is initialized with `id` and `name`, like the base class, plus the `hours_worked` and the `hour_rate` required to calculate the payroll. The `.calculate_payroll()` method is implemented by returning the hours worked times the hour rate.

Finally, the company employs sales associates that are paid through a fixed salary plus a commission based on their sales, so you create a `CommissionEmployee` class:

Python

```
# In hr.py

class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

You derive `CommissionEmployee` from `SalaryEmployee` because both classes have a `weekly_salary` to consider. At the same time, `CommissionEmployee` is initialized with a `commission` value that is based on the sales for the employee.

`.calculate_payroll()` leverages the implementation of the base class to retrieve the `fixed` salary and adds the `commission` value.

Since `CommissionEmployee` derives from `SalaryEmployee`, you have access to the `weekly_salary` property directly, and you could've implemented `.calculate_payroll()` using the value of that property.

The problem with accessing the property directly is that if the implementation of `SalaryEmployee.calculate_payroll()` changes, then you'll have to also change the implementation of `CommissionEmployee.calculate_payroll()`. It's better to rely on the already implemented method in the base class and extend the functionality as needed.

You created your first class hierarchy for the system. The UML diagram of the classes looks like this:

```
classDiagram
    class PayrollSystem {
        <<IPayrollCalculator>>
    }
    class IPayrollCalculator {
        <<IPayrollCalculator>>
    }
    class SalaryEmployee {
        <<IPayrollCalculator>>
    }
    class CommissionEmployee {
        <<IPayrollCalculator>>
    }
    PayrollSystem <|-- IPayrollCalculator
    IPayrollCalculator <|-- SalaryEmployee
    IPayrollCalculator <|-- CommissionEmployee
```

The diagram shows the inheritance hierarchy of the classes. The derived classes implement the `IPayrollCalculator` interface, which is required by the `PayrollSystem`. The `PayrollSystem.calculate_payroll()` implementation requires that the `employee` objects passed contain an `id`, `name`, and `calculate_payroll()` implementation.

Interfaces are represented similarly to classes with the word **interface** above the interface name. Interface names are usually prefixed with a capital I.

The application creates its employees and passes them to the payroll system to process payroll:

Python

```
# In program.py

import hr

salary_employee = hr.SalaryEmployee(1, 'John Smith', 1500)
hourly_employee = hr.HourlyEmployee(2, 'Jane Doe', 40, 15)
commission_employee = hr.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee
])
```

You can run the program in the command line and see the results:

Shell

```
$ python program.py

Calculating Payroll
=====
Payroll for: 1 - John Smith
- Check amount: 1500

Payroll for: 2 - Jane Doe
- Check amount: 600

Payroll for: 3 - Kevin Bacon
- Check amount: 1250
```

The program creates three employee objects, one for each of the derived classes. Then, it creates the payroll system and passes a list of the employees to its `.calculate_payroll()` method, which calculates the payroll for each employee and prints the results.

Notice how the `Employee` base class doesn't define a `.calculate_payroll()` method. This means that if you were to create a plain `Employee` object and pass it to the `PayrollSystem`, then you'd get an error. You can try it in the Python interactive interpreter:

Python

>>>

```
>>> import hr
>>> employee = hr.Employee(1, 'Invalid')
>>> payroll_system = hr.PayrollSystem()
>>> payroll_system.calculate_payroll([employee])

Payroll for: 1 - Invalid
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/hr.py", line 39, in calculate_payroll
      print(f'- Check amount: {employee.calculate_payroll()}')
AttributeError: 'Employee' object has no attribute 'calculate_payroll'
```

While you can instantiate an `Employee` object, the object can't be used by the `PayrollSystem`. Why? Because it can't `.calculate_payroll()` for an `Employee`. To meet the requirements of `PayrollSystem`, you'll want to convert the `Employee` class, which is currently a concrete class, to an abstract class. That way, no employee is ever just an `Employee`, but one that implements `.calculate_payroll()`.

Abstract Base Classes in Python

The `Employee` class in the example above is what is called an abstract base class. Abstract base classes exist to be inherited, but never instantiated. Python provides the `abc` module to define abstract base classes.

You can use [leading underscores](#) in your class name to communicate that objects of that class should not be created. Underscores provide a friendly way to prevent misuse of your code, but they don't prevent eager users from creating instances of that class.

The [abc module](#) in the Python standard library provides functionality to prevent creating objects from abstract base classes.

You can modify the implementation of the `Employee` class to ensure that it can't be instantiated:

Python

```
# In hr.py

from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, id, name):
        self.id = id
        self.name = name

    @abstractmethod
    def calculate_payroll(self):
        pass
```

You derive `Employee` from `ABC`, making it an abstract base class. Then, you decorate the `.calculate_payroll()` method with the [@abstractmethod decorator](#).

This change has two nice side-effects:

1. You're telling users of the module that objects of type `Employee` can't be created.
2. You're telling other developers working on the `hr` module that if they derive from `Employee`, then they must override the `.calculate_payroll()` abstract method.

You can see that objects of type `Employee` can't be created using the interactive interpreter:

Python

>>>

```
>>> import hr
>>> employee = hr.Employee(1, 'abstract')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Employee with abstract methods
calculate_payroll
```

The output shows that the class cannot be instantiated because it contains an abstract method `calculate_payroll()`. Derived classes must override the method to allow creating objects of their type.

Implementation Inheritance vs Interface Inheritance

When you derive one class from another, the derived class inherits both:

1. **The base class interface:** The derived class inherits all the methods, properties, and attributes of the base class.
2. **The base class implementation:** The derived class inherits the code that implements the class interface.

Most of the time, you'll want to inherit the implementation of a class, but you will want to implement multiple interfaces, so your objects can be used in different situations.

Modern programming languages are designed with this basic concept in mind. They allow you to inherit from a single class, but you can implement multiple interfaces.

In Python, you don't have to explicitly declare an interface. Any object that implements the desired interface can be used in place of another object. This is known as [duck typing](#). Duck typing is usually explained as "if it behaves like a duck, then it's a duck."

To illustrate this, you will now add a `DisgruntledEmployee` class to the example above which doesn't derive from `Employee`:

Python

```
# In disgruntled.py

class DisgruntledEmployee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def calculate_payroll(self):
        return 1000000
```

The `DisgruntledEmployee` class doesn't derive from `Employee`, but it exposes the same interface required by the `PayrollSystem`. The `PayrollSystem.calculate_payroll()` requires a list of objects that implement the following interface:

- An `id` property or attribute that returns the employee's id
- A `name` property or attribute that represents the employee's name
- A `.calculate_payroll()` method that doesn't take any parameters and returns the payroll amount to process

All these requirements are met by the `DisgruntledEmployee` class, so the `PayrollSystem` can still calculate its payroll.

You can modify the program to use the `DisgruntledEmployee` class:

Python

```
# In program.py

import hr
import disgruntled

salary_employee = hr.SalaryEmployee(1, 'John Smith', 1500)
hourly_employee = hr.HourlyEmployee(2, 'Jane Doe', 40, 15)
commission_employee = hr.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)
disgruntled_employee = disgruntled.DisgruntledEmployee(20000, 'Anonymous')
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee,
    disgruntled_employee
])
```

The program creates a `DisgruntledEmployee` object and adds it to the list processed by the `PayrollSystem`. You can now run the program and see its output:

Shell

```
$ python program.py

Calculating Payroll
=====
Payroll for: 1 - John Smith
- Check amount: 1500

Payroll for: 2 - Jane Doe
- Check amount: 600

Payroll for: 3 - Kevin Bacon
- Check amount: 1250

Payroll for: 20000 - Anonymous
- Check amount: 1000000
```

As you can see, the `PayrollSystem` can still process the new object because it meets the desired interface.

Since you don't have to derive from a specific class for your objects to be reusable by the program, you may be asking why you should use inheritance instead of just implementing the desired interface. The following rules may help you:

- **Use inheritance to reuse an implementation:** Your derived classes should leverage most of their base class implementation. They must also model an **is a** relationship. A Customer class might also have an id and a name, but a Customer is not an Employee, so you should not use inheritance.
- **Implement an interface to be reused:** When you want your class to be reused by a specific part of your application, you implement the required interface in your class, but you don't need to provide a base class, or inherit from another class.

You can now clean up the example above to move onto the next topic. You can delete the `disgruntled.py` file and then modify the `hr` module to its original state:

Python

```
# In hr.py

class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('=====')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print()

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

You removed the import of the `abc` module since the `Employee` class doesn't need to be abstract. You also removed the abstract `calculate_payroll()` method from it since it doesn't provide any implementation.

Basically, you are inheriting the implementation of the `id` and `name` attributes of the `Employee` class in your derived classes. Since `.calculate_payroll()` is just an interface to the `PayrollSystem.calculate_payroll()` method, you don't need to implement it in the `Employee` base class.

Notice how the `CommissionEmployee` class derives from `SalaryEmployee`. This means that `CommissionEmployee` inherits the implementation and interface of `SalaryEmployee`. You can see how the `CommissionEmployee.calculate_payroll()` method leverages the base class implementation because it relies on the result from `super().calculate_payroll()` to implement its own version.

The Class Explosion Problem

If you are not careful, inheritance can lead you to a huge hierarchical structure of classes that is hard to understand and maintain. This is known as the **class explosion problem**.

You started building a class hierarchy of `Employee` types used by the `PayrollSystem` to calculate payroll. Now, you need to add some functionality to those classes, so they can be used with the new `ProductivitySystem`.

The `ProductivitySystem` tracks productivity based on employee roles. There are different employee roles:

- **Managers:** They walk around yelling at people telling them what to do. They are salaried employees and make more money.
- **Secretaries:** They do all the paper work for managers and ensure that everything gets billed and payed on time. They are also salaried employees but make less money.
- **Sales employees:** They make a lot of phone calls to sell products. They have a salary, but they also get commissions for sales.
- **Factory workers:** They manufacture the products for the company. They are paid by the hour.

With those requirements, you start to see that `Employee` and its derived classes might belong somewhere other than the `hr` module because now they're also used by the `ProductivitySystem`.

You create an `employees` module and move the classes there:

Python

```
# In employees.py

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

The implementation remains the same, but you move the classes to the `employee` module. Now, you change your program to support the change:

Python

```
# In program.py

import hr
import employees

salary_employee = employees.SalaryEmployee(1, 'John Smith', 1500)
hourly_employee = employees.HourlyEmployee(2, 'Jane Doe', 40, 15)
commission_employee = employees.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee
])
```

You run the program and verify that it still works:

Shell

```
$ python program.py

Calculating Payroll
=====
Payroll for: 1 - John Smith
- Check amount: 1500

Payroll for: 2 - Jane Doe
- Check amount: 600

Payroll for: 3 - Kevin Bacon
- Check amount: 1250
```

With everything in place, you start adding the new classes:

Python

```
# In employees.py

class Manager(SalaryEmployee):
    def work(self, hours):
        print(f'{self.name} screams and yells for {hours} hours.')

class Secretary(SalaryEmployee):
    def work(self, hours):
        print(f'{self.name} expends {hours} hours doing office paperwork.')

class SalesPerson(CommissionEmployee):
    def work(self, hours):
        print(f'{self.name} expends {hours} hours on the phone.')

class FactoryWorker(HourlyEmployee):
    def work(self, hours):
        print(f'{self.name} manufactures gadgets for {hours} hours.)
```

First, you add a Manager class that derives from SalaryEmployee. The class exposes a method work() that will be used by the productivity system. The method takes the hours the employee worked.

Then you add Secretary, SalesPerson, and FactoryWorker and then implement the work() interface, so they can be used by the productivity system.

Now, you can add the ProductivitySystem class:

Python

```
# In productivity.py

class ProductivitySystem:
    def track(self, employees, hours):
        print('Tracking Employee Productivity')
        print('=====')
        for employee in employees:
            employee.work(hours)
        print('')
```

The class tracks employees in the `track()` method that takes a list of employees and the number of hours to track. You can now add the productivity system to your program:

Python

```
# In program.py

import hr
import employees
import productivity

manager = employees.Manager(1, 'Mary Poppins', 3000)
secretary = employees.Secretary(2, 'John Smith', 1500)
sales_guy = employees.SalesPerson(3, 'Kevin Bacon', 1000, 250)
factory_worker = employees.FactoryWorker(2, 'Jane Doe', 40, 15)
employees = [
    manager,
    secretary,
    sales_guy,
    factory_worker,
]
productivity_system = productivity.ProductivitySystem()
productivity_system.track(employees, 40)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll(employees)
```

The program creates a list of employees of different types. The employee list is sent to the productivity system to track their work for 40 hours. Then the same list of employees is sent to the payroll system to calculate their payroll.

You can run the program to see the output:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Mary Poppins screams and yells for 40 hours.
John Smith expends 40 hours doing office paperwork.
Kevin Bacon expends 40 hours on the phone.
Jane Doe manufactures gadgets for 40 hours.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000

Payroll for: 2 - John Smith
- Check amount: 1500

Payroll for: 3 - Kevin Bacon
- Check amount: 1250

Payroll for: 4 - Jane Doe
- Check amount: 600
```

The program shows the employees working for 40 hours through the productivity system. Then it calculates and displays the payroll for each of the employees.

The program works as expected, but you had to add four new classes to support the changes. As new requirements come, your class hierarchy will inevitably grow, leading to the class explosion problem where your hierarchies will become so big that they'll be hard to understand and maintain.

The following diagram shows the new class hierarchy:

```
graph TD; Requirements --> Feature1[Feature]; Requirements --> Feature2[Feature]; Requirements --> Feature3[Feature]; Requirements --> Feature4[Feature]; Feature1 --> Implementation1[Implementation]; Feature1 --> Implementation2[Implementation]; Feature1 --> Implementation3[Implementation]; Feature1 --> Implementation4[Implementation]; Feature2 --> Implementation5[Implementation]; Feature2 --> Implementation6[Implementation]; Feature2 --> Implementation7[Implementation]; Feature2 --> Implementation8[Implementation]; Feature3 --> Implementation9[Implementation]; Feature3 --> Implementation10[Implementation]; Feature3 --> Implementation11[Implementation]; Feature3 --> Implementation12[Implementation]; Feature4 --> Implementation13[Implementation]; Feature4 --> Implementation14[Implementation]; Feature4 --> Implementation15[Implementation]; Feature4 --> Implementation16[Implementation]
```

The diagram shows how the class hierarchy is growing. Additional requirements might have an exponential effect in the number of classes with this design.

Inheriting Multiple Classes

Python is one of the few modern programming languages that supports multiple inheritance. Multiple inheritance is the ability to derive a class from multiple base classes at the same time.

Multiple inheritance has a bad reputation to the extent that most modern programming languages don't support it. Instead, modern programming languages support the concept of interfaces. In those languages, you inherit from a single base class and then implement multiple interfaces, so your class can be re-used in different situations.

This approach puts some constraints in your designs. You can only inherit the implementation of one class by directly deriving from it. You can implement multiple interfaces, but you can't inherit the implementation of multiple classes.

This constraint is good for software design because it forces you to design your classes with fewer dependencies on each other. You will see later in this article that you can leverage multiple implementations through composition, which makes software more flexible. This section, however, is about multiple inheritance, so let's take a look at how it works.

It turns out that sometimes temporary secretaries are hired when there is too much paperwork to do. The `TemporarySecretary` class performs the role of a `Secretary` in the context of the `ProductivitySystem`, but for payroll purposes, it is an `HourlyEmployee`.

You look at your class design. It has grown a little bit, but you can still understand how it works. It seems you have two options:

1. **Derive from Secretary:** You can derive from `Secretary` to inherit the `.work()` method for the role, and then override the `.calculate_payroll()` method to implement it as an `HourlyEmployee`.

2. **Derive from HourlyEmployee:** You can derive from `HourlyEmployee` to inherit the `.calculate_payroll()` method, and then override the `.work()` method to implement it as a `Secretary`.

Then, you remember that Python supports multiple inheritance, so you decide to derive from both `Secretary` and `HourlyEmployee`:

Python

```
# In employees.py

class TemporarySecretary(Secretary, HourlyEmployee):
    pass
```

Python allows you to inherit from two different classes by specifying them between parenthesis in the class declaration.

Now, you modify your program to add the new temporary secretary employee:

Python

```
import hr
import employees
import productivity

manager = employees.Manager(1, 'Mary Poppins', 3000)
secretary = employees.Secretary(2, 'John Smith', 1500)
sales_guy = employees.SalesPerson(3, 'Kevin Bacon', 1000, 250)
factory_worker = employees.FactoryWorker(4, 'Jane Doe', 40, 15)
temporary_secretary = employees.TemporarySecretary(5, 'Robin Williams', 40, 9)
company_employees = [
    manager,
    secretary,
    sales_guy,
    factory_worker,
    temporary_secretary,
]
productivity_system = productivity.ProductivitySystem()
productivity_system.track(company_employees, 40)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll(company_employees)
```

You run the program to test it:

Shell

```
$ python program.py

Traceback (most recent call last):
  File ".\program.py", line 9, in <module>
    temporary_secretary = employee.TemporarySecretary(5, 'Robin Williams', 40, 9)
TypeError: __init__() takes 4 positional arguments but 5 were given
```

You get a `TypeError` exception saying that 4 positional arguments where expected, but 5 were given.

This is because you derived `TemporarySecretary` first from `Secretary` and then from `HourlyEmployee`, so the interpreter is trying to use `Secretary.__init__()` to initialize the object.

Okay, let's reverse it:

Python

```
class TemporarySecretary(HourlyEmployee, Secretary):
    pass
```

Now, run the program again and see what happens:

Shell

```
$ python program.py

Traceback (most recent call last):
  File ".\program.py", line 9, in <module>
    temporary_secretary = employee.TemporarySecretary(5, 'Robin Williams', 40, 9)
  File "employee.py", line 16, in __init__
    super().__init__(id, name)
TypeError: __init__() missing 1 required positional argument: 'weekly_salary'
```

Now it seems you are missing a `weekly_salary` parameter, which is necessary to initialize `Secretary`, but that parameter doesn't make sense in the context of a `TemporarySecretary` because it's an `HourlyEmployee`.

Maybe implementing `TemporarySecretary.__init__()` will help:

Python

```
# In employees.py

class TemporarySecretary(HourlyEmployee, Secretary):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name, hours_worked, hour_rate)
```

Try it:

Shell

```
$ python program.py

Traceback (most recent call last):
  File ".\program.py", line 9, in <module>
    temporary_secretary = employee.TemporarySecretary(5, 'Robin Williams', 40, 9)
  File "employee.py", line 54, in __init__
    super().__init__(id, name, hours_worked, hour_rate)
  File "employee.py", line 16, in __init__
    super().__init__(id, name)
TypeError: __init__() missing 1 required positional argument: 'weekly_salary'
```

That didn't work either. Okay, it's time for you to dive into Python's **method resolution order** (MRO) to see what's going on.

When a method or attribute of a class is accessed, Python uses the class [MRO](#) to find it. The MRO is also used by `super()` to determine which method or attribute to invoke. You can learn more about `super()` in [Supercharge Your Classes With Python super\(\)](#).

You can evaluate the `TemporarySecretary` class MRO using the interactive interpreter:

Python

>>>

```
>>> from employees import TemporarySecretary
>>> TemporarySecretary.__mro__

(<class 'employees.TemporarySecretary'>,
 <class 'employees.HourlyEmployee'>,
 <class 'employees.Secretary'>,
 <class 'employees.SalaryEmployee'>,
 <class 'employees.Employee'>,
 <class 'object'>
 )
```

The MRO shows the order in which Python is going to look for a matching attribute or method. In the example, this is what happens when we create the `TemporarySecretary` object:

1. The `TemporarySecretary.__init__(self, id, name, hours_worked, hour_rate)` method is called.
2. The `super().__init__(id, name, hours_worked, hour_rate)` call matches `HourlyEmployee.__init__(self, id, name, hour_worked, hour_rate)`.
3. `HourlyEmployee` calls `super().__init__(id, name)`, which the MRO is going to match to `Secretary.__init__()`, which is inherited from `SalaryEmployee.__init__(self, id, name, weekly_salary)`.

Because the parameters don't match, a `TypeError` exception is raised.

You can bypass the MRO by reversing the inheritance order and directly calling `HourlyEmployee.__init__()` as follows:

Python

```
class TemporarySecretary(Secretary, HourlyEmployee):
    def __init__(self, id, name, hours_worked, hour_rate):
        HourlyEmployee.__init__(self, id, name, hours_worked, hour_rate)
```

That solves the problem of creating the object, but you will run into a similar problem when trying to calculate payroll. You can run the program to see the problem:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Mary Poppins screams and yells for 40 hours.
John Smith expends 40 hours doing office paperwork.
Kevin Bacon expends 40 hours on the phone.
Jane Doe manufactures gadgets for 40 hours.
Robin Williams expends 40 hours doing office paperwork.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000

Payroll for: 2 - John Smith
- Check amount: 1500

Payroll for: 3 - Kevin Bacon
- Check amount: 1250

Payroll for: 4 - Jane Doe
- Check amount: 600

Payroll for: 5 - Robin Williams
Traceback (most recent call last):
  File ".\program.py", line 20, in <module>
    payroll_system.calculate_payroll(employees)
  File "hr.py", line 7, in calculate_payroll
    print(f'- Check amount: {employee.calculate_payroll()}')
  File "employee.py", line 12, in calculate_payroll
    return self.weekly_salary
AttributeError: 'TemporarySecretary' object has no attribute 'weekly_salary'
```

The problem now is that because you reversed the inheritance order, the MRO is finding the `.calculate_payroll()` method of `SalariedEmployee` before the one in `HourlyEmployee`. You need to override `.calculate_payroll()` in `TemporarySecretary` and invoke the right implementation from it:

Python

```
class TemporarySecretary(Secretary, HourlyEmployee):
    def __init__(self, id, name, hours_worked, hour_rate):
        HourlyEmployee.__init__(self, id, name, hours_worked, hour_rate)

    def calculate_payroll(self):
        return HourlyEmployee.calculate_payroll(self)
```

The `calculate_payroll()` method directly invokes `HourlyEmployee.calculate_payroll()` to ensure that you get the correct result. You can run the program again to see it working:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Mary Poppins screams and yells for 40 hours.
John Smith expends 40 hours doing office paperwork.
Kevin Bacon expends 40 hours on the phone.
Jane Doe manufactures gadgets for 40 hours.
Robin Williams expends 40 hours doing office paperwork.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000

Payroll for: 2 - John Smith
- Check amount: 1500

Payroll for: 3 - Kevin Bacon
- Check amount: 1250

Payroll for: 4 - Jane Doe
- Check amount: 600

Payroll for: 5 - Robin Williams
- Check amount: 360
```

The program now works as expected because you're forcing the method resolution order by explicitly telling the interpreter which method we want to use.

As you can see, multiple inheritance can be confusing, especially when you run into the [diamond problem](#).

The following diagram shows the diamond problem in your class hierarchy:

The diagram shows the diamond problem with the current class design. `TemporarySecretary` uses multiple inheritance to derive from two classes that ultimately also derive from `Employee`. This causes two paths to reach the `Employee` base class, which is something you want to avoid in your designs.

The diamond problem appears when you're using multiple inheritance and deriving from two classes that have a common base class. This can cause the wrong version of a method to be called.

As you've seen, Python provides a way to force the right method to be invoked, and analyzing the MRO can help you understand the problem.

Still, when you run into the diamond problem, it's better to re-think the design. You will now make some changes to leverage multiple inheritance, avoiding the diamond problem.

The `Employee` derived classes are used by two different systems:

1. **The productivity system** that tracks employee productivity.
2. **The payroll system** that calculates the employee payroll.

This means that everything related to productivity should be together in one module and everything related to payroll should be together in another. You can start making changes to the productivity module:

Python

```
# In productivity.py

class ProductivitySystem:
    def track(self, employees, hours):
        print('Tracking Employee Productivity')
        print('=====')
        for employee in employees:
            result = employee.work(hours)
            print(f'{employee.name}: {result}')
        print()

class ManagerRole:
    def work(self, hours):
        return f'screams and yells for {hours} hours.'

class SecretaryRole:
    def work(self, hours):
        return f'expends {hours} hours doing office paperwork.'

class SalesRole:
    def work(self, hours):
        return f'expends {hours} hours on the phone.'

class FactoryRole:
    def work(self, hours):
        return f'manufactures gadgets for {hours} hours.'
```

The productivity module implements the `ProductivitySystem` class, as well as the related roles it supports. The classes implement the `work()` interface required by the system, but they don't derive from `Employee`.

You can do the same with the `hr` module:

Python

```
# In hr.py

class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('=====')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print()

class SalaryPolicy:
    def __init__(self, weekly_salary):
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyPolicy:
    def __init__(self, hours_worked, hour_rate):
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionPolicy(SalaryPolicy):
    def __init__(self, weekly_salary, commission):
        super().__init__(weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

The `hr` module implements the `PayrollSystem`, which calculates payroll for the employees. It also implements the policy classes for payroll. As you can see, the policy classes don't derive from `Employee` anymore.

You can now add the necessary classes to the employee module:

Python

```
# In employees.py

from hr import (
    SalaryPolicy,
    CommissionPolicy,
    HourlyPolicy
)
from productivity import (
    ManagerRole,
    SecretaryRole,
    SalesRole,
    FactoryRole
)

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class Manager(Employee, ManagerRole, SalaryPolicy):
    def __init__(self, id, name, weekly_salary):
        SalaryPolicy.__init__(self, weekly_salary)
        super().__init__(id, name)

class Secretary(Employee, SecretaryRole, SalaryPolicy):
    def __init__(self, id, name, weekly_salary):
        SalaryPolicy.__init__(self, weekly_salary)
        super().__init__(id, name)

class SalesPerson(Employee, SalesRole, CommissionPolicy):
    def __init__(self, id, name, weekly_salary, commission):
        CommissionPolicy.__init__(self, weekly_salary, commission)
        super().__init__(id, name)

class FactoryWorker(Employee, FactoryRole, HourlyPolicy):
    def __init__(self, id, name, hours_worked, hour_rate):
        HourlyPolicy.__init__(self, hours_worked, hour_rate)
        super().__init__(id, name)

class TemporarySecretary(Employee, SecretaryRole, HourlyPolicy):
    def __init__(self, id, name, hours_worked, hour_rate):
        HourlyPolicy.__init__(self, hours_worked, hour_rate)
        super().__init__(id, name)
```

The employees module imports policies and roles from the other modules and implements the different Employee types. You are still using multiple inheritance to inherit the implementation of the salary policy classes and the productivity roles, but the implementation of each class only needs to deal with initialization.

Notice that you still need to explicitly initialize the salary policies in the constructors. You probably saw that the initializations of Manager and Secretary are identical. Also, the initializations of FactoryWorker and TemporarySecretary are the same.

You will not want to have this kind of code duplication in more complex designs, so you have to be careful when designing class hierarchies.

Here's the UML diagram for the new design:

The diagram shows the relationships to define the `Secretary` and `TemporarySecretary` using multiple inheritance, but avoiding the diamond problem.

You can run the program and see how it works:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Mary Poppins: screams and yells for 40 hours.
John Smith: expends 40 hours doing office paperwork.
Kevin Bacon: expends 40 hours on the phone.
Jane Doe: manufactures gadgets for 40 hours.
Robin Williams: expends 40 hours doing office paperwork.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000

Payroll for: 2 - John Smith
- Check amount: 1500

Payroll for: 3 - Kevin Bacon
- Check amount: 1250

Payroll for: 4 - Jane Doe
- Check amount: 600

Payroll for: 5 - Robin Williams
- Check amount: 360
```

You've seen how inheritance and multiple inheritance work in Python. You can now explore the topic of composition.

Composition in Python

Composition is an object oriented design concept that models a **has a** relationship. In composition, a class known as **composite** contains an object of another class known to as **component**. In other words, a composite class **has a** component of another class.

Composition allows composite classes to reuse the implementation of the components it contains. The composite class doesn't inherit the component class interface, but it can leverage its implementation.

The composition relation between two classes is considered loosely coupled. That means that changes to the component class rarely affect the composite class, and changes to the composite class never affect the component class.

This provides better adaptability to change and allows applications to introduce new requirements without affecting existing code.

When looking at two competing software designs, one based on inheritance and another based on composition, the composition solution usually is the most flexible. You can now look at how composition works.

You've already used composition in our examples. If you look at the Employee class, you'll see that it contains two attributes:

1. **id** to identify an employee.
2. **name** to contain the name of the employee.

These two attributes are objects that the Employee class has. Therefore, you can say that an Employee **has an id** and **has a name**.

Another attribute for an Employee might be an Address:

Python

```
# In contacts.py

class Address:
    def __init__(self, street, city, state, zipcode, street2=''):
        self.street = street
        self.street2 = street2
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def __str__(self):
        lines = [self.street]
        if self.street2:
            lines.append(self.street2)
        lines.append(f'{self.city}, {self.state} {self.zipcode}')
        return '\n'.join(lines)
```

You implemented a basic address class that contains the usual components for an address. You made the `street2` attribute optional because not all addresses will have that component.

You implemented `__str__()` to provide a pretty representation of an Address. You can see this implementation in the interactive interpreter:

Python

>>>

```
>>> from contacts import Address
>>> address = Address('55 Main St.', 'Concord', 'NH', '03301')
>>> print(address)

55 Main St.
Concord, NH 03301
```

When you `print()` the `address` variable, the special method `__str__()` is invoked. Since you overloaded the method to return a string formatted as an address, you get a nice, readable representation. [Operator and Function Overloading in Custom Python Classes](#) gives a good overview of the special methods available in classes that can be implemented to customize the behavior of your objects.

You can now add the Address to the Employee class through composition:

Python

```
# In employees.py

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name
        self.address = None
```

You initialize the address attribute to None for now to make it optional, but by doing that, you can now assign an Address to an Employee. Also notice that there is no reference in the employee module to the contacts module.

Composition is a loosely coupled relationship that often doesn't require the composite class to have knowledge of the component.

The UML diagram representing the relationship between Employee and Address looks like this:



The diagram shows a composition relationship between Employee and Address. It consists of two rounded rectangles, one labeled "Employee" and the other "Address". A solid line connects them, with a hollow diamond symbol at the "Employee" end, indicating that Employee is the composite class and Address is the component.

The diagram shows the basic composition relationship between Employee and Address.

You can now modify the PayrollSystem class to leverage the address attribute in Employee:

Python

```
# In hr.py

class PayrollSystem:
    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('=====')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            if employee.address:
                print('- Sent to:')
                print(employee.address)
            print()
```

You check to see if the employee object has an address, and if it does, you print it. You can now modify the program to assign some addresses to the employees:

Python

```
# In program.py

import hr
import employees
import productivity
import contacts

manager = employees.Manager(1, 'Mary Poppins', 3000)
manager.address = contacts.Address(
    '121 Admin Rd',
    'Concord',
    'NH',
    '03301'
)
secretary = employees.Secretary(2, 'John Smith', 1500)
secretary.address = contacts.Address(
    '67 Paperwork Ave.',
    'Manchester',
    'NH',
    '03101'
)
sales_guy = employees.SalesPerson(3, 'Kevin Bacon', 1000, 250)
factory_worker = employees.FactoryWorker(4, 'Jane Doe', 40, 15)
temporary_secretary = employees.TemporarySecretary(5, 'Robin Williams', 40, 9)
employees = [
    manager,
    secretary,
    sales_guy,
    factory_worker,
    temporary_secretary,
]
productivity_system = productivity.ProductivitySystem()
productivity_system.track(employees, 40)
payroll_system = hr.PayrollSystem()
payroll_system.calculate_payroll(employees)
```

You added a couple of addresses to the `manager` and `secretary` objects. When you run the program, you will see the addresses printed:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Mary Poppins: screams and yells for {hours} hours.
John Smith: expends {hours} hours doing office paperwork.
Kevin Bacon: expends {hours} hours on the phone.
Jane Doe: manufactures gadgets for {hours} hours.
Robin Williams: expends {hours} hours doing office paperwork.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000
- Sent to:
121 Admin Rd
Concord, NH 03301

Payroll for: 2 - John Smith
- Check amount: 1500
- Sent to:
67 Paperwork Ave.
Manchester, NH 03101

Payroll for: 3 - Kevin Bacon
- Check amount: 1250

Payroll for: 4 - Jane Doe
- Check amount: 600

Payroll for: 5 - Robin Williams
- Check amount: 360
```

Notice how the payroll output for the `manager` and `secretary` objects show the addresses where the checks were sent.

The `Employee` class leverages the implementation of the `Address` class without any knowledge of what an `Address` object is or how it's represented. This type of design is so flexible that you can change the `Address` class without any impact to the `Employee` class.

Flexible Designs With Composition

Composition is more flexible than inheritance because it models a loosely coupled relationship. Changes to a component class have minimal or no effects on the composite class. Designs based on composition are more suitable to change.

You change behavior by providing new components that implement those behaviors instead of adding new classes to your hierarchy.

Take a look at the multiple inheritance example above. Imagine how new payroll policies will affect the design. Try to picture what the class hierarchy will look like if new roles are needed. As you saw before, relying too heavily on inheritance can lead to class explosion.

The biggest problem is not so much the number of classes in your design, but how tightly coupled the relationships between those classes are. Tightly coupled classes affect each other when changes are introduced.

In this section, you are going to use composition to implement a better design that still fits the requirements of the `PayrollSystem` and the `ProductivitySystem`.

You can start by implementing the functionality of the `ProductivitySystem`:

Python

```
# In productivity.py

class ProductivitySystem:
    def __init__(self):
        self._roles = {
            'manager': ManagerRole,
            'secretary': SecretaryRole,
            'sales': SalesRole,
            'factory': FactoryRole,
        }

    def get_role(self, role_id):
        role_type = self._roles.get(role_id)
        if not role_type:
            raise ValueError('role_id')
        return role_type()

    def track(self, employees, hours):
        print('Tracking Employee Productivity')
        print('=====')
        for employee in employees:
            employee.work(hours)
        print('')
```

The `ProductivitySystem` class defines some roles using a string identifier mapped to a role class that implements the role. It exposes a `.get_role()` method that, given a role identifier, returns the role type object. If the role is not found, then a `ValueError` exception is raised.

It also exposes the previous functionality in the `.track()` method, where given a list of employees it tracks the productivity of those employees.

You can now implement the different role classes:

Python

```
# In productivity.py

class ManagerRole:
    def perform_duties(self, hours):
        return f'screams and yells for {hours} hours.'

class SecretaryRole:
    def perform_duties(self, hours):
        return f'does paperwork for {hours} hours.'

class SalesRole:
    def perform_duties(self, hours):
        return f'expends {hours} hours on the phone.'

class FactoryRole:
    def perform_duties(self, hours):
        return f'manufactures gadgets for {hours} hours.'
```

Each of the roles you implemented expose a `.perform_duties()` that takes the number of hours worked. The methods return a string representing the duties.

The role classes are independent of each other, but they expose the same interface, so they are interchangeable. You'll see later how they are used in the application.

Now, you can implement the `PayrollSystem` for the application:

Python

```
# In hr.py

class PayrollSystem:
    def __init__(self):
        self._employee_policies = {
            1: SalaryPolicy(3000),
            2: SalaryPolicy(1500),
            3: CommissionPolicy(1000, 100),
            4: HourlyPolicy(15),
            5: HourlyPolicy(9)
        }

    def get_policy(self, employee_id):
        policy = self._employee_policies.get(employee_id)
        if not policy:
            return ValueError(employee_id)
        return policy

    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('=====')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            if employee.address:
                print('- Sent to:')
                print(employee.address)
            print('')
```

The `PayrollSystem` keeps an internal database of payroll policies for each employee. It exposes a `.get_policy()` that, given an employee id, returns its payroll policy. If a specified id doesn't exist in the system, then the method raises a `ValueError` exception.

The implementation of `.calculate_payroll()` works the same as before. It takes a list of employees, calculates the payroll, and prints the results.

You can now implement the payroll policy classes:

Python

```
# In hr.py

class PayrollPolicy:
    def __init__(self):
        self.hours_worked = 0

    def track_work(self, hours):
        self.hours_worked += hours

class SalaryPolicy(PayrollPolicy):
    def __init__(self, weekly_salary):
        super().__init__()
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyPolicy(PayrollPolicy):
    def __init__(self, hour_rate):
        super().__init__()
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionPolicy(SalaryPolicy):
    def __init__(self, weekly_salary, commission_per_sale):
        super().__init__(weekly_salary)
        self.commission_per_sale = commission_per_sale

    @property
    def commission(self):
        sales = self.hours_worked / 5
        return sales * self.commission_per_sale

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission
```

You first implement a `PayrollPolicy` class that serves as a base class for all the payroll policies. This class tracks the `hours_worked`, which is common to all payroll policies.

The other policy classes derive from `PayrollPolicy`. We use inheritance here because we want to leverage the implementation of `PayrollPolicy`. Also, `SalaryPolicy`, `HourlyPolicy`, and `CommissionPolicy` **are a** `PayrollPolicy`.

`SalaryPolicy` is initialized with a `weekly_salary` value that is then used in `.calculate_payroll()`. `HourlyPolicy` is initialized with the `hour_rate`, and implements `.calculate_payroll()` by leveraging the base class `hours_worked`.

The `CommissionPolicy` class derives from `SalaryPolicy` because it wants to inherit its implementation. It is initialized with the `weekly_salary` parameters, but it also requires a `commission_per_sale` parameter.

The `commission_per_sale` is used to calculate the `.commission`, which is implemented as a property so it gets calculated when requested. In the example, we are assuming that a sale happens every 5 hours worked, and the `.commission` is the number of sales times the `commission_per_sale` value.

`CommissionPolicy` implements the `.calculate_payroll()` method by first leveraging the implementation in `SalaryPolicy` and then adding the calculated commission.

You can now add an `AddressBook` class to manage employee addresses:

Python

```
# In contacts.py

class AddressBook:
    def __init__(self):
        self._employee_addresses = {
            1: Address('121 Admin Rd.', 'Concord', 'NH', '03301'),
            2: Address('67 Paperwork Ave', 'Manchester', 'NH', '03101'),
            3: Address('15 Rose St', 'Concord', 'NH', '03301', 'Apt. B-1'),
            4: Address('39 Sole St.', 'Concord', 'NH', '03301'),
            5: Address('99 Mountain Rd.', 'Concord', 'NH', '03301'),
        }

    def get_employee_address(self, employee_id):
        address = self._employee_addresses.get(employee_id)
        if not address:
            raise ValueError(employee_id)
        return address
```

The `AddressBook` class keeps an internal database of `Address` objects for each employee. It exposes a `get_employee_address()` method that returns the address of the specified employee `id`. If the employee `id` doesn't exist, then it raises a `ValueError`.

The `Address` class implementation remains the same as before:

Python

```
# In contacts.py

class Address:
    def __init__(self, street, city, state, zipcode, street2=''):
        self.street = street
        self.street2 = street2
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def __str__(self):
        lines = [self.street]
        if self.street2:
            lines.append(self.street2)
        lines.append(f'{self.city}, {self.state} {self.zipcode}')
        return '\n'.join(lines)
```

The class manages the address components and provides a pretty representation of an address.

So far, the new classes have been extended to support more functionality, but there are no significant changes to the previous design. This is going to change with the design of the `employees` module and its classes.

You can start by implementing an `EmployeeDatabase` class:

Python

```
# In employees.py

from productivity import ProductivitySystem
from hr import PayrollSystem
from contacts import AddressBook

class EmployeeDatabase:
    def __init__(self):
        self._employees = [
            {
                'id': 1,
                'name': 'Mary Poppins',
                'role': 'manager'
            },
            {
                'id': 2,
                'name': 'John Smith',
                'role': 'secretary'
            },
            {
                'id': 3,
                'name': 'Kevin Bacon',
                'role': 'sales'
            },
            {
                'id': 4,
                'name': 'Jane Doe',
                'role': 'factory'
            },
            {
                'id': 5,
                'name': 'Robin Williams',
                'role': 'secretary'
            },
        ]
        self.productivity = ProductivitySystem()
        self.payroll = PayrollSystem()
        self.employee_addresses = AddressBook()

    @property
    def employees(self):
        return [self._create_employee(**data) for data in self._employees]

    def _create_employee(self, id, name, role):
        address = self.employee_addresses.get_employee_address(id)
        employee_role = self.productivity.get_role(role)
        payroll_policy = self.payroll.get_policy(id)
        return Employee(id, name, address, employee_role, payroll_policy)
```

The `EmployeeDatabase` keeps track of all the employees in the company. For each employee, it tracks the `id`, `name`, and `role`. It **has an** instance of the `ProductivitySystem`, the `PayrollSystem`, and the `AddressBook`. These instances are used to create employees.

It exposes an `.employees` property that returns the list of employees. The `Employee` objects are created in an internal method `._create_employee()`. Notice that you don't have different types of `Employee` classes. You just need to implement a single `Employee` class:

Python

```
# In employees.py

class Employee:
    def __init__(self, id, name, address, role, payroll):
        self.id = id
        self.name = name
        self.address = address
        self.role = role
        self.payroll = payroll

    def work(self, hours):
        duties = self.role.perform_duties(hours)
        print(f'Employee {self.id} - {self.name}:')
        print(f'- {duties}')
        print('')
        self.payroll.track_work(hours)

    def calculate_payroll(self):
        return self.payroll.calculate_payroll()
```

The `Employee` class is initialized with the `id`, `name`, and `address` attributes. It also requires the productivity `role` for the employee and the `payroll` policy.

The class exposes a `.work()` method that takes the hours worked. This method first retrieves the duties from the `role`. In other words, it delegates to the `role` object to perform its duties.

In the same way, it delegates to the `payroll` object to track the work hours. The `payroll`, as you saw, uses those hours to calculate the payroll if needed.

The following diagram shows the composition design used:

The diagram shows the design of composition based policies. There is a single Employee that is composed of other data objects like Address and depends on the IRole and IPayrollCalculator interfaces to delegate the work. There are multiple implementations of these interfaces.

You can now use this design in your program:

Python

```
# In program.py

from hr import PayrollSystem
from productivity import ProductivitySystem
from employees import EmployeeDatabase

productivity_system = ProductivitySystem()
payroll_system = PayrollSystem()
employee_database = EmployeeDatabase()
employees = employee_database.employees
productivity_system.track(employees, 40)
payroll_system.calculate_payroll(employees)
```

You can run the program to see its output:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Employee 1 - Mary Poppins:
- screams and yells for 40 hours.

Employee 2 - John Smith:
- does paperwork for 40 hours.

Employee 3 - Kevin Bacon:
- expends 40 hours on the phone.

Employee 4 - Jane Doe:
- manufactures gadgets for 40 hours.

Employee 5 - Robin Williams:
- does paperwork for 40 hours.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000
- Sent to:
121 Admin Rd.
Concord, NH 03301

Payroll for: 2 - John Smith
- Check amount: 1500
- Sent to:
67 Paperwork Ave
Manchester, NH 03101

Payroll for: 3 - Kevin Bacon
- Check amount: 1800.0
- Sent to:
15 Rose St
Apt. B-1
Concord, NH 03301

Payroll for: 4 - Jane Doe
- Check amount: 600
- Sent to:
39 Sole St.
Concord, NH 03301

Payroll for: 5 - Robin Williams
- Check amount: 360
- Sent to:
99 Mountain Rd.
Concord, NH 03301
```

This design is what is called [policy-based design](#), where classes are composed of policies, and they delegate to those policies to do the work.

Policy-based design was introduced in the book [Modern C++ Design](#), and it uses template metaprogramming in C++ to achieve the results.

Python does not support templates, but you can achieve similar results using composition, as you saw in the example above.

This type of design gives you all the flexibility you'll need as requirements change. Imagine you need to change the way payroll is calculated for an object at run-time.

Customizing Behavior With Composition

If your design relies on inheritance, you need to find a way to change the type of an object to change its behavior. With composition, you just need to change the policy the object uses.

Imagine that our manager all of a sudden becomes a temporary employee that gets paid by the hour. You can modify the object during the execution of the program in the following way:

Python

```
# In program.py

from hr import PayrollSystem, HourlyPolicy
from productivity import ProductivitySystem
from employees import EmployeeDatabase

productivity_system = ProductivitySystem()
payroll_system = PayrollSystem()
employee_database = EmployeeDatabase()
employees = employee_database.employees
manager = employees[0]
manager.payroll = HourlyPolicy(55)

productivity_system.track(employees, 40)
payroll_system.calculate_payroll(employees)
```

The program gets the employee list from the `EmployeeDatabase` and retrieves the first employee, which is the manager we want. Then it creates a new `HourlyPolicy` initialized at \$55 per hour and assigns it to the `manager` object.

The new policy is now used by the `PayrollSystem` modifying the existing behavior. You can run the program again to see the result:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Employee 1 - Mary Poppins:
- screams and yells for 40 hours.

Employee 2 - John Smith:
- does paperwork for 40 hours.

Employee 3 - Kevin Bacon:
- expends 40 hours on the phone.

Employee 4 - Jane Doe:
- manufactures gadgets for 40 hours.

Employee 5 - Robin Williams:
- does paperwork for 40 hours.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 2200
- Sent to:
121 Admin Rd.
Concord, NH 03301

Payroll for: 2 - John Smith
- Check amount: 1500
- Sent to:
67 Paperwork Ave
Manchester, NH 03101

Payroll for: 3 - Kevin Bacon
- Check amount: 1800.0
- Sent to:
15 Rose St
Apt. B-1
Concord, NH 03301

Payroll for: 4 - Jane Doe
- Check amount: 600
- Sent to:
39 Sole St.
Concord, NH 03301

Payroll for: 5 - Robin Williams
- Check amount: 360
- Sent to:
99 Mountain Rd.
Concord, NH 03301
```

The check for Mary Poppins, our manager, is now for \$2200 instead of the fixed salary of \$3000 that she had per week.

Notice how we added that business rule to the program without changing any of the existing classes. Consider what type of changes would've been required with an inheritance design.

You would've had to create a new class and change the type of the manager employee. There is no chance you could've changed the policy at run-time.

Choosing Between Inheritance and Composition in Python

So far, you've seen how inheritance and composition work in Python. You've seen that derived classes inherit the interface and implementation of their base classes. You've also seen that composition allows you to reuse the implementation of another class.

You've implemented two solutions to the same problem. The first solution used multiple inheritance, and the second one used composition.

You've also seen that Python's duck typing allows you to reuse objects with existing parts of a program by implementing the desired interface. In Python, it isn't necessary to derive from a base class for your classes to be reused.

At this point, you might be asking when to use inheritance vs composition in Python. They both enable code reuse. Inheritance and composition can tackle similar problems in your Python programs.

The general advice is to use the relationship that creates fewer dependencies between two classes. This relation is composition. Still, there will be times where inheritance will make more sense.

The following sections provide some guidelines to help you make the right choice between inheritance and composition in Python.

Inheritance to Model “Is A” Relationship

Inheritance should only be used to model an **is a** relationship. Liskov's substitution principle says that an object of type Derived, which inherits from Base, can replace an object of type Base without altering the desirable properties of a program.

Liskov's substitution principle is the most important guideline to determine if inheritance is the appropriate design solution. Still, the answer might not be straightforward in all situations. Fortunately, there is a simple test you can use to determine if your design follows Liskov's substitution principle.

Let's say you have a class A that provides an implementation and interface you want to reuse in another class B. Your initial thought is that you can derive B from A and inherit both the interface and implementation. To be sure this is the right design, you follow these steps:

1. **Evaluate B is an A:** Think about this relationship and justify it. Does it make sense?
2. **Evaluate A is a B:** Reverse the relationship and justify it. Does it also make sense?

If you can justify both relationships, then you should never inherit those classes from one another. Let's look at a more concrete example.

You have a class Rectangle which exposes an .area property. You need a class Square, which also has an .area. It seems that a Square is a special type of Rectangle, so maybe you can derive from it and leverage both the interface and implementation.

Before you jump into the implementation, you use Liskov's substitution principle to evaluate the relationship.

A Square **is a** Rectangle because its area is calculated from the product of its height times its length. The constraint is that Square.height and Square.length must be equal.

It makes sense. You can justify the relationship and explain why a Square **is a** Rectangle. Let's reverse the relationship to see if it makes sense.

A Rectangle **is a** Square because its area is calculated from the product of its height times its length. The difference is that Rectangle.height and Rectangle.width can change independently.

It also makes sense. You can justify the relationship and describe the special constraints for each class. This is a good sign that these two classes should never derive from each other.

You might have seen other examples that derive Square from Rectangle to explain inheritance. You might be skeptical with the little test you just did. Fair enough. Let's write a program that illustrates the problem with deriving Square from Rectangle.

First, you implement Rectangle. You're even going to [encapsulate](#) the attributes to ensure that all the constraints are met:

Python

```
# In rectangle_square_demo.py

class Rectangle:
    def __init__(self, length, height):
        self._length = length
        self._height = height

    @property
    def area(self):
        return self._length * self._height
```

The Rectangle class is initialized with a length and a height, and it provides an .area property that returns the area. The length and height are encapsulated to avoid changing them directly.

Now, you derive Square from Rectangle and override the necessary interface to meet the constraints of a Square:

Python

```
# In rectangle_square_demo.py

class Square(Rectangle):
    def __init__(self, side_size):
        super().__init__(side_size, side_size)
```

The Square class is initialized with a side_size, which is used to initialize both components of the base class. Now, you write a small program to test the behavior:

Python

```
# In rectangle_square_demo.py

rectangle = Rectangle(2, 4)
assert rectangle.area == 8

square = Square(2)
assert square.area == 4

print('OK!')
```

The program creates a Rectangle and a Square and asserts that their .area is calculated correctly. You can run the program and see that everything is OK so far:

Shell

```
$ python rectangle_square_demo.py

OK!
```

The program executes correctly, so it seems that Square is just a special case of a Rectangle.

Later on, you need to support resizing Rectangle objects, so you make the appropriate changes to the class:

Python

```
# In rectangle_square_demo.py

class Rectangle:
    def __init__(self, length, height):
        self._length = length
        self._height = height

    @property
    def area(self):
        return self._length * self._height

    def resize(self, new_length, new_height):
        self._length = new_length
        self._height = new_height
```

.resize() takes the new_length and new_width for the object. You can add the following code to the program to verify that it works correctly:

Python

```
# In rectangle_square_demo.py

rectangle.resize(3, 5)
assert rectangle.area == 15

print('OK!')
```

You resize the rectangle object and assert that the new area is correct. You can run the program to verify the behavior:

Shell

```
$ python rectangle_square_demo.py

OK!
```

The assertion passes, and you see that the program runs correctly.

So, what happens if you resize a square? Modify the program, and try to modify the square object:

Python

```
# In rectangle_square_demo.py

square.resize(3, 5)
print(f'Square area: {square.area}')
```

You pass the same parameters to square.resize() that you used with rectangle, and print the area. When you run the program you see:

Shell

```
$ python rectangle_square_demo.py

Square area: 15
OK!
```

The program shows that the new area is 15 like the rectangle object. The problem now is that the square object no longer meets the Square class constraint that the length and height must be equal.

How can you fix that problem? You can try several approaches, but all of them will be awkward. You can override .resize() in square and ignore the height parameter, but that will be confusing for people looking at other parts of the program where rectangles are being resized and some of them are not getting the expected areas because they are really squares.

In a small program like this one, it might be easy to spot the causes of the weird behavior, but in a more complex program, the problem will be harder to find.

The reality is that if you're able to justify an inheritance relationship between two classes both ways, you should not derive one class from another.

In the example, it doesn't make sense that Square inherits the interface and implementation of .resize() from Rectangle. That doesn't mean that Square objects can't be resized. It means that the interface is different because it only needs a side_size parameter.

This difference in interface justifies not deriving Square from Rectangle like the test above advised.

Mixing Features With Mixin Classes

One of the uses of multiple inheritance in Python is to extend a class features through [mixins](#). A **mixin** is a class that provides methods to other classes but are not considered a base class.

A mixin allows other classes to reuse its interface and implementation without becoming a super class. They implement a unique behavior that can be aggregated to other unrelated classes. They are similar to composition but they create a stronger relationship.

Let's say you want to convert objects of certain types in your application to a dictionary representation of the object. You could provide a `.to_dict()` method in every class that you want to support this feature, but the implementation of `.to_dict()` seems to be very similar.

This could be a good candidate for a mixin. You start by slightly modifying the `Employee` class from the composition example:

Python

```
# In employees.py

class Employee:
    def __init__(self, id, name, address, role, payroll):
        self.id = id
        self.name = name
        self.address = address
        self._role = role
        self._payroll = payroll

    def work(self, hours):
        duties = self._role.perform_duties(hours)
        print(f'Employee {self.id} - {self.name}:')
        print(f'- {duties}')
        print('')
        self._payroll.track_work(hours)

    def calculate_payroll(self):
        return self._payroll.calculate_payroll()
```

The change is very small. You just changed the `role` and `payroll` attributes to be internal by adding a leading underscore to their name. You will see soon why you are making that change.

Now, you add the `AsDictionaryMixin` class:

Python

```
# In representations.py

class AsDictionaryMixin:
    def to_dict(self):
        return {
            prop: self._represent(value)
            for prop, value in self.__dict__.items()
            if not self._is_internal(prop)
        }

    def _represent(self, value):
        if isinstance(value, object):
            if hasattr(value, 'to_dict'):
                return value.to_dict()
            else:
                return str(value)
        else:
            return value

    def _is_internal(self, prop):
        return prop.startswith('_')
```

The `AsDictionaryMixin` class exposes a `.to_dict()` method that returns the representation of itself as a dictionary. The method is implemented as a [dict comprehension](#) that says, “Create a dictionary mapping `prop` to `value` for each item in `self.__dict__.items()` if the `prop` is not internal.”

Note: This is why we made the `role` and `payroll` attributes internal in the `Employee` class, because we don't want to represent them in the dictionary.

As you saw at the beginning, creating a class inherits some members from object, and one of those members is `__dict__`, which is basically a mapping of all the attributes in an object to their value.

You iterate through all the items in `__dict__` and filter out the ones that have a name that starts with an underscore using `.__is_internal()`.

`._represent()` checks the specified value. If the value is an object, then it looks to see if it also has a `.to_dict()` member and uses it to represent the object. Otherwise, it returns a string representation. If the value is not an object, then it simply returns the value.

You can modify the `Employee` class to support this mixin:

Python

```
# In employees.py

from representations import AsDictionaryMixin

class Employee(AsDictionaryMixin):
    def __init__(self, id, name, address, role, payroll):
        self.id = id
        self.name = name
        self.address = address
        self._role = role
        self._payroll = payroll

    def work(self, hours):
        duties = self._role.perform_duties(hours)
        print(f'Employee {self.id} - {self.name}:')
        print(f'- {duties}')
        print('')
        self._payroll.track_work(hours)

    def calculate_payroll(self):
        return self._payroll.calculate_payroll()
```

All you have to do is inherit the `AsDictionaryMixin` to support the functionality. It will be nice to support the same functionality in the `Address` class, so the `Employee.address` attribute is represented in the same way:

Python

```
# In contacts.py

from representations import AsDictionaryMixin

class Address(AsDictionaryMixin):
    def __init__(self, street, city, state, zipcode, street2=''):
        self.street = street
        self.street2 = street2
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def __str__(self):
        lines = [self.street]
        if self.street2:
            lines.append(self.street2)
        lines.append(f'{self.city}, {self.state} {self.zipcode}')
        return '\n'.join(lines)
```

You apply the mixin to the `Address` class to support the feature. Now, you can write a small program to test it:

Python

```
# In program.py

import json
from employees import EmployeeDatabase

def print_dict(d):
    print(json.dumps(d, indent=2))

for employee in EmployeeDatabase().employees:
    print_dict(employee.to_dict())
```

The program implements a `print_dict()` that converts the dictionary to a [JSON](#) string using indentation so the output looks better.

Then, it iterates through all the employees, printing the dictionary representation provided by `.to_dict()`. You can run the program to see its output:

Shell

```
$ python program.py

{
    "id": "1",
    "name": "Mary Poppins",
    "address": {
        "street": "121 Admin Rd.",
        "street2": "",
        "city": "Concord",
        "state": "NH",
        "zipcode": "03301"
    }
}
{
    "id": "2",
    "name": "John Smith",
    "address": {
        "street": "67 Paperwork Ave",
        "street2": "",
        "city": "Manchester",
        "state": "NH",
        "zipcode": "03101"
    }
}
{
    "id": "3",
    "name": "Kevin Bacon",
    "address": {
        "street": "15 Rose St",
        "street2": "Apt. B-1",
        "city": "Concord",
        "state": "NH",
        "zipcode": "03301"
    }
}
{
    "id": "4",
    "name": "Jane Doe",
    "address": {
        "street": "39 Sole St.",
        "street2": "",
        "city": "Concord",
        "state": "NH",
        "zipcode": "03301"
    }
}
{
    "id": "5",
    "name": "Robin Williams",
    "address": {
        "street": "99 Mountain Rd.",
        "street2": "",
        "city": "Concord",
        "state": "NH",
        "zipcode": "03301"
    }
}
```

You leveraged the implementation of `AsDictionaryMixin` in both `Employee` and `Address` classes even when they are not related. Because `AsDictionaryMixin` only provides behavior, it is easy to reuse with other classes without causing problems.

Composition to Model “Has A” Relationship

Composition models a **has a** relationship. With composition, a class `Composite` **has an** instance of class `Component` and can leverage its implementation. The `Component` class can be reused in other classes completely unrelated to the `Composite`.

In the composition example above, the `Employee` class **has an** `Address` object. `Address` implements all the functionality to handle addresses, and it can be reused by other classes.

Other classes like `Customer` or `Vendor` can reuse `Address` without being related to `Employee`. They can leverage the same implementation ensuring that addresses are handled consistently across the application.

A problem you may run into when using composition is that some of your classes may start growing by using multiple components. Your classes may require multiple parameters in the constructor just to pass in the components they are made of. This can make your classes hard to use.

A way to avoid the problem is by using the [Factory Method](#) to construct your objects. You did that with the composition example.

If you look at the implementation of the `EmployeeDatabase` class, you'll notice that it uses `._create_employee()` to construct an `Employee` object with the right parameters.

This design will work, but ideally, you should be able to construct an `Employee` object just by specifying an `id`, for example `employee = Employee(1)`.

The following changes might improve your design. You can start with the `productivity` module:

Python

```
# In productivity.py

class _ProductivitySystem:
    def __init__(self):
        self._roles = {
            'manager': ManagerRole,
            'secretary': SecretaryRole,
            'sales': SalesRole,
            'factory': FactoryRole,
        }

    def get_role(self, role_id):
        role_type = self._roles.get(role_id)
        if not role_type:
            raise ValueError('role_id')
        return role_type()

    def track(self, employees, hours):
        print('Tracking Employee Productivity')
        print('=====')
        for employee in employees:
            employee.work(hours)
        print('')

# Role classes implementation omitted

_productivity_system = _ProductivitySystem()

def get_role(role_id):
    return _productivity_system.get_role(role_id)

def track(employees, hours):
    _productivity_system.track(employees, hours)
```

First, you make the `_ProductivitySystem` class internal, and then provide a `_productivity_system` internal variable to the module. You are communicating to other developers that they should not create or use the `_ProductivitySystem` directly. Instead, you provide two functions, `get_role()` and `track()`, as the public interface to the module. This is what other modules should use.

What you are saying is that the `_ProductivitySystem` is a [Singleton](#), and there should only be one object created from it.

Now, you can do the same with the `hr` module:

Python

```
# In hr.py

class _PayrollSystem:
    def __init__(self):
        self._employee_policies = {
            1: SalaryPolicy(3000),
            2: SalaryPolicy(1500),
            3: CommissionPolicy(1000, 100),
            4: HourlyPolicy(15),
            5: HourlyPolicy(9)
        }

    def get_policy(self, employee_id):
        policy = self._employee_policies.get(employee_id)
        if not policy:
            return ValueError(employee_id)
        return policy

    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('=====')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            if employee.address:
                print('- Sent to:')
                print(employee.address)
            print()

# Policy classes implementation omitted

_payroll_system = _PayrollSystem()

def get_policy(employee_id):
    return _payroll_system.get_policy(employee_id)

def calculate_payroll(employees):
    _payroll_system.calculate_payroll(employees)
```

Again, you make the `_PayrollSystem` internal and provide a public interface to it. The application will use the public interface to get policies and calculate payroll.

You will now do the same with the contacts module:

Python

```
# In contacts.py

class _AddressBook:
    def __init__(self):
        self._employee_addresses = {
            1: Address('121 Admin Rd.', 'Concord', 'NH', '03301'),
            2: Address('67 Paperwork Ave', 'Manchester', 'NH', '03101'),
            3: Address('15 Rose St', 'Concord', 'NH', '03301', 'Apt. B-1'),
            4: Address('39 Sole St.', 'Concord', 'NH', '03301'),
            5: Address('99 Mountain Rd.', 'Concord', 'NH', '03301'),
        }

    def get_employee_address(self, employee_id):
        address = self._employee_addresses.get(employee_id)
        if not address:
            raise ValueError(employee_id)
        return address

# Implementation of Address class omitted

_address_book = _AddressBook()

def get_employee_address(employee_id):
    return _address_book.get_employee_address(employee_id)
```

You are basically saying that there should only be one `_AddressBook`, one `_PayrollSystem`, and one `_ProductivitySystem`. Again, this design pattern is called the [Singleton](#) design pattern, which comes in handy for classes from which there should only be one, single instance.

Now, you can work on the employees module. You will also make a Singleton out of the `_EmployeeDatabase`, but you will make some additional changes:

Python

```
# In employees.py

from productivity import get_role
from hr import get_policy
from contacts import get_employee_address
from representations import AsDictionaryMixin

class _EmployeeDatabase:
    def __init__(self):
        self._employees = {
            1: {
                'name': 'Mary Poppins',
                'role': 'manager'
            },
            2: {
                'name': 'John Smith',
                'role': 'secretary'
            },
            3: {
                'name': 'Kevin Bacon',
                'role': 'sales'
            },
            4: {
                'name': 'Jane Doe',
                'role': 'factory'
            },
            5: {
                'name': 'Robin Williams',
                'role': 'secretary'
            }
        }

    @property
    def employees(self):
        return [Employee(id_) for id_ in sorted(self._employees)]

    def get_employee_info(self, employee_id):
        info = self._employees.get(employee_id)
        if not info:
            raise ValueError(employee_id)
        return info

class Employee(AsDictionaryMixin):
    def __init__(self, id):
        self.id = id
        info = employee_database.get_employee_info(self.id)
        self.name = info.get('name')
        self.address = get_employee_address(self.id)
        self._role = get_role(info.get('role'))
        self._payroll = get_policy(self.id)

    def work(self, hours):
        duties = self._role.perform_duties(hours)
        print(f'Employee {self.id} - {self.name}:')
        print(f'- {duties}')
        print('')
        self._payroll.track_work(hours)

    def calculate_payroll(self):
        return self._payroll.calculate_payroll()

employee_database = _EmployeeDatabase()
```

You first import the relevant functions and classes from other modules. The `_EmployeeDatabase` is made internal, and at the bottom, you create a single instance. This instance is public and part of the interface because you will want to use it in the application.

You changed the `_EmployeeDatabase._employees` attribute to be a dictionary where the key is the employee `id` and the value is the employee information. You also exposed a `.get_employee_info()` method to return the information for the specified employee `employee_id`.

The `_EmployeeDatabase.employees` property now sorts the keys to return the employees sorted by their `id`. You replaced the method that constructed the `Employee` objects with calls to the `Employee` initializer directly.

The `Employee` class now is initialized with the `id` and uses the public functions exposed in the other modules to initialize its attributes.

You can now change the program to test the changes:

Python

```
# In program.py

import json

from hr import calculate_payroll
from productivity import track
from employees import employee_database, Employee

def print_dict(d):
    print(json.dumps(d, indent=2))

employees = employee_database.employees

track(employees, 40)
calculate_payroll(employees)

temp_secretary = Employee(5)
print('Temporary Secretary:')
print_dict(temp_secretary.to_dict())
```

You import the relevant functions from the `hr` and `productivity` modules, as well as the `employee_database` and `Employee` class. The program is cleaner because you exposed the required interface and encapsulated how objects are accessed.

Notice that you can now create an `Employee` object directly just using its `id`. You can run the program to see its output:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Employee 1 - Mary Poppins:
- screams and yells for 40 hours.

Employee 2 - John Smith:
- does paperwork for 40 hours.

Employee 3 - Kevin Bacon:
- expends 40 hours on the phone.

Employee 4 - Jane Doe:
- manufactures gadgets for 40 hours.

Employee 5 - Robin Williams:
- does paperwork for 40 hours.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000
- Sent to:
121 Admin Rd.
Concord, NH 03301

Payroll for: 2 - John Smith
- Check amount: 1500
- Sent to:
67 Paperwork Ave
Manchester, NH 03101

Payroll for: 3 - Kevin Bacon
- Check amount: 1800.0
- Sent to:
15 Rose St
Apt. B-1
Concord, NH 03301

Payroll for: 4 - Jane Doe
- Check amount: 600
- Sent to:
39 Sole St.
Concord, NH 03301

Payroll for: 5 - Robin Williams
- Check amount: 360
- Sent to:
99 Mountain Rd.
Concord, NH 03301

Temporary Secretary:
{
    "id": "5",
    "name": "Robin Williams",
    "address": {
        "street": "99 Mountain Rd.",
        "street2": "",
        "city": "Concord",
        "state": "NH",
        "zipcode": "03301"
    }
}
```

The program works the same as before, but now you can see that a single `Employee` object can be created from its `id` and display its dictionary representation.

Take a closer look at the `Employee` class:

Python

```
# In employees.py

class Employee(AsDictionaryMixin):
    def __init__(self, id):
        self.id = id
        info = employee_database.get_employee_info(self.id)
        self.name = info.get('name')
        self.address = get_employee_address(self.id)
        self._role = get_role(info.get('role'))
        self._payroll = get_policy(self.id)

    def work(self, hours):
        duties = self._role.perform_duties(hours)
        print(f'Employee {self.id} - {self.name}:')
        print(f'- {duties}')
        print('')
        self._payroll.track_work(hours)

    def calculate_payroll(self):
        return self._payroll.calculate_payroll()
```

The `Employee` class is a composite that contains multiple objects providing different functionality. It contains an `Address` that implements all the functionality related to where the employee lives.

`Employee` also contains a productivity role provided by the productivity module, and a payroll policy provided by the `hr` module. These two objects provide implementations that are leveraged by the `Employee` class to track work in the `.work()` method and to calculate the payroll in the `.calculate_payroll()` method.

You are using composition in two different ways. The `Address` class provides additional data to `Employee` where the role and payroll objects provide additional behavior.

Still, the relationship between `Employee` and those objects is loosely coupled, which provides some interesting capabilities that you'll see in the next section.

Composition to Change Run-Time Behavior

Inheritance, as opposed to composition, is a tightly couple relationship. With inheritance, there is only one way to change and customize behavior. Method overriding is the only way to customize the behavior of a base class. This creates rigid designs that are difficult to change.

Composition, on the other hand, provides a loosely coupled relationship that enables flexible designs and can be used to change behavior at run-time.

Imagine you need to support a long-term disability (LTD) policy when calculating payroll. The policy states that an employee on LTD should be paid 60% of their weekly salary assuming 40 hours of work.

With an inheritance design, this can be a very difficult requirement to support. Adding it to the composition example is a lot easier. Let's start by adding the policy class:

Python

```
# In hr.py

class LTDPolicy:
    def __init__(self):
        self._base_policy = None

    def track_work(self, hours):
        self._check_base_policy()
        return self._base_policy.track_work(hours)

    def calculate_payroll(self):
        self._check_base_policy()
        base_salary = self._base_policy.calculate_payroll()
        return base_salary * 0.6

    def apply_to_policy(self, base_policy):
        self._base_policy = base_policy

    def _check_base_policy(self):
        if not self._base_policy:
            raise RuntimeError('Base policy missing')
```

Notice that `LTDPolicy` doesn't inherit `PayrollPolicy`, but implements the same interface. This is because the implementation is completely different, so we don't want to inherit any of the `PayrollPolicy` implementation.

The `LTDPolicy` initializes `_base_policy` to `None`, and provides an internal `._check_base_policy()` method that raises an exception if the `._base_policy` has not been applied. Then, it provides a `.apply_to_policy()` method to assign the `_base_policy`.

The public interface first checks that the `_base_policy` has been applied, and then implements the functionality in terms of that base policy. The `.track_work()` method just delegates to the base policy, and `.calculate_payroll()` uses it to calculate the `base_salary` and then return the 60%.

You can now make a small change to the `Employee` class:

Python

```
# In employees.py

class Employee(AsDictionaryMixin):
    def __init__(self, id):
        self.id = id
        info = employee_database.get_employee_info(self.id)
        self.name = info.get('name')
        self.address = get_employee_address(self.id)
        self._role = get_role(info.get('role'))
        self._payroll = get_policy(self.id)

    def work(self, hours):
        duties = self._role.perform_duties(hours)
        print(f'Employee {self.id} - {self.name}:')
        print(f'- {duties}')
        print('')
        self._payroll.track_work(hours)

    def calculate_payroll(self):
        return self._payroll.calculate_payroll()

    def apply_payroll_policy(self, new_policy):
        new_policy.apply_to_policy(self._payroll)
        self._payroll = new_policy
```

You added an `.apply_payroll_policy()` method that applies the existing payroll policy to the new policy and then substitutes it. You can now modify the program to apply the policy to an `Employee` object:

Python

```
# In program.py

from hr import calculate_payroll, LTDPolicy
from productivity import track
from employees import employee_database

employees = employee_database.employees

sales_employee = employees[2]
ltd_policy = LTDPolicy()
sales_employee.apply_payroll_policy(ltd_policy)

track(employees, 40)
calculate_payroll(employees)
```

The program accesses `sales_employee`, which is located at index 2, creates the `LTDPolicy` object, and applies the policy to the employee. When `.calculate_payroll()` is called, the change is reflected. You can run the program to evaluate the output:

Shell

```
$ python program.py

Tracking Employee Productivity
=====
Employee 1 - Mary Poppins:
- screams and yells for 40 hours.

Employee 2 - John Smith:
- Does paperwork for 40 hours.

Employee 3 - Kevin Bacon:
- Expend 40 hours on the phone.

Employee 4 - Jane Doe:
- Manufactures gadgets for 40 hours.

Employee 5 - Robin Williams:
- Does paperwork for 40 hours.

Calculating Payroll
=====
Payroll for: 1 - Mary Poppins
- Check amount: 3000
- Sent to:
121 Admin Rd.
Concord, NH 03301

Payroll for: 2 - John Smith
- Check amount: 1500
- Sent to:
67 Paperwork Ave
Manchester, NH 03101

Payroll for: 3 - Kevin Bacon
- Check amount: 1080.0
- Sent to:
15 Rose St
Apt. B-1
Concord, NH 03301

Payroll for: 4 - Jane Doe
- Check amount: 600
- Sent to:
39 Sole St.
Concord, NH 03301

Payroll for: 5 - Robin Williams
- Check amount: 360
- Sent to:
99 Mountain Rd.
Concord, NH 03301
```

The check amount for employee Kevin Bacon, who is the sales employee, is now for \$1080 instead of \$1800. That's because the `LTDPolicy` has been applied to the salary.

As you can see, you were able to support the changes just by adding a new policy and modifying a couple interfaces. This is the kind of flexibility that policy design based on composition gives you.

Choosing Between Inheritance and Composition in Python

Python, as an object oriented programming language, supports both inheritance and composition. You saw that inheritance is best used to model an **is a** relationship, whereas composition models a **has a** relationship.

Sometimes, it's hard to see what the relationship between two classes should be, but you can follow these guidelines:

- **Use inheritance over composition in Python** to model a clear **is a** relationship. First, justify the relationship between the derived class and its base. Then, reverse the relationship and try to justify it. If you can justify the relationship in both directions, then you should not use inheritance between them.

- **Use inheritance over composition in Python** to leverage both the interface and implementation of the base class.
- **Use inheritance over composition in Python** to provide **mixin** features to several unrelated classes when there is only one implementation of that feature.
- **Use composition over inheritance in Python** to model a **has a** relationship that leverages the implementation of the component class.
- **Use composition over inheritance in Python** to create components that can be reused by multiple classes in your Python applications.
- **Use composition over inheritance in Python** to implement groups of behaviors and policies that can be applied interchangeably to other classes to customize their behavior.
- **Use composition over inheritance in Python** to enable run-time behavior changes without affecting existing classes.

Conclusion

You explored **inheritance and composition in Python**. You learned about the type of relationships that inheritance and composition create. You also went through a series of exercises to understand how inheritance and composition are implemented in Python.

In this article, you learned how to:

- Use inheritance to express an **is a** relationship between two classes
- Evaluate if inheritance is the right relationship
- Use multiple inheritance in Python and evaluate Python's MRO to troubleshoot multiple inheritance problems
- Extend classes with mixins and reuse their implementation
- Use composition to express a **has a** relationship between two classes
- Provide flexible designs using composition
- Reuse existing code through policy design based on composition

Recommended Reading

Here are some books and articles that further explore object oriented design and can be useful to help you understand the correct use of inheritance and composition in Python or other languages:

- [Design Patterns: Elements of Reusable Object-Oriented Software](#)
- [Head First Design Patterns: A Brain-Friendly Guide](#)
- [Clean Code: A Handbook of Agile Software Craftsmanship](#)
- [SOLID Principles](#)
- [Liskov Substitution Principle](#)

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Inheritance and Composition: A Python OOP Guide](#)

About Isaac Rodriguez

Hi, I'm Isaac. I build, lead, and mentor software development teams, and for the past few years I've been focusing on cloud services and back-end applications using Python among other languages. Love to hear from you here at Real Python.

[» More about Isaac](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Alex](#)

[Aldren](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [python](#)

Recommended Video Course: [Inheritance and Composition: A Python OOP Guide](#)



Real Python

Thinking Recursively in Python

by Abhirag Awasthi 25 Comments Intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Dear Pythonic Santa Claus...](#)
- [Recursive Functions in Python](#)
- [Maintaining State](#)
- [Recursive Data Structures in Python](#)
- [Naive Recursion is Naive](#)
- [Pesky Details](#)
- [Fin](#)
- [References](#)

A Python Best Practices Handbook

python-guide.org



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Thinking Recursively in Python](#)

“Of all ideas I have introduced to children, recursion stands out as the one idea that is particularly able to evoke an excited response.”

— Seymour Papert, *Mindstorms*

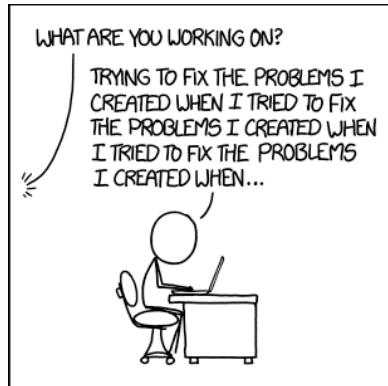


Image: xkcd.com

Problems (in life and also in computer science) can often seem big and scary. But if we keep chipping away at them, more often than not we can break them down into smaller chunks trivial enough to solve. This is the essence of thinking recursively, and my aim in this article is to provide you, my dear reader, with the conceptual tools necessary to approach problems from this recursive point of view.

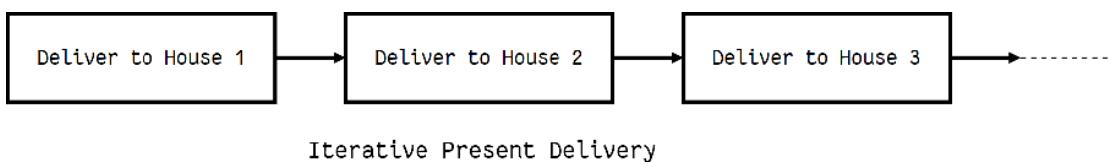
Together, we'll learn how to work with recursion in our Python programs by mastering concepts such as recursive functions and recursive data structures. We'll also talk about maintaining state during recursion and avoiding recomputation by [caching results](#). This is going to be a lot of fun. Onwards and upwards!

[Remove ads](#)

Dear Pythonic Santa Claus...

I realize that as fellow Pythonistas we are all consenting adults here, but children seem to grok the beauty of recursion better. So let's *not* be adults here for a moment and talk about how we can use recursion to help Santa Claus.

Have you ever wondered how Christmas presents are delivered? I sure have, and I believe Santa Claus has a list of houses he loops through. He goes to a house, drops off the presents, eats the cookies and milk, and moves on to the next house on the list. Since this algorithm for delivering presents is based on an explicit loop construction, it is called an iterative algorithm.



The algorithm for iterative present delivery implemented in Python:

Python

```
houses = ["Eric's house", "Kenny's house", "Kyle's house", "Stan's house"]

def deliver_presents_iteratively():
    for house in houses:
        print("Delivering presents to", house)
```

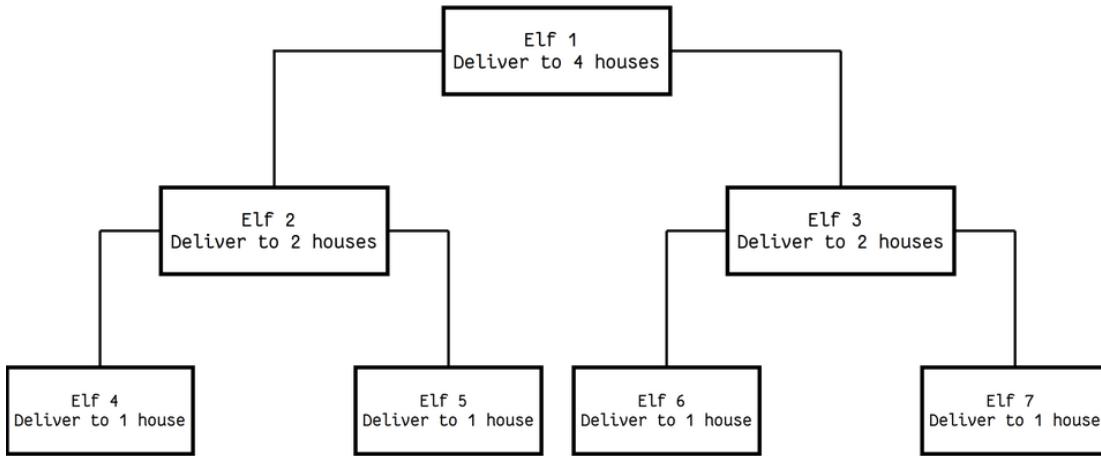
Python

>>>

```
>>> deliver_presents_iteratively()
Delivering presents to Eric's house
Delivering presents to Kenny's house
Delivering presents to Kyle's house
Delivering presents to Stan's house
```

But I feel for Santa. At his age, he shouldn't have to deliver all the presents by himself. I propose an algorithm with which he can divide the work of delivering presents among his elves:

1. Appoint an elf and give all the work to him
2. Assign titles and responsibilities to the elves based on the number of houses for which they are responsible:
 - > 1 He is a manager and can appoint two elves and divide his work among them
 - = 1 He is a worker and has to deliver the presents to the house assigned to him



Recursive Present Delivery

This is the typical structure of a recursive algorithm. If the current problem represents a simple case, solve it. If not, divide it into subproblems and apply the same strategy to them.

The algorithm for recursive present delivery implemented in Python:

Python

```
houses = ["Eric's house", "Kenny's house", "Kyle's house", "Stan's house"]

# Each function call represents an elf doing his work
def deliver_presents_recursively(houses):
    # Worker elf doing his work
    if len(houses) == 1:
        house = houses[0]
        print("Delivering presents to", house)

    # Manager elf doing his work
    else:
        mid = len(houses) // 2
        first_half = houses[:mid]
        second_half = houses[mid:]

        # Divides his work among two elves
        deliver_presents_recursively(first_half)
        deliver_presents_recursively(second_half)
```

Python

>>>

```
>>> deliver_presents_recursively(houses)
Delivering presents to Eric's house
Delivering presents to Kenny's house
Delivering presents to Kyle's house
Delivering presents to Stan's house
```

Recursive Functions in Python

Now that we have some intuition about recursion, let's introduce the formal definition of a recursive function. A recursive function is a function defined in terms of itself via self-referential expressions.

This means that the function will continue to call itself and repeat its behavior until some condition is met to return a result. All recursive functions share a common structure made up of two parts: base case and recursive case.

To demonstrate this structure, let's write a recursive function for calculating $n!$:

1. Decompose the original problem into simpler instances of the same problem. This is the recursive case:

```
n! = n x (n-1) x (n-2) x (n-3) ... x 3 x 2 x 1
n! = n x (n-1)!
```

2. As the large problem is broken down into successively less complex ones, those subproblems must eventually become so simple that they can be solved without further subdivision. This is the base case:

```
n! = n x (n-1)!
n! = n x (n-1) x (n-2)!
n! = n x (n-1) x (n-2) x (n-3)!
.
.
.
n! = n x (n-1) x (n-2) x (n-3) ... x 3!
n! = n x (n-1) x (n-2) x (n-3) ... x 3 x 2!
n! = n x (n-1) x (n-2) x (n-3) ... x 3 x 2 x 1!
```

Here, $1!$ is our base case, and it equals 1.

Recursive function for calculating $n!$ implemented in Python:

Python

```
def factorial_recursive(n):
    # Base case: 1! = 1
    if n == 1:
        return 1

    # Recursive case: n! = n * (n-1)!
    else:
        return n * factorial_recursive(n-1)
```

Python

>>>

```
>>> factorial_recursive(5)
120
```

Behind the scenes, each recursive call adds a stack frame (containing its execution context) to the call stack until we reach the base case. Then, the stack begins to unwind as each call returns its results:

[Remove ads](#)

Maintaining State

When dealing with recursive functions, keep in mind that each recursive call has its own execution context, so to maintain state during recursion you have to either:

- Thread the state through each recursive call so that the current state is part of the current call's execution context
- Keep the state in global scope

A demonstration should make things clearer. Let's calculate $1 + 2 + 3 \dots + 10$ using recursion. The state that we have to maintain is (*current number we are adding, accumulated sum till now*).

Here's how you do that by threading it through each recursive call (i.e. passing the updated current state to each recursive call as arguments):

Python

```
def sum_recursive(current_number, accumulated_sum):
    # Base case
    # Return the final state
    if current_number == 11:
        return accumulated_sum

    # Recursive case
    # Thread the state through the recursive call
    else:
        return sum_recursive(current_number + 1, accumulated_sum + current_number)
```

Python

>>>

```
# Pass the initial state
>>> sum_recursive(1, 0)
55
```

Here's how you maintain the state by keeping it in global [scope](#):

Python

```
# Global mutable state
current_number = 1
accumulated_sum = 0

def sum_recursive():
    global current_number
    global accumulated_sum
    # Base case
    if current_number == 11:
        return accumulated_sum
    # Recursive case
    else:
        accumulated_sum = accumulated_sum + current_number
        current_number = current_number + 1
        return sum_recursive()
```

Python

>>>

```
>>> sum_recursive()
55
```

I prefer threading the state through each recursive call because I find global mutable state to be evil, but that's a discussion for a later time.

Recursive Data Structures in Python

A data structure is recursive if it can be defined in terms of a smaller version of itself. A list is an example of a recursive data structure. Let me demonstrate. Assume that you have only an empty list at your disposal, and the only operation you can perform on it is this:

Python

```
# Return a new list that is the result of
# adding element to the head (i.e. front) of input_list
def attach_head(element, input_list):
    return [element] + input_list
```

Using the empty list and the attach_head operation, you can generate any list. For example, let's generate [1, 46, -31, "hello"]:

Python

```
attach_head(1,                               # Will return [1, 46, -31, "hello"]
            attach_head(46,                 # Will return [46, -31, "hello"]
                        attach_head(-31,      # Will return [-31, "hello"]
                                      attach_head("hello", []))) # Will return ["hello"]
```

```
[1, 46, -31, 'hello']
```

1. Starting with an empty list, you can generate any list by recursively applying the attach_head function, and thus the list data structure can be defined recursively as:

```
+---- attach_head(element, smaller list)
list = +
+---- empty list
```

2. Recursion can also be seen as self-referential function composition. We apply a function to an argument, then pass that result on as an argument to a second application of the same function, and so on. Repeatedly composing attach_head with itself is the same as attach_head calling itself repeatedly.

List is not the only recursive data structure. Other examples include set, tree, dictionary, etc.

Recursive data structures and recursive functions go together like bread and butter. The recursive function's structure can often be modeled after the definition of the recursive data structure it takes as an input. Let me demonstrate this by calculating the sum of all the elements of a list recursively:

Python

```
def list_sum_recursive(input_list):
    # Base case
    if input_list == []:
        return 0

    # Recursive case
    # Decompose the original problem into simpler instances of the same problem
    # by making use of the fact that the input is a recursive data structure
    # and can be defined in terms of a smaller version of itself
    else:
        head = input_list[0]
        smaller_list = input_list[1:]
        return head + list_sum_recursive(smaller_list)
```

Python

>>>

```
>>> list_sum_recursive([1, 2, 3])
6
```

[Remove ads](#)

Naive Recursion is Naive

The Fibonacci numbers were originally defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Fibonacci surmised that the number of pairs of rabbits born in a given year is equal to the number of pairs of rabbits born in each of the two previous years, starting from one pair of rabbits in the first year.

To count the number of rabbits born in the nth year, he defined the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

The base cases are:

$$F_0 = 0 \text{ and } F_1 = 1$$

Let's write a recursive function to compute the nth Fibonacci number:

Python

```
def fibonacci_recursive(n):
    print("Calculating F", "(", n, ")", sep="", end=", ")
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Python

>>>

```
>>> fibonacci_recursive(5)
Calculating F(5), Calculating F(4), Calculating F(3), Calculating F(2), Calculating F(1),
Calculating F(0), Calculating F(1), Calculating F(2), Calculating F(1), Calculating F(0),
Calculating F(3), Calculating F(2), Calculating F(1), Calculating F(0), Calculating F(1),
5
```

Naively following the recursive definition of the nth Fibonacci number was rather inefficient. As you can see from the output above, we are unnecessarily recomputing values. Let's try to improve `fibonacci_recursive` by caching the results of each Fibonacci computation F_k :

Python

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci_recursive(n):
    print("Calculating F", "(", n, ")\"", sep="", end=", ")
    # Base case
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Recursive case
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Python

>>>

```
>>> fibonacci_recursive(5)
Calculating F(5), Calculating F(4), Calculating F(3), Calculating F(2), Calculating F(1), Calculatin
5
```

`lru_cache` is a [decorator](#) that caches the results. Thus, we avoid recomputation by explicitly checking for the value before trying to compute it. One thing to keep in mind about `lru_cache` is that since it uses a dictionary to cache results, the positional and keyword arguments (which serve as keys in that dictionary) to the function must be hashable.

Pesky Details

Python doesn't have support for [tail-call elimination](#). As a result, you can cause a stack overflow if you end up using more stack frames than the [default call stack depth](#):

Python

>>>

```
>>> import sys
>>> sys.getrecursionlimit()
3000
```

Keep this limitation in mind if you have a program that requires deep recursion.

Also, Python's mutable data structures don't support structural sharing, so treating them like [immutable data structures](#) is going to negatively affect your space and [GC \(garbage collection\)](#) efficiency because you are going to end up unnecessarily copying a lot of mutable objects. For example, I have used this pattern to decompose lists and recurse over them:

Python

>>>

```
>>> input_list = [1, 2, 3]
>>> head = input_list[0]
>>> tail = input_list[1:]
>>> print("head --", head)
head -- 1
>>> print("tail --", tail)
tail -- [2, 3]
```

I did that to simplify things for the sake of clarity. Keep in mind that `tail` is being created by copying. Recursively doing that over large lists can negatively affect your space and GC efficiency.

 [Remove ads](#)

Fin

I was once asked to explain recursion in an interview. I took a sheet of paper and wrote Please turn over on both sides. The interviewer didn't get the joke, but now that you have read this article, hopefully you do ☺ Happy Pythoning!

References

1. [Thinking Recursively](#)
2. [The Little Schemer](#)
3. [Concepts, Techniques, and Models of Computer Programming](#)
4. [The Algorithm Design Manual](#)
5. [Haskell Programming from First Principles](#)

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Thinking Recursively in Python](#)

About Abhirag Awasthi

Programmer/Musician, constantly trying to create something worthwhile, getting better at my craft in the process

[» More about Abhirag](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

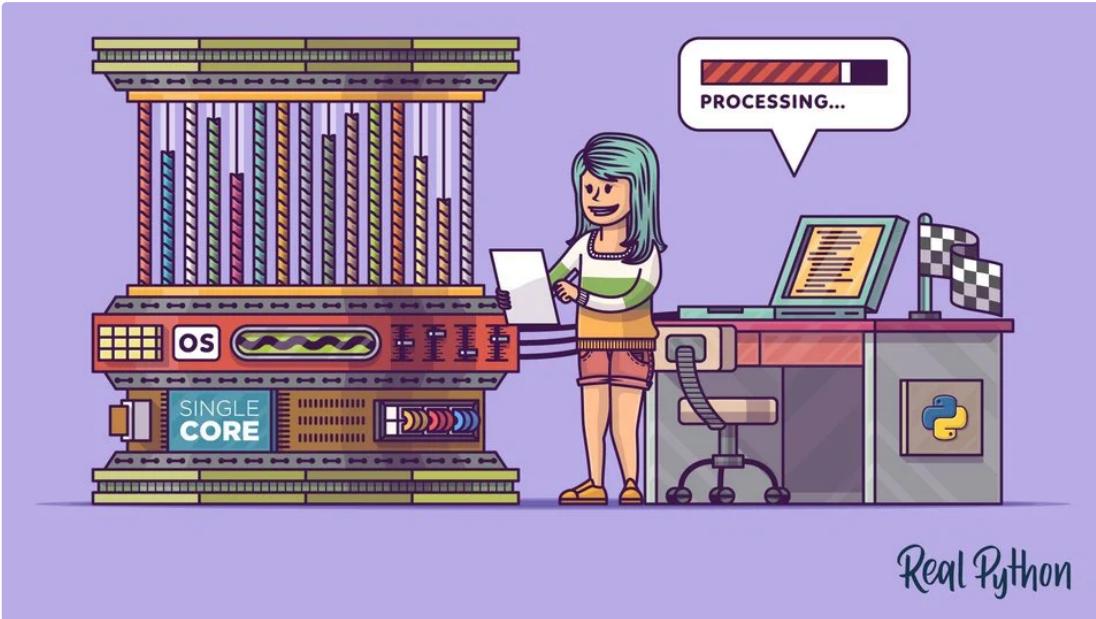
[Aldren](#)

[Dan](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Real Python

An Intro to Threading in Python

by Jim Anderson 56 Comments best-practices intermediate

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Is a Thread?](#)
- [Starting a Thread](#)
 - [Daemon Threads](#)
 - [join\(\) a Thread](#)
- [Working With Many Threads](#)
- [Using a ThreadPoolExecutor](#)
- [Race Conditions](#)
 - [One Thread](#)
 - [Two Threads](#)
 - [Why This Isn't a Silly Example](#)
- [Basic Synchronization Using Lock](#)
- [Deadlock](#)
- [Producer-Consumer Threading](#)
 - [Producer-Consumer Using Lock](#)
 - [Producer-Consumer Using Queue](#)
- [Threading Objects](#)
 - [Semaphore](#)
 - [Timer](#)
 - [Barrier](#)
- [Conclusion: Threading in Python](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Threading in Python](#)

Python threading allows you to have different parts of your program run concurrently and can simplify your design. If you've got some experience in Python and want to speed up your program using threads, then this tutorial is for

you!

In this article, you'll learn:

- What threads are
- How to create threads and wait for them to finish
- How to use a ThreadPoolExecutor
- How to avoid race conditions
- How to use the common tools that Python threading provides

This article assumes you've got the Python basics down pat and that you're using at least version 3.6 to run the examples. If you need a refresher, you can start with the [Python Learning Paths](#) and get up to speed.

If you're not sure if you want to use Python threading, asyncio, or multiprocessing, then you can check out [Speed Up Your Python Program With Concurrency](#).

All of the sources used in this tutorial are available to you in the [Real Python GitHub repo](#).

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

 **Take the Quiz:** Test your knowledge with our interactive "Python Threading" quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

What Is a Thread?

A thread is a separate flow of execution. This means that your program will have two things happening at once. But for most Python 3 implementations the different threads do not actually execute at the same time: they merely appear to.

It's tempting to think of threading as having two (or more) different processors running on your program, each one doing an independent task at the same time. That's almost right. The threads may be running on different processors, but they will only be running one at a time.

Getting multiple tasks running simultaneously requires a non-standard implementation of Python, writing some of your code in a different language, or using multiprocessing which comes with some extra overhead.

Because of the way CPython implementation of Python works, threading may not speed up all tasks. This is due to interactions with the [GIL](#) that essentially limit one Python thread to run at a time.

Tasks that spend much of their time waiting for external events are generally good candidates for threading. Problems that require heavy CPU computation and spend little time waiting for external events might not run faster at all.

This is true for code written in Python and running on the standard CPython implementation. If your threads are written in C they have the ability to release the GIL and run concurrently. If you are running on a different Python implementation, check with the documentation too see how it handles threads.

If you are running a standard Python implementation, writing in only Python, and have a CPU-bound problem, you should check out the `multiprocessing` module instead.

Architecting your program to use threading can also provide gains in design clarity. Most of the examples you'll learn about in this tutorial are not necessarily going to run faster because they use threads. Using threading in them helps to make the design cleaner and easier to reason about.

So, let's stop talking about threading and start using it!

Starting a Thread

Now that you've got an idea of what a thread is, let's learn how to make one. The Python standard library provides [threading](#), which contains most of the primitives you'll see in this article. Thread, in this module, nicely encapsulates threads, providing a clean interface to work with them.

To start a separate thread, you create a Thread instance and then tell it to `.start()`:

Python

```
1 import logging
2 import threading
3 import time
4
5 def thread_function(name):
6     logging.info("Thread %s: starting", name)
7     time.sleep(2)
8     logging.info("Thread %s: finishing", name)
9
10 if __name__ == "__main__":
11     format = "%(asctime)s: %(message)s"
12     logging.basicConfig(format=format, level=logging.INFO,
13                         datefmt="%H:%M:%S")
14
15 logging.info("Main    : before creating thread")
16 x = threading.Thread(target=thread_function, args=(1,))
17 logging.info("Main    : before running thread")
18 x.start()
19 logging.info("Main    : wait for the thread to finish")
20 # x.join()
21 logging.info("Main    : all done")
```

If you look around the logging statements, you can see that the `main` section is creating and starting the thread:

Python

```
x = threading.Thread(target=thread_function, args=(1,))
x.start()
```

When you create a Thread, you pass it a function and a list containing the arguments to that function. In this case, you're telling the Thread to run `thread_function()` and to pass it `1` as an argument.

For this article, you'll use sequential integers as names for your threads. There is `threading.get_ident()`, which returns a unique name for each thread, but these are usually neither short nor easily readable.

`thread_function()` itself doesn't do much. It simply logs some messages with a `time.sleep()` in between them.

When you run this program as it is (with line twenty commented out), the output will look like this:

Shell

```
$ ./single_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
Thread 1: finishing
```

You'll notice that the Thread finished after the Main section of your code did. You'll come back to why that is and talk about the mysterious line twenty in the next section.

Daemon Threads

In computer science, a [daemon](#) is a process that runs in the background.

Python threading has a more specific meaning for daemon. A daemon thread will shut down immediately when the program exits. One way to think about these definitions is to consider the daemon thread a thread that runs in the

background of the application, and is terminated when the application exits.

background without worrying about shutting it down.

If a program is running threads that are not daemons, then the program will wait for those threads to complete before it terminates. Threads that *are* daemons, however, are just killed wherever they are when the program is exiting.

Let's look a little more closely at the output of your program above. The last two lines are the interesting bit. When you run the program, you'll notice that there is a pause (of about 2 seconds) after `__main__` has printed its `all done` message and before the thread is finished.

This pause is Python waiting for the non-daemonic thread to complete. When your Python program ends, part of the shutdown process is to clean up the threading routine.

If you look at the [source for Python threading](#), you'll see that `threading._shutdown()` walks through all of the running threads and calls `.join()` on every one that does not have the `daemon` flag set.

So your program waits to exit because the thread itself is waiting in a sleep. As soon as it has completed and printed the message, `.join()` will return and the program can exit.

Frequently, this behavior is what you want, but there are other options available to us. Let's first repeat the program with a daemon thread. You do that by changing how you construct the Thread, adding the `daemon=True` flag:

Python

```
x = threading.Thread(target=thread_function, args=(1,), daemon=True)
```

When you run the program now, you should see this output:

Shell

```
$ ./daemon_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
```

The difference here is that the final line of the output is missing. `thread_function()` did not get a chance to complete. It was a daemon thread, so when `__main__` reached the end of its code and the program wanted to finish, the daemon was killed.

join() a Thread

Daemon threads are handy, but what about when you want to wait for a thread to stop? What about when you want to do that and not exit your program? Now let's go back to your original program and look at that commented out line twenty:

Python

```
# x.join()
```

To tell one thread to wait for another thread to finish, you call `.join()`. If you uncomment that line, the main thread will pause and wait for the thread `x` to complete running.

Did you test this on the code with the daemon thread or the regular thread? It turns out that it doesn't matter. If you `.join()` a thread, that statement will wait until either kind of thread is finished.

Working With Many Threads

The example code so far has only been working with two threads: the main thread and one you started with the `threading.Thread` object.

Frequently, you'll want to start a number of threads and have them do interesting work. Let's start by looking at the harder way of doing that, and then you'll move on to an easier method.

The harder way of starting multiple threads is the one you already know:

Python

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    threads = list()
    for index in range(3):
        logging.info("Main    : create and start thread %d.", index)
        x = threading.Thread(target=thread_function, args=(index,))
        threads.append(x)
        x.start()

    for index, thread in enumerate(threads):
        logging.info("Main    : before joining thread %d.", index)
        thread.join()
        logging.info("Main    : thread %d done", index)
```

This code uses the same mechanism you saw above to start a thread, create a Thread object, and then call `.start()`. The program keeps a list of Thread objects so that it can then wait for them later using `.join()`.

Running this code multiple times will likely produce some interesting results. Here's an example output from my machine:

Shell

```
$ ./multiple_threads.py
Main    : create and start thread 0.
Thread 0: starting
Main    : create and start thread 1.
Thread 1: starting
Main    : create and start thread 2.
Thread 2: starting
Main    : before joining thread 0.
Thread 2: finishing
Thread 1: finishing
Thread 0: finishing
Main    : thread 0 done
Main    : before joining thread 1.
Main    : thread 1 done
Main    : before joining thread 2.
Main    : thread 2 done
```

If you walk through the output carefully, you'll see all three threads getting started in the order you might expect.

If you walk through the output carefully, you'll see all three threads getting started in the order you might expect, but in this case they finish in the opposite order! Multiple runs will produce different orderings. Look for the Thread x: finishing message to tell you when each thread is done.

The order in which threads are run is determined by the operating system and can be quite hard to predict. It may (and likely will) vary from run to run, so you need to be aware of that when you design algorithms that use threading.

Fortunately, Python gives you several primitives that you'll look at later to help coordinate threads and get them running together. Before that, let's look at how to make managing a group of threads a bit easier.

Using a ThreadPoolExecutor

There's an easier way to start up a group of threads than the one you saw above. It's called a `ThreadPoolExecutor`, and it's part of the standard library in [concurrent.futures](#) (as of Python 3.2).

The easiest way to create it is as a context manager, using the `with` statement to manage the creation and destruction of the pool.

Here's the `__main__` from the last example rewritten to use a `ThreadPoolExecutor`:

Python

```
import concurrent.futures

# [rest of code]

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        executor.map(thread_function, range(3))
```

The code creates a `ThreadPoolExecutor` as a context manager, telling it how many worker threads it wants in the pool. It then uses `.map()` to step through an iterable of things, in your case `range(3)`, passing each one to a thread in the pool.

The end of the `with` block causes the `ThreadPoolExecutor` to do a `.join()` on each of the threads in the pool. It is *strongly* recommended that you use `ThreadPoolExecutor` as a context manager when you can so that you never forget to `.join()` the threads.

Note: Using a `ThreadPoolExecutor` can cause some confusing errors.

For example, if you call a function that takes no parameters, but you pass it parameters in `.map()`, the thread will throw an exception.

Unfortunately, `ThreadPoolExecutor` will hide that exception, and (in the case above) the program terminates with no output. This can be quite confusing to debug at first.

Running your corrected example code will produce output that looks like this:

Shell

```
$ ./executor.py
Thread 0: starting
Thread 1: starting
Thread 2: starting
Thread 1: finishing
Thread 0: finishing
Thread 2: finishing
```

Again, notice how Thread 1 finished before Thread 0. The scheduling of threads is done by the operating system and does not follow a plan that's easy to figure out.

Race Conditions

Before you move on to some of the other features tucked away in Python threading, let's talk a bit about one of the more difficult issues you'll run into when writing threaded programs: [race conditions](#).

Once you've seen what a race condition is and looked at one happening, you'll move on to some of the primitives provided by the standard library to prevent race conditions from happening.

Race conditions can occur when two or more threads access a shared piece of data or resource. In this example, you're going to create a large race condition that happens every time, but be aware that most race conditions are not this obvious. Frequently, they only occur rarely, and they can produce confusing results. As you can imagine, this makes them quite difficult to debug.

Fortunately, this race condition will happen every time, and you'll walk through it in detail to explain what is happening.

For this example, you're going to write a class that updates a database. Okay, you're not really going to have a database: you're just going to fake it, because that's not the point of this article.

Your FakeDatabase will have `__init__()` and `update()` methods:

Python

```
class FakeDatabase:
    def __init__(self):
        self.value = 0

    def update(self, name):
        logging.info("Thread %s: starting update", name)
        local_copy = self.value
        local_copy += 1
        time.sleep(0.1)
        self.value = local_copy
        logging.info("Thread %s: finishing update", name)
```

FakeDatabase is keeping track of a single number: `.value`. This is going to be the shared data on which you'll see the race condition.

`__init__()` simply initializes `.value` to zero. So far, so good.

`update()` looks a little strange. It's simulating reading a value from a database, doing some computation on it, and then writing a new value back to the database.

In this case, reading from the database just means copying `.value` to a local variable. The computation is just to add one to the value and then `.sleep()` for a little bit. Finally, it writes the value back by copying the local value back to `.value`.

Here's how you'll use this FakeDatabase:

Python

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    database = FakeDatabase()
    logging.info("Testing update. Starting value is %d.", database.value)
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        for index in range(2):
            executor.submit(database.update, index)
    logging.info("Testing update. Ending value is %d.", database.value)
```

The program creates a `ThreadPoolExecutor` with two threads and then calls `.submit()` on each of them, telling them to run `database.update()`.

`.submit()` has a signature that allows both positional and named arguments to be passed to the function running in the thread:

Python

```
.submit(function, *args, **kwargs)
```

In the usage above, `index` is passed as the first and only positional argument to `database.update()`. You'll see later in this article where you can pass multiple arguments in a similar manner.

Since each thread runs `.update()`, and `.update()` adds one to `.value`, you might expect `database.value` to be 2 when it's printed out at the end. But you wouldn't be looking at this example if that was the case. If you run the above code, the output looks like this:

Shell

```
$ ./racecond.py
Testing unlocked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing unlocked update. Ending value is 1.
```

You might have expected that to happen, but let's look at the details of what's really going on here, as that will make the solution to this problem easier to understand.

One Thread

Before you dive into this issue with two threads, let's step back and talk a bit about some details of how threads work.

You won't be diving into all of the details here, as that's not important at this level. We'll also be simplifying a few things in a way that won't be technically accurate but will give you the right idea of what is happening.

When you tell your `ThreadPoolExecutor` to run each thread, you tell it which function to run and what parameters to pass to it: `executor.submit(database.update, index)`.

The result of this is that each of the threads in the pool will call `database.update(index)`. Note that `database` is a reference to the one `FakeDatabase` object created in `__main__`. Calling `.update()` on that object calls an [instance method](#) on that object.

Each thread is going to have a reference to the same `FakeDatabase` object, `database`. Each thread will also have a unique value, `index`, to make the logging statements a bit easier to read:

When the thread starts running `.update()`, it has its own version of all of the data **local** to the function. In the case of `.update()`, this is `local_copy`. This is definitely a good thing. Otherwise, two threads running the same function would always confuse each other. It means that all variables that are scoped (or local) to a function are **thread-safe**.

Now you can start walking through what happens if you run the program above with a single thread and a single call to `.update()`.

The image below steps through the execution of `.update()` if only a single thread is run. The statement is shown on the left followed by a diagram showing the values in the thread's `local_value` and the shared `database.value`:

The diagram is laid out so that time increases as you move from top to bottom. It begins when Thread 1 is created and ends when it is terminated.

When Thread 1 starts, `FakeDatabase.value` is zero. The first line of code in the method, `local_copy = self.value`, copies the value zero to the local variable. Next it increments the value of `local_copy` with the `local_copy += 1` statement. You can see `.value` in Thread 1 getting set to one.

Next `time.sleep()` is called, which makes the current thread pause and allows other threads to run. Since there is only one thread in this example, this has no effect.

When Thread 1 wakes up and continues, it copies the new value from `local_copy` to `FakeDatabase.value`, and then the thread is complete. You can see that `database.value` is set to one.

So far, so good. You ran `.update()` once and `FakeDatabase.value` was incremented to one.

Two Threads

Getting back to the race condition, the two threads will be running concurrently but not at the same time. They will each have their own version of `local_copy` and will each point to the same database. It is this shared database object that is going to cause the problems.

The program starts with Thread 1 running `.update()`:

When Thread 1 calls `time.sleep()`, it allows the other thread to start running. This is where things get interesting.

Thread 2 starts up and does the same operations. It's also copying `database.value` into its private `local_copy` and

Thread 2 starts up and does the same operations. It's also copying database.value into its private local_copy, and this shared database.value has not yet been updated:

When Thread 2 finally goes to sleep, the shared database.value is still unmodified at zero, and both private versions of local_copy have the value one.

Thread 1 now wakes up and saves its version of local_copy and then terminates, giving Thread 2 a final chance to run. Thread 2 has no idea that Thread 1 ran and updated database.value while it was sleeping. It stores *its* version of local_copy into database.value, also setting it to one:

The two threads have interleaving access to a single shared object, overwriting each other's results. Similar race conditions can arise when one thread frees memory or closes a file handle before the other thread is finished accessing it.

Why This Isn't a Silly Example

The example above is contrived to make sure that the race condition happens every time you run your program. Because the operating system can swap out a thread at any time, it is possible to interrupt a statement like $x = x + 1$ after it has read the value of x but before it has written back the incremented value.

The details of how this happens are quite interesting, but not needed for the rest of this article, so feel free to skip over this hidden section.

Now that you've seen a race condition in action, let's find out how to solve them!

Basic Synchronization Using Lock

There are a number of ways to avoid or solve race conditions. You won't look at all of them here, but there are a couple that are used frequently. Let's start with Lock.

To solve your race condition above, you need to find a way to allow only one thread at a time into the read-modify-write section of your code. The most common way to do this is called Lock in Python. In some other languages this same idea is called a mutex. Mutex comes from MUTual EXclusion, which is exactly what a Lock does.

A Lock is an object that acts like a hall pass. Only one thread at a time can have the Lock. Any other thread that wants the Lock must wait until the owner of the Lock gives it up.

The basic functions to do this are `.acquire()` and `.release()`. A thread will call `my_lock.acquire()` to get the lock. If the lock is already held, the calling thread will wait until it is released. There's an important point here. If one thread gets the lock but never gives it back, your program will be stuck. You'll read more about this later.

Fortunately, Python's Lock will also operate as a context manager, so you can use it in a `with` statement, and it gets released automatically when the `with` block exits for any reason.

Let's look at the `FakeDatabase` with a Lock added to it. The calling function stays the same:

Python

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()

    def locked_update(self, name):
        logging.info("Thread %s: starting update", name)
        logging.debug("Thread %s about to lock", name)
        with self._lock:
            logging.debug("Thread %s has lock", name)
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug("Thread %s about to release lock", name)
            logging.debug("Thread %s after release", name)
        logging.info("Thread %s: finishing update", name)
```

Other than adding a bunch of debug logging so you can see the locking more clearly, the big change here is to add a member called `_lock`, which is a `threading.Lock()` object. This `_lock` is initialized in the unlocked state and locked and released by the `with` statement.

It's worth noting here that the thread running this function will hold on to that Lock until it is completely finished updating the database. In this case, that means it will hold the Lock while it copies, updates, sleeps, and then writes the value back to the database.

If you run this version with logging set to warning level, you'll see this:

Shell

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

Look at that. Your program finally works!

You can turn on full logging by setting the level to DEBUG by adding this statement after you configure the logging output in `__main__`:

Python

```
logging.getLogger().setLevel(logging.DEBUG)
```

Running this program with DEBUG logging turned on looks like this:

Shell

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 0 about to lock
Thread 0 has lock
Thread 1: starting update
Thread 1 about to lock
Thread 0 about to release lock
Thread 0 after release
Thread 0: finishing update
Thread 1 has lock
Thread 1 about to release lock
Thread 1 after release
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

In this output you can see Thread 0 acquires the lock and is still holding it when it goes to sleep. Thread 1 then starts and attempts to acquire the same lock. Because Thread 0 is still holding it, Thread 1 has to wait. This is the mutual exclusion that a Lock provides.

Many of the examples in the rest of this article will have WARNING and DEBUG level logging. We'll generally only show the WARNING level output, as the DEBUG logs can be quite lengthy. Try out the programs with the logging turned up and see what they do.

Deadlock

Before you move on, you should look at a common problem when using Locks. As you saw, if the Lock has already been acquired, a second call to `.acquire()` will wait until the thread that is holding the Lock calls `.release()`. What do you think happens when you run this code:

Python

```
import threading

l = threading.Lock()
print("before first acquire")
l.acquire()
print("before second acquire")
l.acquire()
print("acquired lock twice")
```

When the program calls `l.acquire()` the second time, it hangs waiting for the Lock to be released. In this example, you can fix the deadlock by removing the second call, but deadlocks usually happen from one of two subtle things:

1. An implementation bug where a Lock is not released properly
2. A design issue where a utility function needs to be called by functions that might or might not already have the Lock

The first situation happens sometimes, but using a Lock as a context manager greatly reduces how often. It is recommended to write code whenever possible to make use of context managers, as they help to avoid situations where an exception skips you over the `.release()` call.

The design issue can be a bit trickier in some languages. Thankfully, Python threading has a second object, called RLock, that is designed for just this situation. It allows a thread to `.acquire()` an RLock multiple times before it calls `.release()`. That thread is still required to call `.release()` the same number of times it called `.acquire()`, but it should be doing that anyway.

Lock and RLock are two of the basic tools used in threaded programming to prevent race conditions. There are a few other that work in different ways. Before you look at them, let's shift to a slightly different problem domain.

Producer-Consumer Threading

The [Producer-Consumer Problem](#) is a standard computer science problem used to look at threading or process synchronization issues. You're going to look at a variant of it to get some ideas of what primitives the Python threading module provides.

For this example, you're going to imagine a program that needs to read messages from a network and write them to disk. The program does not request a message when it wants. It must be listening and accept messages as they come in. The messages will not come in at a regular pace, but will be coming in bursts. This part of the program is called the producer.

On the other side, once you have a message, you need to write it to a database. The database access is slow, but fast enough to keep up to the average pace of messages. It is *not* fast enough to keep up when a burst of messages comes in. This part is the consumer.

In between the producer and the consumer, you will create a Pipeline that will be the part that changes as you learn about different synchronization objects.

That's the basic layout. Let's look at a solution using Lock. It doesn't work perfectly, but it uses tools you already know, so it's a good place to start.

Producer-Consumer Using Lock

Since this is an article about Python threading, and since you just read about the Lock primitive, let's try to solve this problem with two threads using a Lock or two.

The general design is that there is a producer thread that reads from the fake network and puts the message into a Pipeline:

Python

```
import random

SENTINEL = object()

def producer(pipeline):
    """Pretend we're getting a message from the network."""
    for index in range(10):
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        pipeline.set_message(message, "Producer")

    # Send a sentinel message to tell consumer we're done
    pipeline.set_message(SENTINEL, "Producer")
```

To generate a fake message, the producer gets a random number between one and one hundred. It calls `.set_message()` on the pipeline to send it to the consumer.

The producer also uses a SENTINEL value to signal the consumer to stop after it has sent ten values. This is a little awkward, but don't worry, you'll see ways to get rid of this SENTINEL value after you work through this example.

On the other side of the pipeline is the consumer:

Python

```
def consumer(pipeline):
    """Pretend we're saving a number in the database."""
    message = 0
    while message is not SENTINEL:
        message = pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info("Consumer storing message: %s", message)
```

The consumer reads a message from the pipeline and writes it to a fake database, which in this case is just printing it to the display. If it gets the `SENTINEL` value, it returns from the function, which will terminate the thread.

Before you look at the really interesting part, the `Pipeline`, here's the `__main__` section, which spawns these threads:

Python

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    # logging.getLogger().setLevel(logging.DEBUG)

    pipeline = Pipeline()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline)
        executor.submit(consumer, pipeline)
```

This should look fairly familiar as it's close to the `__main__` code in the previous examples.

Remember that you can turn on `DEBUG` logging to see all of the logging messages by uncommenting this line:

Python

```
# logging.getLogger().setLevel(logging.DEBUG)
```

It can be worthwhile to walk through the `DEBUG` logging messages to see exactly where each thread acquires and releases the locks.

Now let's take a look at the `Pipeline` that passes messages from the producer to the consumer:

Python

```
class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer.
    """

    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self, name):
        logging.debug("%s:about to acquire getlock", name)
        self.consumer_lock.acquire()
        logging.debug("%s:have getlock", name)
        message = self.message
        logging.debug("%s:about to release setlock", name)
        self.producer_lock.release()
        logging.debug("%s:setlock released", name)
        return message

    def set_message(self, message, name):
        logging.debug("%s:about to acquire setlock", name)
        self.producer_lock.acquire()
        logging.debug("%s:have setlock", name)
        self.message = message
        logging.debug("%s:about to release getlock", name)
        self.consumer_lock.release()
```

```
logging.debug("%s:getlock released", name)
```

Woah! That's a lot of code. A pretty high percentage of that is just logging statements to make it easier to see what's happening when you run it. Here's the same code with all of the logging statements removed:

Python

```
class Pipeline:  
    """  
    Class to allow a single element pipeline between producer and consumer.  
    """  
  
    def __init__(self):  
        self.message = 0  
        self.producer_lock = threading.Lock()  
        self.consumer_lock = threading.Lock()  
        self.consumer_lock.acquire()  
  
    def get_message(self, name):  
        self.consumer_lock.acquire()  
        message = self.message  
        self.producer_lock.release()  
        return message  
  
    def set_message(self, message, name):  
        self.producer_lock.acquire()  
        self.message = message  
        self.consumer_lock.release()
```

That seems a bit more manageable. The `Pipeline` in this version of your code has three members:

1. `.message` stores the message to pass.
2. `.producer_lock` is a `threading.Lock` object that restricts access to the message by the producer thread.
3. `.consumer_lock` is also a `threading.Lock` that restricts access to the message by the consumer thread.

`__init__()` initializes these three members and then calls `.acquire()` on the `.consumer_lock`. This is the state you want to start in. The producer is allowed to add a new message, but the consumer needs to wait until a message is present.

`.get_message()` and `.set_message()` are nearly opposites. `.get_message()` calls `.acquire()` on the `consumer_lock`. This is the call that will make the consumer wait until a message is ready.

Once the consumer has acquired the `.consumer_lock`, it copies out the value in `.message` and then calls `.release()` on the `.producer_lock`. Releasing this lock is what allows the producer to insert the next message into the pipeline.

Before you go on to `.set_message()`, there's something subtle going on in `.get_message()` that's pretty easy to miss. It might seem tempting to get rid of `message` and just have the function end with `return self.message`. See if you can figure out why you don't want to do that before moving on.

Here's the answer. As soon as the consumer calls `.producer_lock.release()`, it can be swapped out, and the producer can start running. That could happen before `.release()` returns! This means that there is a slight possibility that when the function returns `self.message`, that could actually be the *next* message generated, so you would lose the first message. This is another example of a race condition.

Moving on to `.set_message()`, you can see the opposite side of the transaction. The producer will call this with a message. It will acquire the `.producer_lock`, set the `.message`, and the call `.release()` on the `consumer_lock`, which will allow the consumer to read that value.

Let's run the code that has logging set to `WARNING` and see what it looks like:

Shell

```
$ ./prodcom_lock.py
Producer got data 43
Producer got data 45
Consumer storing data: 43
Producer got data 86
Consumer storing data: 45
Producer got data 40
Consumer storing data: 86
Producer got data 62
Consumer storing data: 40
Producer got data 15
Consumer storing data: 62
Producer got data 16
Consumer storing data: 15
Producer got data 61
Consumer storing data: 16
Producer got data 73
Consumer storing data: 61
Producer got data 22
Consumer storing data: 73
Consumer storing data: 22
```

At first, you might find it odd that the producer gets two messages before the consumer even runs. If you look back at the producer and `.set_message()`, you will notice that the only place it will wait for a Lock is when it attempts to put the message into the pipeline. This is done after the producer gets the message and logs that it has it.

When the producer attempts to send this second message, it will call `.set_message()` the second time and it will block.

The operating system can swap threads at any time, but it generally lets each thread have a reasonable amount of time to run before swapping it out. That's why the producer usually runs until it blocks in the second call to `.set_message()`.

Once a thread is blocked, however, the operating system will always swap it out and find a different thread to run. In this case, the only other thread with anything to do is the consumer.

The consumer calls `.get_message()`, which reads the message and calls `.release()` on the `.producer_lock`, thus allowing the producer to run again the next time threads are swapped.

Notice that the first message was 43, and that is exactly what the consumer read, even though the producer had already generated the 45 message.

While it works for this limited test, it is not a great solution to the producer-consumer problem in general because it only allows a single value in the pipeline at a time. When the producer gets a burst of messages, it will have nowhere to put them.

Let's move on to a better way to solve this problem, using a queue.

Producer-Consumer Using Queue

If you want to be able to handle more than one value in the pipeline at a time, you'll need a data structure for the pipeline that allows the number to grow and shrink as data backs up from the producer.

Python's standard library has a queue module which, in turn, has a Queue class. Let's change the Pipeline to use a queue instead of just a variable protected by a Lock. You'll also use a different way to stop the worker threads by using a different primitive from Python threading, an Event.

Let's start with the Event. The `threading.Event` object allows one thread to signal an event while many other threads can be waiting for that event to happen. The key usage in this code is that the threads that are waiting for the event do not necessarily need to stop what they are doing, they can just check the status of the Event every once in a while.

The triggering of the event can be many things. In this example, the main thread will simply sleep for a while and

then .set() it:

Python

```
1 if __name__ == "__main__":
2     format = "%(asctime)s: %(message)s"
3     logging.basicConfig(format=format, level=logging.INFO,
4                         datefmt="%H:%M:%S")
5     # logging.getLogger().setLevel(logging.DEBUG)
6
7     pipeline = Pipeline()
8     event = threading.Event()
9     with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
10         executor.submit(producer, pipeline, event)
11         executor.submit(consumer, pipeline, event)
12
13         time.sleep(0.1)
14         logging.info("Main: about to set event")
15         event.set()
```

The only changes here are the creation of the event object on line 6, passing the event as a parameter on lines 8 and 9, and the final section on lines 11 to 13, which sleep for a second, log a message, and then call .set() on the event.

The producer also did not have to change too much:

Python

```
1 def producer(pipeline, event):
2     """Pretend we're getting a number from the network."""
3     while not event.is_set():
4         message = random.randint(1, 101)
5         logging.info("Producer got message: %s", message)
6         pipeline.set_message(message, "Producer")
7
8     logging.info("Producer received EXIT event. Exiting")
```

It now will loop until it sees that the event was set on line 3. It also no longer puts the SENTINEL value into the pipeline.

consumer had to change a little more:

Python

```
1 def consumer(pipeline, event):
2     """Pretend we're saving a number in the database."""
3     while not event.is_set() or not pipeline.empty():
4         message = pipeline.get_message("Consumer")
5         logging.info(
6             "Consumer storing message: %s (queue size=%s)",
7             message,
8             pipeline.qsize(),
9         )
10
11     logging.info("Consumer received EXIT event. Exiting")
```

While you got to take out the code related to the SENTINEL value, you did have to do a slightly more complicated while condition. Not only does it loop until the event is set, but it also needs to keep looping until the pipeline has been emptied.

Making sure the queue is empty before the consumer finishes prevents another fun issue. If the consumer does exit while the pipeline has messages in it, there are two bad things that can happen. The first is that you lose those final messages, but the more serious one is that the producer can get caught attempting to add a message to a full queue and never return.

This happens if the event gets triggered after the producer has checked the .is_set() condition but before it calls pipeline.set_message().

If that happens, it's possible for the producer to wake up and exit with the queue still completely full. The producer

will then call `.set_message()` which will wait until there is space on the queue for the new message. The consumer has already exited, so this will not happen and the producer will not exit.

The rest of the consumer should look familiar.

The Pipeline has changed dramatically, however:

Python

```
1 class Pipeline(queue.Queue):
2     def __init__(self):
3         super().__init__(maxsize=10)
4
5     def get_message(self, name):
6         logging.debug("%s:about to get from queue", name)
7         value = self.get()
8         logging.debug("%s:got %d from queue", name, value)
9         return value
10
11    def set_message(self, value, name):
12        logging.debug("%s:about to add %d to queue", name, value)
13        self.put(value)
14        logging.debug("%s:added %d to queue", name, value)
```

You can see that Pipeline is a subclass of `queue.Queue`. Queue has an optional parameter when initializing to specify a maximum size of the queue.

If you give a positive number for `maxsize`, it will limit the queue to that number of elements, causing `.put()` to block until there are fewer than `maxsize` elements. If you don't specify `maxsize`, then the queue will grow to the limits of your computer's memory.

`.get_message()` and `.set_message()` got much smaller. They basically wrap `.get()` and `.put()` on the Queue. You might be wondering where all of the locking code that prevents the threads from causing race conditions went.

The core devs who wrote the standard library knew that a Queue is frequently used in multi-threading environments and incorporated all of that locking code inside the Queue itself. Queue is thread-safe.

Running this program looks like the following:

Shell

```
$ ./prodcom_queue.py
Producer got message: 32
Producer got message: 51
Producer got message: 25
Producer got message: 94
Producer got message: 29
Consumer storing message: 32 (queue size=3)
Producer got message: 96
```

```
Consumer storing message: 51 (queue size=3)
Producer got message: 6
Consumer storing message: 25 (queue size=3)
Producer got message: 31

[many lines deleted]

Producer got message: 80
Consumer storing message: 94 (queue size=6)
Producer got message: 33
Consumer storing message: 20 (queue size=6)
Producer got message: 48
Consumer storing message: 31 (queue size=6)
Producer got message: 52
Consumer storing message: 98 (queue size=6)
Main: about to set event
Producer got message: 13
Consumer storing message: 59 (queue size=6)
Producer received EXIT event. Exiting
Consumer storing message: 75 (queue size=6)
Consumer storing message: 97 (queue size=5)
Consumer storing message: 80 (queue size=4)
Consumer storing message: 33 (queue size=3)
Consumer storing message: 48 (queue size=2)
Consumer storing message: 52 (queue size=1)
Consumer storing message: 13 (queue size=0)
Consumer received EXIT event. Exiting
```

If you read through the output in my example, you can see some interesting things happening. Right at the top, you can see the producer got to create five messages and place four of them on the queue. It got swapped out by the operating system before it could place the fifth one.

The consumer then ran and pulled off the first message. It printed out that message as well as how deep the queue was at that point:

Shell

```
Consumer storing message: 32 (queue size=3)
```

This is how you know that the fifth message hasn't made it into the pipeline yet. The queue is down to size three after a single message was removed. You also know that the queue can hold ten messages, so the producer thread didn't get blocked by the queue. It was swapped out by the OS.

Note: Your output will be different. Your output will change from run to run. That's the fun part of working with threads!

As the program starts to wrap up, can you see the main thread generating the event which causes the producer to exit immediately. The consumer still has a bunch of work to do, so it keeps running until it has cleaned out the pipeline.

Try playing with different queue sizes and calls to `time.sleep()` in the producer or the consumer to simulate longer network or disk access times respectively. Even slight changes to these elements of the program will make large differences in your results.

This is a much better solution to the producer-consumer problem, but you can simplify it even more. The Pipeline really isn't needed for this problem. Once you take away the logging, it just becomes a `queue.Queue`.

Here's what the final code looks like using `queue.Queue` directly:

Python

```
import concurrent.futures
import logging
import queue
import random
import threading
import time
```

```

def producer(queue, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        queue.put(message)

    logging.info("Producer received event. Exiting")

def consumer(queue, event):
    """Pretend we're saving a number in the database."""
    while not event.is_set() or not queue.empty():
        message = queue.get()
        logging.info(
            "Consumer storing message: %s (size=%d)", message, queue.qsize()
        )

    logging.info("Consumer received event. Exiting")

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    pipeline = queue.Queue(maxsize=10)
    event = threading.Event()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline, event)
        executor.submit(consumer, pipeline, event)

    time.sleep(0.1)
    logging.info("Main: about to set event")
    event.set()

```

That's easier to read and shows how using Python's built-in primitives can simplify a complex problem.

Lock and Queue are handy classes to solve concurrency issues, but there are others provided by the standard library. Before you wrap up this tutorial, let's do a quick survey of some of them.

Threading Objects

There are a few more primitives offered by the Python threading module. While you didn't need these for the examples above, they can come in handy in different use cases, so it's good to be familiar with them.

Semaphore

The first Python threading object to look at is `threading.Semaphore`. A Semaphore is a counter with a few special properties. The first one is that the counting is atomic. This means that there is a guarantee that the operating system will not swap out the thread in the middle of incrementing or decrementing the counter.

The internal counter is incremented when you call `.release()` and decremented when you call `.acquire()`.

The next special property is that if a thread calls `.acquire()` when the counter is zero, that thread will block until a different thread calls `.release()` and increments the counter to one.

Semaphores are frequently used to protect a resource that has a limited capacity. An example would be if you have a pool of connections and want to limit the size of that pool to a specific number.

Timer

A `threading.Timer` is a way to schedule a function to be called after a certain amount of time has passed. You create a Timer by passing in a number of seconds to wait and a function to call:

Python

```
t = threading.Timer(30.0, my_function)
```

You start the timer by calling `.start()`. The function will be called on a new thread at some point after the specified time, but be aware that there is no promise that it will be called exactly at the time you want.

If you want to stop a Timer that you've already started, you can cancel it by calling `.cancel()`. Calling `.cancel()` after the Timer has triggered does nothing and does not produce an exception.

A Timer can be used to prompt a user for action after a specific amount of time. If the user does the action before the Timer expires, `.cancel()` can be called.

Barrier

A `threading.Barrier` can be used to keep a fixed number of threads in sync. When creating a Barrier, the caller must specify how many threads will be synchronizing on it. Each thread calls `.wait()` on the Barrier. They all will remain blocked until the specified number of threads are waiting, and then they are all released at the same time.

Remember that threads are scheduled by the operating system so, even though all of the threads are released simultaneously, they will be scheduled to run one at a time.

One use for a Barrier is to allow a pool of threads to initialize themselves. Having the threads wait on a Barrier after they are initialized will ensure that none of the threads start running before all of the threads are finished with their initialization.

Conclusion: Threading in Python

You've now seen much of what Python threading has to offer and some examples of how to build threaded programs and the problems they solve. You've also seen a few instances of the problems that arise when writing and debugging threaded programs.

If you'd like to explore other options for concurrency in Python, check out [Speed Up Your Python Program With Concurrency](#).

If you're interested in doing a deep dive on the `asyncio` module, go read [Async IO in Python: A Complete Walkthrough](#).

Whatever you do, you now have the information and confidence you need to write programs using Python threading!

Special thanks to reader JL Diaz for helping to clean up the introduction.

 **Take the Quiz:** Test your knowledge with our interactive “Python Threading” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Threading in Python](#)

About Jim Anderson

Jim has been programming for a long time in a variety of languages. He has worked on embedded systems, built distributed build systems, done off-shore vendor management, and spoken in many, many meetings.

[» More about Jim](#)

worked on this tutorial are:

[Aldren](#)

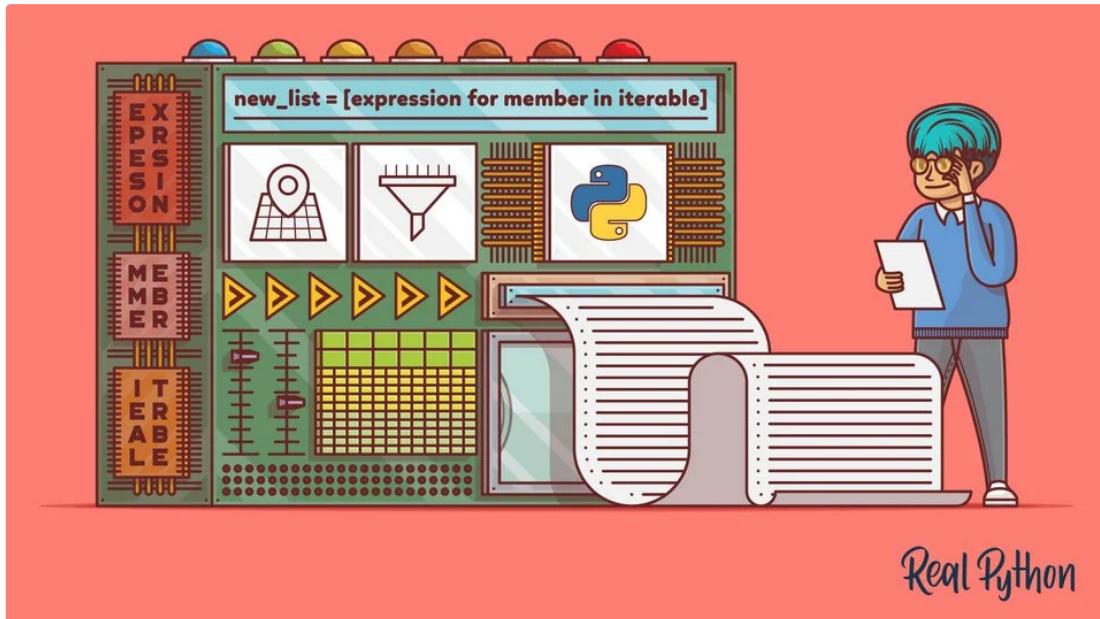
[Brad](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#)

Recommended Video Course: [Threading in Python](#)



Real Python

When to Use a List Comprehension in Python

by James Timmins · Nov 06, 2019 · 17 Comments · basics · python

Tweet Share Email

Table of Contents

- [How to Create Lists in Python](#)
 - [Using for Loops](#)
 - [Using map\(\) Objects](#)
 - [Using List Comprehensions](#)
 - [Benefits of Using List Comprehensions](#)
- [How to Supercharge Your Comprehensions](#)
 - [Using Conditional Logic](#)
 - [Using Set and Dictionary Comprehensions](#)
 - [Using the Walrus Operator](#)
- [When Not to Use a List Comprehension in Python](#)
 - [Watch Out for Nested Comprehensions](#)
 - [Choose Generators for Large Datasets](#)
 - [Profile to Optimize Performance](#)
- [Conclusion](#)



Python is famous for allowing you to write code that's elegant, easy to write, and almost as easy to read as plain English. One of the language's most distinctive features is the **list comprehension**, which you can use to create powerful functionality within a single line of code. However, many developers struggle to fully leverage the more advanced features of a list comprehension in Python. Some programmers even use them too much, which can lead to code that's less efficient and harder to read.

By the end of this tutorial, you'll understand the full power of Python list comprehensions and how to use their features comfortably. You'll also gain an understanding of the trade-offs that come with using them so that you can determine when other approaches are more preferable.

In this tutorial, you'll learn how to:

- Rewrite loops and `map()` calls as a list comprehension in Python

- Rewrite loops and map() calls as a [list comprehension](#) in Python

- **Choose** between comprehensions, loops, and map() calls
- Supercharge your comprehensions with **conditional logic**
- **Use comprehensions** to replace filter()
- **Profile** your code to solve performance questions

Free Bonus: [Click here to get access to a chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

[Remove ads](#)

How to Create Lists in Python

There are a few different ways you can create [lists](#) in Python. To better understand the trade-offs of using a list comprehension in Python, let's first see how to create lists with these approaches.

Using for Loops

The most common type of loop is the [for](#) loop. You can use a for loop to create a list of elements in three steps:

1. Instantiate an empty list.
2. Loop over an iterable or [range](#) of elements.
3. Append each element to the end of the list.

If you want to create a list containing the first ten perfect squares, then you can complete these steps in three lines of code:

```
Python >>>
>>> squares = []
>>> for i in range(10):
...     squares.append(i * i)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here, you instantiate an empty list, squares. Then, you use a for loop to iterate over range(10). Finally, you multiply each number by itself and append the result to the end of the list.

Using map() Objects

[map\(\)](#) provides an alternative approach that's based in [functional programming](#). You pass in a function and an iterable, and map() will create an object. This object contains the output you would get from running each iterable element through the supplied function.

As an example, consider a situation in which you need to calculate the price after tax for a list of transactions:

```
Python >>>
>>> txns = [1.09, 23.56, 57.84, 4.56, 6.78]
>>> TAX_RATE = .08
>>> def get_price_with_tax(txn):
...     return txn * (1 + TAX_RATE)
>>> final_prices = map(get_price_with_tax, txns)
[1.19, 25.81, 63.61, 4.91, 7.31]
```

```
>>> list(final_prices)
[1.1772000000000002, 25.4448, 62.46720000000005, 4.9248, 7.32240000000001]
```

Here, you have an iterable `txns` and a function `get_price_with_tax()`. You pass both of these arguments to `map()`, and store the resulting object in `final_prices`. You can easily convert this map object into a list using `list()`.

Using List Comprehensions

List comprehensions are a third way of making lists. With this elegant approach, you could rewrite the `for` loop from the first example in just a single line of code:

```
Python >>>
>>> squares = [i * i for i in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Rather than creating an empty list and adding each element to the end, you simply define the list and its contents at the same time by following this format:

```
Python >>>
new_list = [expression for member in iterable]
```

Every list comprehension in Python includes three elements:

1. **expression** is the member itself, a call to a method, or any other valid expression that returns a value. In the example above, the expression `i * i` is the square of the member value.
2. **member** is the object or value in the list or iterable. In the example above, the member value is `i`.
3. **iterable** is a list, [set](#), sequence, [generator](#), or any other object that can return its elements one at a time. In the example above, the iterable is `range(10)`.

Because the **expression** requirement is so flexible, a list comprehension in Python works well in many places where you would use `map()`. You can rewrite the pricing example with its own list comprehension:

```
Python >>>
>>> txns = [1.09, 23.56, 57.84, 4.56, 6.78]
>>> TAX_RATE = .08
>>> def get_price_with_tax(txn):
...     return txn * (1 + TAX_RATE)
>>> final_prices = [get_price_with_tax(i) for i in txns]
>>> final_prices
[1.1772000000000002, 25.4448, 62.46720000000005, 4.9248, 7.32240000000001]
```

The only distinction between this implementation and `map()` is that the list comprehension in Python returns a list, not a map object.

[Remove ads](#)

Benefits of Using List Comprehensions

List comprehensions are often described as being more [Pythonic](#) than loops or `map()`. But rather than blindly accepting that assessment, it's worth it to understand the benefits of using a list comprehension in Python when compared to the alternatives. Later on, you'll learn about a few scenarios where the alternatives are a better choice.

One main benefit of using a list comprehension in Python is that it's a single tool that you can use in many different situations. In addition to standard [list creation](#), list comprehensions can also be used for mapping and filtering. You don't have to use a different approach for each scenario.

This is the main reason why list comprehensions are considered **Pythonic**, as Python embraces simple, powerful tools that you can use in a wide variety of situations. As an added side benefit, whenever you use a list

comprehension in Python, you won't need to remember the proper order of arguments like you would when you call `map()`.

List comprehensions are also more **declarative** than loops, which means they're easier to read and understand. Loops require you to focus on how the list is created. You have to manually create an empty list, loop over the elements, and add each of them to the end of the list. With a list comprehension in Python, you can instead focus on *what* you want to go in the list and trust that Python will take care of *how* the list construction takes place.

How to Supercharge Your Comprehensions

In order to understand the full value that [list comprehensions](#) can provide, it's helpful to understand their range of possible functionality. You'll also want to understand the changes that are coming to the list comprehension in Python 3.8.

Using Conditional Logic

Earlier, you saw this formula for how to create list comprehensions:

```
Python >>>
new_list = [expression for member in iterable]
```

While this formula is accurate, it's also a bit incomplete. A more complete description of the comprehension formula adds support for optional **conditionals**. The most common way to add [conditional logic](#) to a list comprehension is to add a conditional to the end of the expression:

```
Python >>>
new_list = [expression for member in iterable (if conditional)]
```

Here, your conditional statement comes just before the closing bracket.

Conditionals are important because they allow list comprehensions to filter out unwanted values, which would normally require a call to `filter()`:

```
Python >>>
>>> sentence = 'the rocket came back from mars'
>>> vowels = [i for i in sentence if i in 'aeiou']
>>> vowels
['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

In this code block, the conditional statement filters out any characters in `sentence` that aren't a vowel.

The conditional can test any valid expression. If you need a more complex filter, then you can even move the conditional logic to a separate function:

```
Python >>>
>>> sentence = 'The rocket, who was named Ted, came back \
... from Mars because he missed his friends.'
>>> def is_consonant(letter):
...     vowels = 'aeiou'
...     return letter.isalpha() and letter.lower() not in vowels
>>> consonants = [i for i in sentence if is_consonant(i)]
['T', 'h', 'r', 'c', 'k', 't', 'w', 'h', 'w', 's', 'n', 'm', 'd', ' ', \
'T', 'd', 'c', 'm', 'b', 'c', 'k', 'f', 'r', 'm', 'M', 'r', 's', 'b', ' ', \
'c', 's', 'h', 'm', 's', 'd', 'h', 's', 'f', 'r', 'n', 'd', 's']
```

Here, you create a complex filter `is_consonant()` and pass this function as the conditional statement for your list comprehension. Note that the member value `i` is also passed as an argument to your function.

You can place the conditional at the end of the statement for simple filtering, but what if you want to *change* a member value instead of filtering it out? In this case, it's useful to place the conditional near the *beginning* of the expression:

Python

>>>

```
new_list = [expression (if conditional) for member in iterable]
```

With this formula, you can use conditional logic to select from multiple possible output options. For example, if you have a list of prices, then you may want to replace negative prices with 0 and leave the positive values unchanged:

Python

>>>

```
>>> original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
>>> prices = [i if i > 0 else 0 for i in original_prices]
>>> prices
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Here, your expression `i` contains a conditional statement, `if i > 0 else 0`. This tells Python to output the value of `i` if the number is positive, but to change `i` to 0 if the number is negative. If this seems overwhelming, then it may be helpful to view the conditional logic as its own function:

Python

>>>

```
>>> def get_price(price):
...     return price if price > 0 else 0
>>> prices = [get_price(i) for i in original_prices]
>>> prices
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Now, your conditional statement is contained within `get_price()`, and you can use it as part of your list comprehension expression.

[Remove ads](#)

Using Set and Dictionary Comprehensions

While the list comprehension in Python is a common tool, you can also create set and dictionary comprehensions. A **set comprehension** is almost exactly the same as a list comprehension in Python. The difference is that set comprehensions make sure the output contains no duplicates. You can create a set comprehension by using curly braces instead of brackets:

Python

>>>

```
>>> quote = "life, uh, finds a way"
>>> unique_vowels = {i for i in quote if i in 'aeiou'}
>>> unique_vowels
{'a', 'e', 'u', 'i'}
```

Your set comprehension outputs all the unique vowels it found in `quote`. Unlike lists, sets don't guarantee that items will be saved in any particular order. This is why the first member of the set is `a`, even though the first vowel in `quote` is `i`.

Dictionary comprehensions are similar, with the additional requirement of defining a key:

Python

>>>

```
>>> squares = {i: i * i for i in range(10)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

To create the squares dictionary, you use curly braces ({}) as well as a key-value pair (i: i * i) in your expression.

Using the Walrus Operator

Python 3.8 will introduce the [assignment expression](#), also known as the **walrus operator**. To understand how you can use it, consider the following example.

Say you need to make ten requests to an API that will return temperature data. You only want to return results that are greater than 100 degrees Fahrenheit. Assume that each request will return different data. In this case, there's no way to use a list comprehension in Python to solve the problem. The formula expression for member in iterable (if conditional) provides no way for the conditional to assign data to a variable that the expression can access.

The **walrus operator** solves this problem. It allows you to run an expression while simultaneously assigning the output value to a variable. The following example shows how this is possible, using get_weather_data() to generate fake weather data:

Python

>>>

```
>>> import random
>>> def get_weather_data():
...     return random.randrange(90, 110)
>>> hot_temps = [temp for _ in range(20) if (temp := get_weather_data()) >= 100]
>>> hot_temps
[107, 102, 109, 104, 107, 109, 108, 101, 104]
```

You won't often need to use the assignment expression inside of a list comprehension in Python, but it's a useful tool to have at your disposal when necessary.

When Not to Use a List Comprehension in Python

List comprehensions are useful and can help you write elegant code that's easy to read and debug, but they're not the right choice for all circumstances. They might make your code run more slowly or use more memory. If your code is less performant or harder to understand, then it's probably better to choose an alternative.

Watch Out for Nested Comprehensions

Comprehensions can be **nested** to create combinations of lists, dictionaries, and sets within a collection. For example, say a climate laboratory is tracking the high temperature in five different cities for the first week of June. The perfect data structure for storing this data could be a Python list comprehension nested within a dictionary comprehension:

Python

>>>

```
>>> cities = ['Austin', 'Tacoma', 'Topeka', 'Sacramento', 'Charlotte']
>>> temps = {city: [0 for _ in range(7)] for city in cities}
>>> temps
{
    'Austin': [0, 0, 0, 0, 0, 0, 0],
    'Tacoma': [0, 0, 0, 0, 0, 0, 0],
    'Topeka': [0, 0, 0, 0, 0, 0, 0],
    'Sacramento': [0, 0, 0, 0, 0, 0, 0],
    'Charlotte': [0, 0, 0, 0, 0, 0, 0]
}
```

You create the outer collection temps with a dictionary comprehension. The expression is a key-value pair, which contains yet another comprehension. This code will quickly generate a list of data for each city in cities.

Nested lists are a common way to create **matrices**, which are often used for mathematical purposes. Take a look at

the code block below:

```
Python >>>
>>> matrix = [[i for i in range(5)] for _ in range(6)]
>>> matrix
[
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4]
]
```

The outer list comprehension `[... for _ in range(6)]` creates six rows, while the inner list comprehension `[i for i in range(5)]` fills each of these rows with values.

So far, the purpose of each nested comprehension is pretty intuitive. However, there are other situations, such as **flattening** nested lists, where the logic arguably makes your code more confusing. Take this example, which uses a nested list comprehension to flatten a matrix:

```
Python >>>
matrix = [
...     [0, 0, 0],
...     [1, 1, 1],
...     [2, 2, 2],
...
]
>>> flat = [num for row in matrix for num in row]
>>> flat
[0, 0, 0, 1, 1, 1, 2, 2, 2]
```

The code to flatten the matrix is concise, but it may not be so intuitive to understand how it works. On the other hand, if you were to use for loops to flatten the same matrix, then your code will be much more straightforward:

```
Python >>>
>>> matrix = [
...     [0, 0, 0],
...     [1, 1, 1],
...     [2, 2, 2],
...
]
>>> flat = []
>>> for row in matrix:
...     for num in row:
...         flat.append(num)
...
>>> flat
[0, 0, 0, 1, 1, 1, 2, 2, 2]
```

Now you can see that the code traverses one row of the matrix at a time, pulling out all the elements in that row before moving on to the next one.

While the single-line nested list comprehension might seem more Pythonic, what's most important is to write code that your team can easily understand and modify. When you choose your approach, you'll have to make a judgment call based on whether you think the comprehension helps or hurts readability.

[Remove ads](#)

Choose Generators for Large Datasets

A list comprehension in Python works by loading the entire output list into memory. For small or even medium-sized lists, this is generally fine. If you want to sum the squares of the first one-thousand integers, then a list comprehension will solve this problem admirably:

Python

>>>

```
>>> sum([i * i for i in range(1000)])
332833500
```

But what if you wanted to sum the squares of the first *billion* integers? If you tried then on your machine, then you may notice that your computer becomes non-responsive. That's because Python is trying to create a list with one billion integers, which consumes more memory than your computer would like. Your computer may not have the resources it needs to generate an enormous list and store it in memory. If you try to do it anyway, then your machine could slow down or even crash.

When the size of a list becomes problematic, it's often helpful to use a [generator](#) instead of a list comprehension in Python. A **generator** doesn't create a single, large data structure in memory, but instead returns an iterable. Your code can ask for the next value from the iterable as many times as necessary or until you've reached the end of your sequence, while only storing a single value at a time.

If you were to sum the first billion squares with a generator, then your program will likely run for a while, but it shouldn't cause your computer to freeze. The example below uses a generator:

Python

>>>

```
>>> sum(i * i for i in range(1000000000))
333333328333333335000000000
```

You can tell this is a generator because the expression isn't surrounded by brackets or curly braces. Optionally, generators can be surrounded by parentheses.

The example above still requires a lot of work, but it performs the operations **lazily**. Because of lazy evaluation, values are only calculated when they're explicitly requested. After the generator yields a value (for example, 567 * 567), it can add that value to the running sum, then discard that value and generate the next value (568 * 568). When the sum function requests the next value, the cycle starts over. This process keeps the memory footprint small.

`map()` also operates lazily, meaning memory won't be an issue if you choose to use it in this case:

Python

>>>

```
>>> sum(map(lambda i: i*i, range(1000000000)))
333333328333333335000000000
```

It's up to you whether you prefer the generator expression or `map()`.

Profile to Optimize Performance

So, which approach is faster? Should you use list comprehensions or one of their alternatives? Rather than adhere to a single rule that's true in all cases, it's more useful to ask yourself whether or not performance **matters** in your specific circumstance. If not, then it's usually best to choose whatever approach leads to the cleanest code!

If you're in a scenario where performance is important, then it's typically best to **profile** different approaches and listen to the data. `timeit` is a useful library for timing how long it takes chunks of code to run. You can use `timeit` to compare the runtime of `map()`, `for` loops, and list comprehensions:

```
Python >>>
>>> import random
>>> import timeit
>>> TAX_RATE = .08
>>> txns = [random.randrange(100) for _ in range(100000)]
>>> def get_price(txn):
...     return txn * (1 + TAX_RATE)
...
>>> def get_prices_with_map():
...     return list(map(get_price, txns))
...
>>> def get_prices_with_comprehension():
...     return [get_price(txn) for txn in txns]
...
>>> def get_prices_with_loop():
...     prices = []
...     for txn in txns:
...         prices.append(get_price(txn))
...     return prices
...
>>> timeit.timeit(get_prices_with_map, number=100)
2.0554370979998566
>>> timeit.timeit(get_prices_with_comprehension, number=100)
2.3982384680002724
>>> timeit.timeit(get_prices_with_loop, number=100)
3.0531821520007725
```

Here, you define three methods that each use a different approach for creating a list. Then, you tell `timeit` to run each of those functions 100 times each. `timeit` returns the total time it took to run those 100 executions.

As the code demonstrates, the biggest difference is between the loop-based approach and `map()`, with the loop taking 50% longer to execute. Whether or not this matters depends on the needs of your application.

Conclusion

In this tutorial, you learned how to use a **list comprehension** in Python to accomplish complex tasks without making your code overly complicated.

Now you can:

- Simplify loops and `map()` calls with declarative **list comprehensions**
- Supercharge your comprehensions with **conditional logic**
- Create **set** and **dictionary** comprehensions
- Determine when code clarity or performance dictates an **alternative approach**

Whenever you have to choose a list creation method, try multiple implementations and consider what's easiest to read and understand in your specific scenario. If performance is important, then you can use profiling tools to give you actionable data instead of relying on hunches or guesses about what works the best.

Remember that while Python list comprehensions get a lot of attention, your intuition and ability to use data when it counts will help you write clean code that serves the task at hand. This, ultimately, is the key to making your code Pythonic!

About James Timmins

James is a software consultant and Python developer. When he's not writing Python, he's usual writing about it in blog or book form.

[» More about James](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

[Jaya](#)

[Joanna](#)

[Mike](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)



Real Python

Linked Lists in Python: An Introduction

by [Pedro Pregueiro](#) · Apr 01, 2020 · [30 Comments](#) · [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Understanding Linked Lists](#)
 - [Main Concepts](#)
 - [Practical Applications](#)
 - [Performance Comparison: Lists vs Linked Lists](#)
- [Introducing collections.deque](#)
 - [How to Use collections.deque](#)
 - [How to Implement Queues and Stacks](#)
- [Implementing Your Own Linked List](#)
 - [How to Create a Linked List](#)
 - [How to Traverse a Linked List](#)
 - [How to Insert a New Node](#)
 - [How to Remove a Node](#)
- [Using Advanced Linked Lists](#)
 - [How to Use Doubly Linked Lists](#)
 - [How to Use Circular Linked Lists](#)
- [Conclusion](#)



Linked lists are like a lesser-known cousin of [lists](#). They're not as popular or as cool, and you might not even remember them from your algorithms class. But in the right context, they can really shine.

In this article, you'll learn:

- What linked lists are and when you should use them
- How to use `collections.deque` for all of your linked list needs
- How to implement your own linked lists
- What the other types of linked lists are and what they can be used for

If you're looking to brush up on your coding skills for a [job interview](#), or if you want to learn more about [Python data structures](#) besides the usual [dictionaries](#) and [lists](#), then you've come to the right place!

You can follow along with the examples in this tutorial by downloading the source code available at the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about linked lists in this tutorial.

Understanding Linked Lists

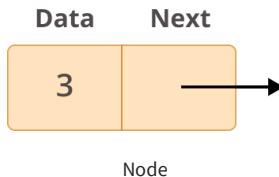
Linked lists are an ordered collection of objects. So what makes them different from normal lists? Linked lists differ from lists in the way that they store elements in memory. While lists use a contiguous memory block to store references to their data, linked lists store references as part of their own elements.

Main Concepts

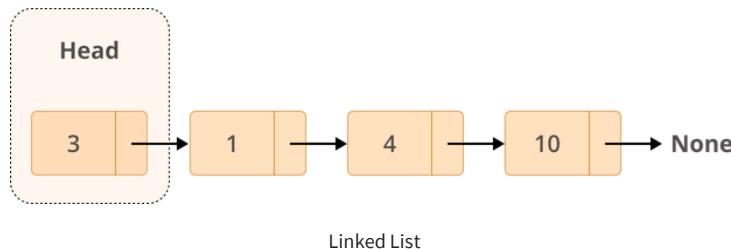
Before going more in depth on what linked lists are and how you can use them, you should first learn how they are structured. Each element of a linked list is called a **node**, and every node has two different fields:

1. **Data** contains the value to be stored in the node.
2. **Next** contains a reference to the next node on the list.

Here's what a typical node looks like:



A linked list is a collection of nodes. The first node is called the **head**, and it's used as the starting point for any iteration through the list. The last node must have its next reference pointing to [None](#) to determine the end of the list. Here's how it looks:



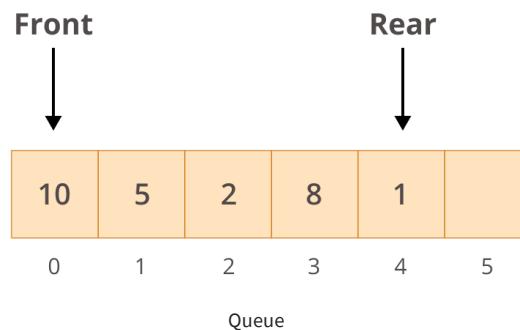
Now that you know how a linked list is structured, you're ready to look at some practical use cases for it.

Practical Applications

Linked lists serve a variety of purposes in the real world. They can be used to implement (*spoiler alert!*) queues or [stacks](#) as well as graphs. They're also useful for much more complex tasks, such as lifecycle management for an operating system application.

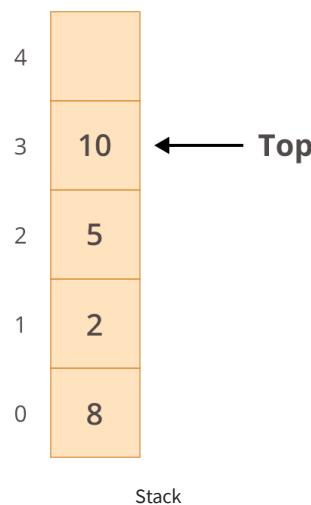
Queues or Stacks

Queues and stacks differ only in the way elements are retrieved. For a queue, you use a **First-In/First-Out** (FIFO) approach. That means that the first element inserted in the list is the first one to be retrieved:



In the diagram above, you can see the **front** and **rear** elements of the queue. When you append new elements to the queue, they'll go to the rear end. When you retrieve elements, they'll be taken from the front of the queue.

For a stack, you use a **Last-In/Fist-Out** (LIFO) approach, meaning that the last element inserted in the list is the first to be retrieved:



In the above diagram you can see that the first element inserted on the stack (index 0) is at the bottom, and the last element inserted is at the top. Since stacks use the LIFO approach, the last element inserted (at the top) will be the first to be retrieved.

Because of the way you insert and retrieve elements from the edges of queues and stacks, linked lists are one of the most convenient ways to implement these data structures. You'll see examples of these implementations later in the article.

Graphs

Graphs can be used to show relationships between objects or to represent different types of networks. For example, a visual representation of a graph—say a directed acyclic graph (DAG)—might look like this:

Directed Acyclic Graph

There are different ways to implement graphs like the above, but one of the most common is to use an **adjacency list**. An adjacency list is, in essence, a list of linked lists where each vertex of the graph is stored alongside a collection of connected vertices:

Vertex

Linked List of Vertices

Vertex	Linked List of Vertices
1	2 → 3 → None
2	4 → None
3	None
4	5 → 6 → None
5	6 → None
6	None

In the table above, each vertex of your graph is listed in the left column. The right column contains a series of linked lists storing the other vertices connected with the corresponding vertex in the left column. This adjacency list could also be represented in code using a dict:

Python

>>>

```
>>> graph = {
...     1: [2, 3, None],
...     2: [4, None],
...     3: [None],
...     4: [5, 6, None],
...     5: [6, None],
...     6: [None]
... }
```

The keys of this dictionary are the source vertices, and the value for each key is a list. This list is usually implemented as a linked list.

Note: In the above example you could avoid storing the `None` values, but we've retained them here for clarity and consistency with later examples.

In terms of both speed and memory, implementing graphs using adjacency lists is very efficient in comparison with, for example, an **adjacency matrix**. That's why linked lists are so useful for graph implementation.

Performance Comparison: Lists vs Linked Lists

In most programming languages, there are clear differences in the way linked lists and arrays are stored in memory. In Python, however, lists are [dynamic arrays](#). That means that the memory usage of both lists and linked lists is very similar.

Further reading: Python's [implementation of dynamic arrays](#) is quite interesting and definitely worth reading about. Make sure to have a look and use that knowledge to stand out at your next company party!

Since the difference in memory usage between lists and linked lists is so insignificant, it's better if you focus on their performance differences when it comes to [time complexity](#).

Insertion and Deletion of Elements

In Python, you can insert elements into a list using `.insert()` or `.append()`. For removing elements from a list, you can use their counterparts: `.remove()` and `.pop()`.

The main difference between these methods is that you use `.insert()` and `.remove()` to insert or remove elements at a specific position in a list, but you use `.append()` and `.pop()` only to insert or remove elements at the end of a list.

Now, something you need to know about Python lists is that inserting or removing elements that are *not* at the end of the list requires some element shifting in the background, making the operation more complex in terms of time spent. You can read the article mentioned above on [how lists are implemented in Python](#) to better understand how the implementation of `.insert()`, `.remove()`, `.append()` and `.pop()` affects their performance.

With all this in mind, even though inserting elements at the end of a list using `.append()` or `.insert()` will have constant time, $O(1)$, when you try inserting an element closer to or at the beginning of the list, the average time complexity will grow along with the size of the list: $O(n)$.

Linked lists, on the other hand, are much more straightforward when it comes to insertion and deletion of elements at the beginning or end of a list, where their time complexity is always constant: $O(1)$.

For this reason, linked lists have a performance advantage over normal lists when implementing a queue (FIFO), in which elements are continuously inserted and removed at the beginning of the list. But they perform similarly to a list when implementing a stack (LIFO), in which elements are inserted and removed at the end of the list.

Retrieval of Elements

When it comes to element lookup, lists perform much better than linked lists. When you know which element you want to access, lists can perform this operation in $O(1)$ time. Trying to do the same with a linked list would take $O(n)$ because you need to traverse the whole list to find the element.

When searching for a specific element, however, both lists and linked lists perform very similarly, with a time complexity of $O(n)$. In both cases, you need to iterate through the entire list to find the element you're looking for.

Introducing `collections.deque`

In Python, there's a specific object in the `collections` module that you can use for linked lists called [`deque`](#) (pronounced “deck”), which stands for **double-ended queue**.

`collections.deque` uses an implementation of a linked list in which you can access, insert, or remove elements from the beginning or end of a list with constant $O(1)$ performance.

How to Use `collections.deque`

There are quite a few [methods](#) that come, by default, with a `deque` object. However, in this article you'll only touch on a few of them, mostly for adding or removing elements.

First, you need to create a linked list. You can use the following piece of code to do that with `deque`:

```
Python >>>
>>> from collections import deque
>>> deque()
deque([])
```

The code above will create an empty linked list. If you want to populate it at creation, then you can give it an **iterable** as input:

```
Python >>>
>>> deque(['a', 'b', 'c'])
deque(['a', 'b', 'c'])

>>> deque('abc')
deque(['a', 'b', 'c'])

>>> deque([{'data': 'a'}, {'data': 'b'}])
deque([{'data': 'a'}, {'data': 'b'}])
```

When initializing a `deque` object, you can pass any iterable as an input, such as a string (also an iterable) or a list of objects.

Now that you know how to create a `deque` object, you can interact with it by adding or removing elements. You can create an abcde linked list and add a new element f like this:

Python

>>>

```
>>> llist = deque("abcde")
>>> llist
deque(['a', 'b', 'c', 'd', 'e'])

>>> llist.append("f")
>>> llist
deque(['a', 'b', 'c', 'd', 'e', 'f'])

>>> llist.pop()
'f'

>>> llist
deque(['a', 'b', 'c', 'd', 'e'])
```

Both append() and pop() add or remove elements from the right side of the linked list. However, you can also use deque to quickly add or remove elements from the left side, or head, of the list:

Python

>>>

```
>>> llist.appendleft("z")
>>> llist
deque(['z', 'a', 'b', 'c', 'd', 'e'])

>>> llist.popleft()
'z'

>>> llist
deque(['a', 'b', 'c', 'd', 'e'])
```

Adding or removing elements from both ends of the list is pretty straightforward using the deque object. Now you're ready to learn how to use collections.deque to implement a queue or a stack.

How to Implement Queues and Stacks

As you learned above, the main difference between a queue and a stack is the way you retrieve elements from each. Next, you'll find out how to use collections.deque to implement both data structures.

Queues

With queues, you want to add values to a list (enqueue), and when the timing is right, you want to remove the element that has been on the list the longest (dequeue). For example, imagine a queue at a trendy and fully booked restaurant. If you were trying to implement a fair system for seating guests, then you'd start by creating a queue and adding people as they arrive:

Python

>>>

```
>>> from collections import deque
>>> queue = deque()
>>> queue
deque([])

>>> queue.append("Mary")
>>> queue.append("John")
>>> queue.append("Susan")
>>> queue
deque(['Mary', 'John', 'Susan'])
```

Now you have Mary, John, and Susan in the queue. Remember that since queues are FIFO, the first person who got into the queue should be the first to get out.

Now imagine some time goes by and a few tables become available. At this stage, you want to remove people from the queue in the correct order. This is how you would do that:

Python

>>>

```
>>> queue.popleft()
'Mary'

>>> queue
deque(['John', 'Susan'])

>>> queue.popleft()
'John'

>>> queue
deque(['Susan'])
```

Every time you call `popleft()`, you remove the head element from the linked list, mimicking a real-life queue.

Stacks

What if you wanted to [create a stack](#) instead? Well, the idea is more or less the same as with the queue. The only difference is that the stack uses the LIFO approach, meaning that the last element to be inserted in the stack should be the first to be removed.

Imagine you're creating a web browser's history functionality in which store every page a user visits so they can go back in time easily. Assume these are the actions a random user takes on their browser:

1. Visits [Real Python's website](#)
2. Navigates to [Pandas: How to Read and Write Files](#)
3. Clicks on a link for [Reading and Writing CSV Files in Python](#)

If you'd like to map this behavior into a stack, then you could do something like this:

Python

>>>

```
>>> from collections import deque
>>> history = deque()

>>> history.appendleft("https://realpython.com/")
>>> history.appendleft("https://realpython.com/pandas-read-write-files/")
>>> history.appendleft("https://realpython.com/python-csv/")
>>> history
deque(['https://realpython.com/python-csv/',
       'https://realpython.com/pandas-read-write-files/',
       'https://realpython.com/'])
```

In this example, you created an empty `history` object, and every time the user visited a new site, you added it to your `history` variable using `appendleft()`. Doing so ensured that each new element was added to the head of the linked list.

Now suppose that after the user read both articles, they wanted to go back to the Real Python home page to pick a new article to read. Knowing that you have a stack and want to remove elements using LIFO, you could do the following:

Python

>>>

```
>>> history.popleft()
'https://realpython.com/python-csv/'

>>> history.popleft()
'https://realpython.com/pandas-read-write-files/'

>>> history
deque(['https://realpython.com/'])
```

There you go! Using `popleft()`, you removed elements from the head of the linked list until you reached the Real Python home page.

From the examples above, you can see how useful it can be to have `collections.deque` in your toolbox, so make sure to use it the next time you have a queue- or stack-based challenge to solve.

Implementing Your Own Linked List

Now that you know how to use `collections.deque` for handling linked lists, you might be wondering why you would ever implement your own linked list in Python. There are a few reasons to do it:

1. Practicing your Python algorithm skills
2. Learning about data structure theory
3. Preparing for job interviews

Feel free to skip this next section if you're not interested in any of the above, or if you already aced implementing your own linked list in Python. Otherwise, it's time to implement some linked lists!

How to Create a Linked List

First things first, create a [class](#) to represent your linked list:

Python

```
class LinkedList:  
    def __init__(self):  
        self.head = None
```

The only information you need to store for a linked list is where the list starts (the head of the list). Next, create another class to represent each node of the linked list:

Python

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

In the above class definition, you can see the two main elements of every single node: `data` and `next`. You can also add a `__repr__` to both classes to have a more helpful representation of the objects:

Python

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
    def __repr__(self):  
        return self.data  
  
class LinkedList:  
    def __init__(self):  
        self.head = None  
  
    def __repr__(self):  
        node = self.head  
        nodes = []  
        while node is not None:  
            nodes.append(node.data)  
            node = node.next  
        nodes.append("None")  
        return " -> ".join(nodes)
```

Have a look at an example of using the above classes to quickly create a linked list with three nodes:

Python

>>>

```
>>> llist = LinkedList()
>>> llist
None

>>> first_node = Node("a")
>>> llist.head = first_node
>>> llist
a -> None

>>> second_node = Node("b")
>>> third_node = Node("c")
>>> first_node.next = second_node
>>> second_node.next = third_node
>>> llist
a -> b -> c -> None
```

By defining a node's data and next values, you can create a linked list quite quickly. These `LinkedList` and `Node` classes are the starting points for our implementation. From now on, it's all about increasing their functionality.

Here's a slight change to the linked list's `__init__()` that allows you to quickly create linked lists with some data:

Python

```
def __init__(self, nodes=None):
    self.head = None
    if nodes is not None:
        node = Node(data=nodes.pop(0))
        self.head = node
        for elem in nodes:
            node.next = Node(data=elem)
            node = node.next
```

With the above modification, creating linked lists to use in the examples below will be much faster.

How to Traverse a Linked List

One of the most common things you will do with a linked list is to **traverse** it. Traversing means going through every single node, starting with the head of the linked list and ending on the node that has a next value of `None`.

Traversing is just a fancier way to say iterating. So, with that in mind, create an `__iter__` to add the same behavior to linked lists that you would expect from a normal list:

Python

```
def __iter__(self):
    node = self.head
    while node is not None:
        yield node
        node = node.next
```

The method above goes through the list and yields every single node. The most important thing to remember about this `__iter__` is that you need to always validate that the current node is not `None`. When that condition is `True`, it means you've reached the end of your linked list.

After yielding the current node, you want to move to the next node on the list. That's why you add `node = node.next`. Here's an example of traversing a random list and printing each node:

Python

>>>

```
>>> llist = LinkedList(["a", "b", "c", "d", "e"])
>>> llist
a -> b -> c -> d -> e -> None

>>> for node in llist:
...     print(node)
a
b
c
d
e
```

In other articles, you might see the traversing defined into a specific method called `traverse()`. However, using Python's [built-in methods](#) to achieve said behavior makes this linked list implementation a bit more [Pythonic](#).

How to Insert a New Node

There are different ways to insert new nodes into a linked list, each with its own implementation and level of complexity. That's why you'll see them split into specific methods for inserting at the beginning, end, or between nodes of a list.

Inserting at the Beginning

Inserting a new node at the beginning of a list is probably the most straightforward insertion since you don't have to traverse the whole list to do it. It's all about creating a new node and then pointing the head of the list to it.

Have a look at the following implementation of `add_first()` for the class `LinkedList`:

Python

```
def add_first(self, node):
    node.next = self.head
    self.head = node
```

In the above example, you're setting `self.head` as the next reference of the new node so that the new node points to the old `self.head`. After that, you need to state that the new head of the list is the inserted node.

Here's how it behaves with a sample list:

Python

>>>

```
>>> llist = LinkedList()
>>> llist
None

>>> llist.add_first(Node("b"))
>>> llist
b -> None

>>> llist.add_first(Node("a"))
>>> llist
a -> b -> None
```

As you can see, `add_first()` always adds the node to the head of the list, even if the list was empty before.

Inserting at the End

Inserting a new node at the end of the list forces you to traverse the whole linked list first and to add the new node when you reach the end. You can't just append to the end as you would with a normal list because in a linked list you don't know which node is last.

Here's an example implementation of a function for inserting a node to the end of a linked list:

Python

```
def add_last(self, node):
    if not self.head:
        self.head = node
        return
    for current_node in self:
        pass
    current_node.next = node
```

First, you want to traverse the whole list until you reach the end (that is, until the [for loop](#) raises a `StopIteration` exception). Next, you want to set the `current_node` as the last node on the list. Finally, you want to add the new node as the next value of that `current_node`.

Here's an example of `add_last()` in action:

Python

>>>

```
>>> llist = LinkedList(["a", "b", "c", "d"])
>>> llist
a -> b -> c -> d -> None

>>> llist.add_last(Node("e"))
>>> llist
a -> b -> c -> d -> e -> None

>>> llist.add_last(Node("f"))
>>> llist
a -> b -> c -> d -> e -> f -> None
```

In the code above, you start by creating a list with four values (a, b, c, and d). Then, when you add new nodes using `add_last()`, you can see that the nodes are always appended to the end of the list.

Inserting Between Two Nodes

Inserting between two nodes adds yet another layer of complexity to the linked list's already complex insertions because there are two different approaches that you can use:

1. Inserting *after* an existing node
2. Inserting *before* an existing node

It might seem weird to split these into two methods, but linked lists behave differently than normal lists, and you need a different implementation for each case.

Here's a method that adds a node *after* an existing node with a specific data value:

Python

```
def add_after(self, target_node_data, new_node):
    if not self.head:
        raise Exception("List is empty")

    for node in self:
        if node.data == target_node_data:
            new_node.next = node.next
            node.next = new_node
            return

    raise Exception("Node with data '%s' not found" % target_node_data)
```

In the above code, you're traversing the linked list looking for the node with data indicating where you want to insert a new node. When you find the node you're looking for, you'll insert the new node immediately after it and rewire the next reference to maintain the consistency of the list.

The only exceptions are if the list is empty, making it impossible to insert a new node after an existing node, or if the list does not contain the value you're searching for. Here are a few examples of how `add_after()` behaves:

Python

>>>

```
>>> llist = LinkedList()
>>> llist.add_after("a", Node("b"))
Exception: List is empty

>>> llist = LinkedList(["a", "b", "c", "d"])
>>> llist
a -> b -> c -> d -> None

>>> llist.add_after("c", Node("cc"))
>>> llist
a -> b -> c -> cc -> d -> None

>>> llist.add_after("f", Node("g"))
Exception: Node with data 'f' not found
```

Trying to use `add_after()` on an empty list results in an [exception](#). The same happens when you try to add after a nonexistent node. Everything else works as expected.

Now, if you want to implement `add_before()`, then it will look something like this:

Python

```
1 def add_before(self, target_node_data, new_node):
2     if not self.head:
3         raise Exception("List is empty")
4
5     if self.head.data == target_node_data:
6         return self.add_first(new_node)
7
8     prev_node = self.head
9     for node in self:
10        if node.data == target_node_data:
11            prev_node.next = new_node
12            new_node.next = node
13            return
14        prev_node = node
15
16    raise Exception("Node with data '%s' not found" % target_node_data)
```

There are a few things to keep in mind while implementing the above. First, as with `add_after()`, you want to make sure to raise an exception if the linked list is empty (line 2) or the node you're looking for is not present (line 16).

Second, if you're trying to add a new node before the head of the list (line 5), then you can reuse `add_first()` because the node you're inserting will be the new head of the list.

Finally, for any other case (line 9), you should keep track of the last-checked node using the `prev_node` variable. Then, when you find the target node, you can use that `prev_node` variable to rewire the next values.

Once again, an example is worth a thousand words:

Python

>>>

```
>>> llist = LinkedList()
>>> llist.add_before("a", Node("a"))
Exception: List is empty

>>> llist = LinkedList(["b", "c"])
>>> llist
b -> c -> None

>>> llist.add_before("b", Node("a"))
>>> llist
a -> b -> c -> None

>>> llist.add_before("b", Node("aa"))
>>> llist.add_before("c", Node("bb"))
>>> llist
a -> aa -> b -> bb -> c -> None

>>> llist.add_before("n", Node("m"))
Exception: Node with data 'n' not found
```

With `add_before()`, you now have all the methods you need to insert nodes anywhere you'd like in your list.

How to Remove a Node

To remove a node from a linked list, you first need to traverse the list until you find the node you want to remove. Once you find the target, you want to link its previous and next nodes. This re-linking is what removes the target node from the list.

That means you need to keep track of the previous node as you traverse the list. Have a look at an example implementation:

Python

```
1 def remove_node(self, target_node_data):
2     if not self.head:
3         raise Exception("List is empty")
4
5     if self.head.data == target_node_data:
6         self.head = self.head.next
7         return
8
9     previous_node = self.head
10    for node in self:
11        if node.data == target_node_data:
12            previous_node.next = node.next
13            return
14        previous_node = node
15
16    raise Exception("Node with data '%s' not found" % target_node_data)
```

In the above code, you first check that your list is not empty (line 2). If it is, then you raise an exception. After that, you check if the node to be removed is the current head of the list (line 5). If it is, then you want the next node in the list to become the new head.

If none of the above happens, then you start traversing the list looking for the node to be removed (line 10). If you find it, then you need to update its previous node to point to its next node, automatically removing the found node from the list. Finally, if you traverse the whole list without finding the node to be removed (line 16), then you raise an exception.

Notice how in the above code you use `previous_node` to keep track of the, well, previous node. Doing so ensures that the whole process will be much more straightforward when you find the right node to be deleted.

Here's an example using a list:

Python

>>>

```
>>> llist = LinkedList()
>>> llist.remove_node("a")
Exception: List is empty

>>> llist = LinkedList(["a", "b", "c", "d", "e"])
>>> llist
a -> b -> c -> d -> e -> None

>>> llist.remove_node("a")
>>> llist
b -> c -> d -> e -> None

>>> llist.remove_node("e")
>>> llist
b -> c -> d -> None

>>> llist.remove_node("c")
>>> llist
b -> d -> None

>>> llist.remove_node("a")
Exception: Node with data 'a' not found
```

That's it! You now know how to implement a linked list and all of the main methods for traversing, inserting, and removing nodes. If you feel comfortable with what you've learned and you're craving more, then feel free to pick one of the challenges below:

1. Create a method to retrieve an element from a specific position: `get(i)` or even `llist[i]`.
2. Create a method to reverse the linked list: `llist.reverse()`.
3. Create a `Queue()` object inheriting this article's linked list with `enqueue()` and `dequeue()` methods.

Apart from being great practice, doing some extra challenges on your own is an effective way to assimilate all the knowledge you've gained. If you want to get a head start by reusing all the source code from this article, then you can download everything you need at the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about linked lists in this tutorial.

Using Advanced Linked Lists

Until now, you've been learning about a specific type of linked list called **singly linked lists**. But there are more types of linked lists that can be used for slightly different purposes.

How to Use Doubly Linked Lists

Doubly linked lists are different from singly linked lists in that they have two references:

1. The previous field references the previous node.
2. The next field references the next node.

The end result looks like this:

Node (Doubly Linked List)

If you wanted to implement the above, then you could make some changes to your existing `Node` class in order to include a `previous` field:

Python

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
        self.previous = None
```

This kind of implementation would allow you to traverse a list in both directions instead of only traversing using next. You could use next to go forward and previous to go backward.

In terms of structure, this is how a doubly linked list would look:

Doubly Linked List

You learned earlier that `collections.deque` uses a linked list as part of its data structure. This is the kind of [linked list it uses](#). With doubly linked lists, deque is capable of inserting or deleting elements from both ends of a queue with constant $O(1)$ performance.

How to Use Circular Linked Lists

Circular linked lists are a type of linked list in which the last node points back to the head of the list instead of pointing to None. This is what makes them circular. Circular linked lists have quite a few interesting use cases:

- Going around each player's turn in a multiplayer game
- Managing the application life cycle of a given operating system
- Implementing a [Fibonacci heap](#)

This is what a circular linked list looks like:

Circular Linked List

One of the advantages of circular linked lists is that you can traverse the whole list starting at any node. Since the last node points to the head of the list, you need to make sure that you stop traversing when you reach the starting point. Otherwise, you'll end up in an infinite loop.

In terms of implementation, circular linked lists are very similar to singly linked list. The only difference is that you can define the starting point when you traverse the list:

Python

```
class CircularLinkedList:
    def __init__(self):
        self.head = None

    def traverse(self, starting_point=None):
        if starting_point is None:
            starting_point = self.head
        node = starting_point
        while node is not None and (node.next != starting_point):
            yield node
            node = node.next
        yield node

    def print_list(self, starting_point=None):
        nodes = []
        for node in self.traverse(starting_point):
            nodes.append(str(node))
        print(" -> ".join(nodes))
```

Traversing the list now receives an additional argument, `starting_point`, that is used to define the start and (because the list is circular) the end of the iteration process. Apart from that, much of the code is the same as what we had in our `LinkedList` class.

To wrap up with a final example, have a look at how this new type of list behaves when you give it some data:

Python

>>>

```
>>> circular_llist = CircularLinkedList()
>>> circular_llist.print_list()
None

>>> a = Node("a")
>>> b = Node("b")
>>> c = Node("c")
>>> d = Node("d")
>>> a.next = b
>>> b.next = c
>>> c.next = d
>>> d.next = a
>>> circular_llist.head = a
>>> circular_llist.print_list()
a -> b -> c -> d

>>> circular_llist.print_list(b)
b -> c -> d -> a

>>> circular_llist.print_list(d)
d -> a -> b -> c
```

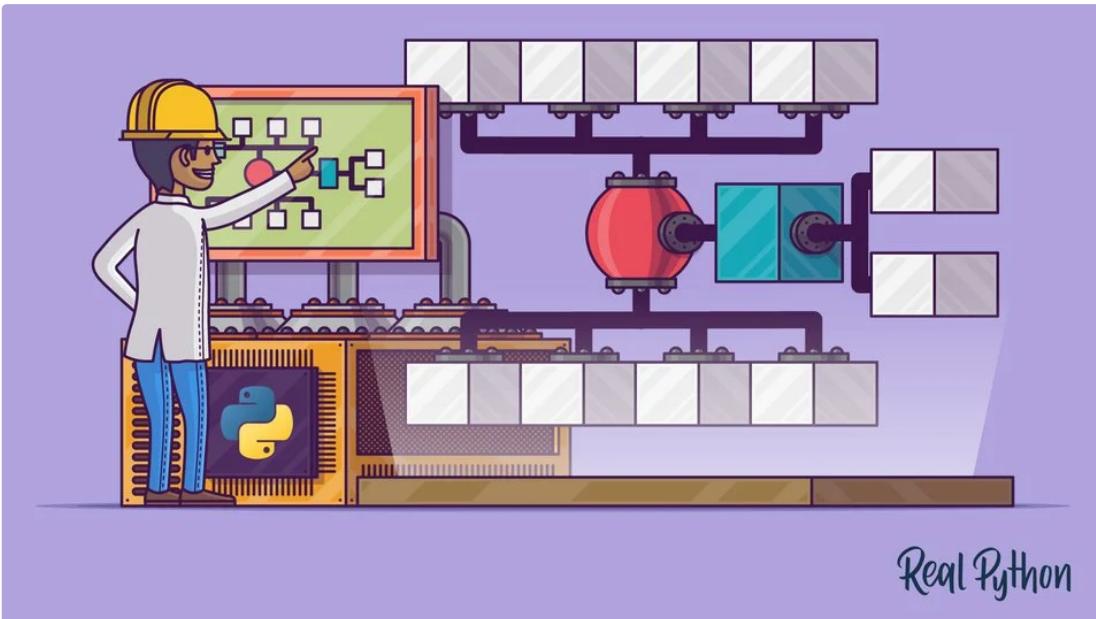
There you have it! You'll notice that you no longer have the `None` while traversing the list. That's because there is no specific end to a circular list. You can also see that choosing different starting nodes will render slightly different representations of the same list.

Conclusion

In this article, you learned quite a few things! The most important are:

- What linked lists are and when you should use them
- How to use `collections.deque` to implement queues and stacks
- How to implement your own linked list and node classes, plus relevant methods
- What the other types of linked lists are and what they can be used for

If you want to learn more about linked lists, then check out [Vaidehi Joshi's Medium post](#) for a nice visual explanation. If you're interested in a more in-depth guide, then the [Wikipedia article](#) is quite thorough. Finally, if you're curious about the reasoning behind the current implementation of `collections.deque`, then check out [Raymond Hettinger's thread](#).



Real Python

Common Python Data Structures (Guide)

by [Dan Bader](#) ⏲ Aug 26, 2020 🗣 3 Comments 🛍 [basics](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Dictionaries, Maps, and Hash Tables](#)
 - [dict: Your Go-To Dictionary](#)
 - [collections.OrderedDict: Remember the Insertion Order of Keys](#)
 - [collections.defaultdict: Return Default Values for Missing Keys](#)
 - [collections.ChainMap: Search Multiple Dictionaries as a Single Mapping](#)
 - [types.MappingProxyType: A Wrapper for Making Read-Only Dictionaries](#)
 - [Dictionaries in Python: Summary](#)
- [Array Data Structures](#)
 - [list: Mutable Dynamic Arrays](#)
 - [tuple: Immutable Containers](#)
 - [array.array: Basic Typed Arrays](#)
 - [str: Immutable Arrays of Unicode Characters](#)
 - [bytes: Immutable Arrays of Single Bytes](#)
 - [bytearray: Mutable Arrays of Single Bytes](#)
 - [Arrays in Python: Summary](#)
- [Records, Structs, and Data Transfer Objects](#)
 - [dict: Simple Data Objects](#)
 - [tuple: Immutable Groups of Objects](#)
 - [Write a Custom Class: More Work, More Control](#)
 - [dataclasses.dataclass: Python 3.7+ Data Classes](#)
 - [collections.namedtuple: Convenient Data Objects](#)
 - [typing.NamedTuple: Improved Namedtuples](#)
 - [struct.Struct: Serialized C Structs](#)
 - [types.SimpleNamespace: Fancy Attribute Access](#)
 - [Records, Structs, and Data Objects in Python: Summary](#)
- [Sets and Multisets](#)
 - [set: Your Go-To Set](#)
 - [frozenset: Immutable Sets](#)
 - [collections.Counter: Multisets](#)
- [Sets and Multisets in Python: Summary](#)
- [Stacks \(LIFOs\)](#)

- [list: Simple, Built-In Stacks](#)
- [collections.deque: Fast and Robust Stacks](#)
- [queue.LifoQueue: Locking Semantics for Parallel Computing](#)
- [Stack Implementations in Python: Summary](#)
- [Queues \(FIFOs\)](#)
 - [list: Terribly Sloooow Queues](#)
 - [collections.deque: Fast and Robust Queues](#)
 - [queue.Queue: Locking Semantics for Parallel Computing](#)
 - [multiprocessing.Queue: Shared Job Queues](#)
 - [Queues in Python: Summary](#)
- [Priority Queues](#)
 - [list: Manually Sorted Queues](#)
 - [heapq: List-Based Binary Heaps](#)
 - [queue.PriorityQueue: Beautiful Priority Queues](#)
 - [Priority Queues in Python: Summary](#)
- [Conclusion: Python Data Structures](#)



Data structures are the fundamental constructs around which you build your programs. Each data structure provides a particular way of organizing data so it can be accessed efficiently, depending on your use case. Python ships with an extensive set of data structures in its [standard library](#).

However, Python's naming convention doesn't provide the same level of clarity that you'll find in other languages. In Java, a list isn't just a `list`—it's either a `LinkedList` or an `ArrayList`. Not so in Python. Even experienced Python developers sometimes wonder whether the built-in `list` type is implemented as a [linked list](#) or a dynamic array.

In this tutorial, you'll learn:

- Which common **abstract data types** are built into the Python standard library
- How the most common abstract data types map to Python's **naming scheme**
- How to put abstract data types to **practical use** in various algorithms

Note: This tutorial is adapted from the chapter “Common Data Structures in Python” in [Python Tricks: The Book](#). If you enjoy what you read below, then be sure to check out [the rest of the book](#).

Free Bonus: [Click here to get access to a chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

Dictionaries, Maps, and Hash Tables

In Python, [dictionaries](#) (or **dicts** for short) are a central data structure. Dicts store an arbitrary number of objects, each identified by a unique dictionary **key**.

Dictionaries are also often called **maps**, **hashmaps**, **lookup tables**, or **associative arrays**. They allow for the efficient lookup, insertion, and deletion of any object associated with a given key.

Phone books make a decent real-world analog for dictionary objects. They allow you to quickly retrieve the information (phone number) associated with a given key (a person's name). Instead of having to read a phone book front to back to find someone's number, you can jump more or less directly to a name and look up the associated information.

This analogy breaks down somewhat when it comes to *how* the information is organized to allow for fast lookups. But the fundamental performance characteristics hold. Dictionaries allow you to quickly find the information associated with a given key.

Dictionaries are one of the most important and frequently used data structures in computer science. So, how does Python handle dictionaries? Let's take a tour of the dictionary implementations available in core Python and the

Python standard library.

dict: Your Go-To Dictionary

Because dictionaries are so important, Python features a robust dictionary implementation that's built directly into the core language: the [dict](#) data type.

Python also provides some useful **syntactic sugar** for working with dictionaries in your programs. For example, the curly-brace ({} {}) dictionary expression syntax and [dictionary comprehensions](#) allow you to conveniently define new dictionary objects:

```
Python >>>
>>> phonebook = {
...     "bob": 7387,
...     "alice": 3719,
...     "jack": 7052,
... }

>>> squares = {x: x * x for x in range(6)}

>>> phonebook["alice"]
3719

>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

There are some restrictions on which objects can be used as valid keys.

Python's dictionaries are indexed by keys that can be of any [hashable](#) type. A **hashable** object has a hash value that never changes during its lifetime (see `__hash__`), and it can be compared to other objects (see `__eq__`). Hashable objects that compare as equal must have the same hash value.

[Immutable types](#) like [strings](#) and [numbers](#) are hashable and work well as dictionary keys. You can also use [tuple objects](#) as dictionary keys as long as they contain only hashable types themselves.

For most use cases, Python's built-in dictionary implementation will do everything you need. Dictionaries are highly optimized and underlie many parts of the language. For example, [class attributes](#) and variables in a [stack frame](#) are both stored internally in dictionaries.

Python dictionaries are based on a well-tested and finely tuned hash table implementation that provides the performance characteristics you'd expect: $O(1)$ time complexity for lookup, insert, update, and delete operations in the average case.

There's little reason not to use the standard `dict` implementation included with Python. However, specialized third-party dictionary implementations exist, such as [skip lists](#) or [B-tree-based](#) dictionaries.

Besides plain `dict` objects, Python's standard library also includes a number of specialized dictionary implementations. These specialized dictionaries are all based on the built-in dictionary class (and share its performance characteristics) but also include some additional convenience features.

Let's take a look at them.

`collections.OrderedDict`: Remember the Insertion Order of Keys

Python includes a specialized `dict` subclass that remembers the insertion order of keys added to it:

[collections.OrderedDict](#).

Note: `OrderedDict` is not a built-in part of the core language and must be imported from the `collections` module in the standard library.

While standard `dict` instances preserve the insertion order of keys in CPython 3.6 and above, this was simply a [side effect](#) of the CPython implementation and was not defined in the language spec until Python 3.7. So, if key order is important for your algorithm to work, then it's best to communicate this clearly by explicitly using the `OrderedDict`.

class:

```
Python >>>
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)

>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3)])

>>> d["four"] = 4
>>> d
OrderedDict([('one', 1), ('two', 2),
              ('three', 3), ('four', 4)])

>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

Until Python 3.8, you couldn't iterate over dictionary items in reverse order using `reversed()`. Only `OrderedDict` instances offered that functionality. Even in Python 3.8, `dict` and `OrderedDict` objects aren't exactly the same. `OrderedDict` instances have a [_move_to_end\(\) method](#) that is unavailable on plain `dict` instance, as well as a more customizable [_popitem\(\) method](#) than the one plain `dict` instances.

`collections.defaultdict`: Return Default Values for Missing Keys

The `defaultdict` class is another dictionary subclass that accepts a callable in its constructor whose return value will be used if a requested key cannot be found.

This can save you some typing and make your intentions clearer as compared to using `get()` or catching a [KeyError exception](#) in regular dictionaries:

```
Python >>>
>>> from collections import defaultdict
>>> dd = defaultdict(list)

>>> # Accessing a missing key creates it and
>>> # initializes it using the default factory,
>>> # i.e. list() in this example:
>>> dd["dogs"].append("Rufus")
>>> dd["dogs"].append("Kathrin")
>>> dd["dogs"].append("Mr Snuffles")

>>> dd["dogs"]
['Rufus', 'Kathrin', 'Mr Snuffles']
```

`collections.ChainMap`: Search Multiple Dictionaries as a Single Mapping

The `collections.ChainMap` data structure groups multiple dictionaries into a single mapping. Lookups search the underlying mappings one by one until a key is found. Insertions, updates, and deletions only affect the first mapping added to the chain:

```
Python >>>
>>> from collections import ChainMap
>>> dict1 = {"one": 1, "two": 2}
>>> dict2 = {"three": 3, "four": 4}
>>> chain = ChainMap(dict1, dict2)

>>> chain
ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four': 4})

>>> # ChainMap searches each collection in the chain
>>> # from left to right until it finds the key (or fails):
>>> chain["three"]
3
>>> chain["one"]
1
```

```
>>> chain["missing"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'missing'
```

types.MappingProxyType: A Wrapper for Making Read-Only Dictionaries

[MappingProxyType](#) is a wrapper around a standard dictionary that provides a read-only view into the wrapped dictionary's data. This class was added in Python 3.3 and can be used to create immutable proxy versions of dictionaries.

MappingProxyType can be helpful if, for example, you'd like to return a dictionary carrying internal state from a class or module while discouraging write access to this object. Using MappingProxyType allows you to put these restrictions in place without first having to create a full copy of the dictionary:

```
Python >>>

>>> from types import MappingProxyType
>>> writable = {"one": 1, "two": 2}
>>> read_only = MappingProxyType(writable)

>>> # The proxy is read-only:
>>> read_only["one"]
1
>>> read_only["one"] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment

>>> # Updates to the original are reflected in the proxy:
>>> writable["one"] = 42
>>> read_only
mappingproxy({'one': 42, 'two': 2})
```

Dictionaries in Python: Summary

All the Python dictionary implementations listed in this tutorial are valid implementations that are built into the Python standard library.

If you're looking for a general recommendation on which mapping type to use in your programs, I'd point you to the built-in `dict` data type. It's a versatile and optimized hash table implementation that's built directly into the core language.

I would recommend that you use one of the other data types listed here only if you have special requirements that go beyond what's provided by `dict`.

All the implementations are valid options, but your code will be clearer and easier to maintain if it relies on standard Python dictionaries most of the time.

Array Data Structures

An **array** is a fundamental data structure available in most programming languages, and it has a wide range of uses across different algorithms.

In this section, you'll take a look at array implementations in Python that use only core language features or functionality that's included in the Python standard library. You'll see the strengths and weaknesses of each approach so you can decide which implementation is right for your use case.

But before we jump in, let's cover some of the basics first. How do arrays work, and what are they used for? Arrays consist of fixed-size data records that allow each element to be efficiently located based on its index:

Because arrays store information in adjoining blocks of memory, they're considered **contiguous** data structures (as opposed to **linked** data structures like linked lists, for example).

A real-world analogy for an array data structure is a parking lot. You can look at the parking lot as a whole and treat it as a single object, but inside the lot there are parking spots indexed by a unique number. Parking spots are containers for vehicles—each parking spot can either be empty or have a car, a motorbike, or some other vehicle parked on it.

But not all parking lots are the same. Some parking lots may be restricted to only one type of vehicle. For example, a motor home parking lot wouldn't allow bikes to be parked on it. A restricted parking lot corresponds to a **typed** array data structure that allows only elements that have the same data type stored in them.

Performance-wise, it's very fast to look up an element contained in an array given the element's index. A proper array implementation guarantees a constant $O(1)$ access time for this case.

Python includes several array-like data structures in its standard library that each have slightly different characteristics. Let's take a look.

list: Mutable Dynamic Arrays

Lists are a part of the core Python language. Despite their name, Python's lists are implemented as **dynamic arrays** behind the scenes.

This means a list allows elements to be added or removed, and the list will automatically adjust the backing store that holds these elements by allocating or releasing memory.

Python lists can hold arbitrary elements—everything is an object in Python, including functions. Therefore, you can mix and match different kinds of data types and store them all in a single list.

This can be a powerful feature, but the downside is that supporting multiple data types at the same time means that data is generally less tightly packed. As a result, the whole structure takes up more space:

```
Python >>>
>>> arr = ["one", "two", "three"]
>>> arr[0]
'one'

>>> # Lists have a nice repr:
>>> arr
['one', 'two', 'three']

>>> # Lists are mutable:
>>> arr[1] = "hello"
>>> arr
['one', 'hello', 'three']

>>> del arr[1]
>>> arr
['one', 'three']

>>> # Lists can hold arbitrary data types:
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

tuple: Immutable Containers

Just like lists, [tuples](#) are part of the Python core language. Unlike lists, however, Python's tuple objects are immutable. This means elements can't be added or removed dynamically—all elements in a tuple must be defined at creation time.

Tuples are another data structure that can hold elements of arbitrary data types. Having this flexibility is powerful, but again, it also means that data is less tightly packed than it would be in a typed array:

```
Python >>>
>>> arr = ("one", "two", "three")
>>> arr[0]
'one'

>>> # Tuples have a nice repr:
>>> arr
('one', 'two', 'three')

>>> # Tuples are immutable:
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> del arr[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion

>>> # Tuples can hold arbitrary data types:
>>> # (Adding elements creates a copy of the tuple)
>>> arr + (23,)
('one', 'two', 'three', 23)
```

array.array: Basic Typed Arrays

Python's `array` module provides space-efficient storage of basic C-style data types like bytes, 32-bit integers, floating-point numbers, and so on.

Arrays created with the [array.array](#) class are mutable and behave similarly to lists except for one important difference: they're **typed arrays** constrained to a single data type.

Because of this constraint, `array.array` objects with many elements are more space efficient than lists and tuples. The elements stored in them are tightly packed, and this can be useful if you need to store many elements of the same type.

Also, arrays support many of the same methods as regular lists, and you might be able to use them as a drop-in replacement without requiring other changes to your application code.

```
Python >>>
>>> import array
>>> arr = array.array("f", (1.0, 1.5, 2.0, 2.5))
>>> arr[1]
1.5

>>> # Arrays have a nice repr:
>>> arr
array('f', [1.0, 1.5, 2.0, 2.5])

>>> # Arrays are mutable:
>>> arr[1] = 23.0
>>> arr
array('f', [1.0, 23.0, 2.0, 2.5])

>>> del arr[1]
>>> arr
```

```

array('f', [1.0, 2.0, 2.5])

>>> arr.append(42.0)
>>> arr
array('f', [1.0, 2.0, 2.5, 42.0])

>>> # Arrays are "typed":
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be real number, not str

```

str: Immutable Arrays of Unicode Characters

Python 3.x uses `str` objects to store textual data as immutable sequences of [Unicode characters](#). Practically speaking, that means a `str` is an immutable array of characters. Oddly enough, it's also a **recursive** data structure—each character in a string is itself a `str` object of length 1.

String objects are space efficient because they're tightly packed and they specialize in a single data type. If you're storing Unicode text, then you should use a string.

Because strings are immutable in Python, modifying a string requires creating a modified copy. The closest equivalent to a mutable string is storing individual characters inside a list:

```

Python >>>

>>> arr = "abcd"
>>> arr[1]
'b'

>>> arr
'abcd'

>>> # Strings are immutable:
>>> arr[1] = "e"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> del arr[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion

>>> # Strings can be unpacked into a list to
>>> # get a mutable representation:
>>> list("abcd")
['a', 'b', 'c', 'd']
>>> "".join(list("abcd"))
'abcd'

>>> # Strings are recursive data structures:
>>> type("abc")
<class 'str'>
>>> type("abc"[0])
<class 'str'>

```

bytes: Immutable Arrays of Single Bytes

[bytes](#) objects are immutable sequences of single bytes, or integers in the range $0 \leq x \leq 255$. Conceptually, bytes objects are similar to [str](#) objects, and you can also think of them as immutable arrays of bytes.

Like strings, bytes have their own literal syntax for creating objects and are space efficient. bytes objects are immutable, but unlike strings, there's a dedicated mutable byte array data type called [bytearray](#) that they can be unpacked into:

```
Python >>>
>>> arr = bytes((0, 1, 2, 3))
>>> arr[1]
1

>>> # Bytes literals have their own syntax:
>>> arr
b'\x00\x01\x02\x03'
>>> arr = b"\x00\x01\x02\x03"

>>> # Only valid `bytes` are allowed:
>>> bytes((0, 300))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)

>>> # Bytes are immutable:
>>> arr[1] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment

>>> del arr[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object doesn't support item deletion
```

bytearray: Mutable Arrays of Single Bytes

The [bytearray](#) type is a mutable sequence of integers in the range $0 \leq x \leq 255$. The bytearray object is closely related to the bytes object, with the main difference being that a bytearray can be modified freely—you can overwrite elements, remove existing elements, or add new ones. The bytearray object will grow and shrink accordingly.

A bytearray can be converted back into immutable bytes objects, but this involves copying the stored data in full—a slow operation taking $O(n)$ time:

```
Python >>>
>>> arr = bytearray((0, 1, 2, 3))
>>> arr[1]
1

>>> # The bytearray repr:
>>> arr
bytearray(b'\x00\x01\x02\x03')

>>> # Bytearrays are mutable:
>>> arr[1] = 23
>>> arr
bytearray(b'\x00\x17\x02\x03')

>>> arr[1]
23

>>> # Bytearrays can grow and shrink in size:
>>> del arr[1]
>>> arr
bytearray(b'\x00\x02\x03')

>>> arr.append(42)
>>> arr
bytearray(b'\x00\x02\x03\x2a')

>>> # Bytearrays can only hold `bytes`:
>>> # (integers in the range 0 <= x <= 255)
>>> arr[1] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer

>>> arr[1] = 300
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: byte must be in range(0, 256)

>>> # Bytearrays can be converted back into bytes objects:
>>> # (This will copy the data)
>>> bytes(arr)
b'\x00\x02\x03\x2a'
```

Arrays in Python: Summary

There are a number of built-in data structures you can choose from when it comes to implementing arrays in Python. In this section, you've focused on core language features and data structures included in the standard library.

If you're willing to go beyond the Python standard library, then third-party packages like [NumPy](#) and [pandas](#) offer a wide range of fast array implementations for scientific computing and data science.

If you want to restrict yourself to the array data structures included with Python, then here are a few guidelines:

- If you need to store arbitrary objects, potentially with mixed data types, then use a `list` or a `tuple`, depending on whether or not you want an immutable data structure.
- If you have numeric (integer or floating-point) data and tight packing and performance is important, then try out `array.array`.
- If you have textual data represented as Unicode characters, then use Python's built-in `str`. If you need a mutable string-like data structure, then use a `list` of characters.
- If you want to store a contiguous block of bytes, then use the immutable `bytes` type or a `bytearray` if you need a mutable data structure.

In most cases, I like to start out with a simple `list`. I'll only specialize later on if performance or storage space becomes an issue. Most of the time, using a general-purpose array data structure like `list` gives you the fastest development speed and the most programming convenience.

I've found that this is usually much more important in the beginning than trying to squeeze out every last drop of performance right from the start.

Records, Structs, and Data Transfer Objects

Compared to arrays, **record** data structures provide a fixed number of fields. Each field can have a name and may also have a different type.

In this section, you'll see how to implement records, structs, and plain old data objects in Python using only built-in data types and classes from the standard library.

Note: I'm using the definition of a record loosely here. For example, I'm also going to discuss types like Python's built-in `tuple` that may or may not be considered records in a strict sense because they don't provide named fields.

Python offers several data types that you can use to implement records, structs, and data transfer objects. In this section, you'll get a quick look at each implementation and its unique characteristics. At the end, you'll find a summary and a decision-making guide that will help you make your own picks.

Note: This tutorial is adapted from the chapter "Common Data Structures in Python" in [Python Tricks: The Book](#). If you enjoy what you're reading, then be sure to check out [the rest of the book](#).

Alright, let's get started!

dict: Simple Data Objects

As mentioned [previously](#), Python dictionaries store an arbitrary number of objects, each identified by a unique key. Dictionaries are also often called **maps** or **associative arrays** and allow for efficient lookup, insertion, and deletion of any object associated with a given key.

Using dictionaries as a record data type or data object in Python is possible. Dictionaries are easy to create in Python as they have their own syntactic sugar built into the language in the form of **dictionary literals**. The dictionary syntax is concise and quite convenient to type.

Data objects created using dictionaries are mutable, and there's little protection against misspelled field names as fields can be added and removed freely at any time. Both of these properties can introduce surprising bugs, and there's always a trade-off to be made between convenience and error resilience:

```
Python >>> car1 = {  
...     "color": "red",  
...     "mileage": 3812.4,  
...     "automatic": True,  
... }  
>>> car2 = {  
...     "color": "blue",  
...     "mileage": 40231,  
...     "automatic": False,  
... }  
  
>>> # Dicts have a nice repr:  
>>> car2  
{'color': 'blue', 'automatic': False, 'mileage': 40231}  
  
>>> # Get mileage:  
>>> car2["mileage"]  
40231  
  
>>> # Dicts are mutable:  
>>> car2["mileage"] = 12  
>>> car2["windshield"] = "broken"  
>>> car2  
{'windshield': 'broken', 'color': 'blue',  
    'automatic': False, 'mileage': 12}  
  
>>> # No protection against wrong field names,  
>>> # or missing/extraneous fields:  
>>> car3 = {  
...     "colr": "green",  
...     "automatic": False,  
...     "windshield": "broken",  
... }
```

tuple: Immutable Groups of Objects

Python's tuples are a straightforward data structure for grouping arbitrary objects. Tuples are immutable—they can't be modified once they've been created.

Performance-wise, tuples take up [slightly less memory](#) than [lists in CPython](#), and they're also faster to construct.

As you can see in the bytecode disassembly below, constructing a tuple constant takes a single `LOAD_CONST` opcode, while constructing a list object with the same contents requires several more operations:

```
Python >>>
```

```

>>> import dis
>>> dis.dis(compile("(23, 'a', 'b', 'c')", "", "eval"))
  0 LOAD_CONST      4 ((23, "a", "b", "c"))
  3 RETURN_VALUE

>>> dis.dis(compile("[23, 'a', 'b', 'c']", "", "eval"))
  0 LOAD_CONST      0 (23)
  3 LOAD_CONST      1 ('a')
  6 LOAD_CONST      2 ('b')
  9 LOAD_CONST      3 ('c')
 12 BUILD_LIST      4
 15 RETURN_VALUE

```

However, you shouldn't place too much emphasis on these differences. In practice, the performance difference will often be negligible, and trying to squeeze extra performance out of a program by switching from lists to tuples will likely be the wrong approach.

A potential downside of plain tuples is that the data you store in them can only be pulled out by accessing it through integer indexes. You can't give names to individual properties stored in a tuple. This can impact code readability.

Also, a tuple is always an ad-hoc structure: it's difficult to ensure that two tuples have the same number of fields and the same properties stored in them.

This makes it easy to introduce slip-of-the-mind bugs, such as mixing up the field order. Therefore, I would recommend that you keep the number of fields stored in a tuple as low as possible:

```

Python >>>
>>> # Fields: color, mileage, automatic
>>> car1 = ("red", 3812.4, True)
>>> car2 = ("blue", 40231.0, False)

>>> # Tuple instances have a nice repr:
>>> car1
('red', 3812.4, True)
>>> car2
('blue', 40231.0, False)

>>> # Get mileage:
>>> car2[1]
40231.0

>>> # Tuples are immutable:
>>> car2[1] = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> # No protection against missing or extra fields
>>> # or a wrong order:
>>> car3 = (3431.5, "green", True, "silver")

```

Write a Custom Class: More Work, More Control

Classes allow you to define reusable blueprints for data objects to ensure each object provides the same set of fields.

Using regular Python classes as record data types is feasible, but it also takes manual work to get the convenience features of other implementations. For example, adding new fields to the `__init__` constructor is verbose and takes time.

Also, the default string representation for objects instantiated from custom classes isn't very helpful. To fix that, you may have to add your own `__repr__` method, which again is usually quite verbose and must be updated each time you add a new field.

Fields stored on classes are mutable, and new fields can be added freely, which you may or may not like. It's possible to provide more access control and to create read-only fields using the `@property` decorator, but once again, this requires writing more glue code.

Writing a custom class is a great option whenever you'd like to add business logic and behavior to your record objects using methods. However, this means that these objects are technically no longer plain data objects:

Python

>>>

```
>>> class Car:
...     def __init__(self, color, mileage, automatic):
...         self.color = color
...         self.mileage = mileage
...         self.automatic = automatic
...
...>>> car1 = Car("red", 3812.4, True)
...>>> car2 = Car("blue", 40231.0, False)

>>> # Get the mileage:
>>> car2.mileage
40231.0

>>> # Classes are mutable:
>>> car2.mileage = 12
>>> car2.windshield = "broken"

>>> # String representation is not very useful
>>> # (must add a manually written __repr__ method):
>>> car1
<Car object at 0x1081e69e8>
```

dataclasses.dataclass: Python 3.7+ Data Classes

[Data classes](#) are available in Python 3.7 and above. They provide an excellent alternative to defining your own data storage classes from scratch.

By writing a data class instead of a plain Python class, your object instances get a few useful features out of the box that will save you some typing and manual implementation work:

- The syntax for defining instance variables is shorter, since you don't need to implement the `__init__()` method.
- Instances of your data class automatically get nice-looking string representation via an auto-generated `__repr__()` method.
- Instance variables accept type annotations, making your data class self-documenting to a degree. Keep in mind that type annotations are just hints that are not enforced without a separate type-checking tool.

Data classes are typically created using the `@dataclass` [decorator](#), as you'll see in the code example below:

Python

>>>

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Car:
...     color: str
...     mileage: float
...     automatic: bool
...
... car1 = Car("red", 3812.4, True)

>>> # Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

>>> # Accessing fields:
>>> car1.mileage
3812.4

>>> # Fields are mutable:
>>> car1.mileage = 12
>>> car1.windshield = "broken"

>>> # Type annotations are not enforced without
>>> # a separate type checking tool like mypy:
>>> Car("red", "NOT_A_FLOAT", 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

To learn more about Python data classes, check out the [The Ultimate Guide to Data Classes in Python 3.7](#).

`collections.namedtuple`: Convenient Data Objects

The [`namedtuple`](#) class available in Python 2.6+ provides an extension of the built-in `tuple` data type. Similar to defining a custom class, using `namedtuple` allows you to define reusable blueprints for your records that ensure the correct field names are used.

`namedtuple` objects are immutable, just like regular tuples. This means you can't add new fields or modify existing fields after the `namedtuple` instance is created.

Besides that, `namedtuple` objects are, well ... named tuples. Each object stored in them can be accessed through a unique identifier. This frees you from having to remember integer indexes or resort to workarounds like defining **integer constants** as mnemonics for your indexes.

`namedtuple` objects are implemented as regular Python classes internally. When it comes to memory usage, they're also better than regular classes and just as memory efficient as regular tuples:

Python

>>>

```
>>> from collections import namedtuple
>>> from sys import getsizeof

>>> p1 = namedtuple("Point", "x y z")(1, 2, 3)
>>> p2 = (1, 2, 3)

>>> getsizeof(p1)
64
>>> getsizeof(p2)
64
```

`namedtuple` objects can be an easy way to clean up your code and make it more readable by enforcing a better structure for your data.

I find that going from ad-hoc data types like dictionaries with a fixed format to `namedtuple` objects helps me to express the intent of my code more clearly. Often when I apply this refactoring, I magically come up with a better solution for the problem I'm facing.

Using `namedtuple` objects over regular (unstructured) tuples and dicts can also make your coworkers' lives easier by making the data that's being passed around self-documenting, at least to a degree:

```
Python >>>
>>> from collections import namedtuple
>>> Car = namedtuple("Car", "color mileage automatic")
>>> car1 = Car("red", 3812.4, True)

>>> # Instances have a nice repr:
>>> car1
Car(color="red", mileage=3812.4, automatic=True)

>>> # Accessing fields:
>>> car1.mileage
3812.4

>>> # Fields are immutable:
>>> car1.mileage = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

>>> car1.windshield = "broken"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'windshield'
```

typing.NamedTuple: Improved Namedtuples

Added in Python 3.6, `typing.NamedTuple` is the younger sibling of the `namedtuple` class in the `collections` module. It's very similar to `namedtuple`, with the main difference being an updated syntax for defining new record types and added support for [type hints](#).

Please note that type annotations are not enforced without a separate type-checking tool like [mypy](#). But even without tool support, they can provide useful hints for other programmers (or be terribly confusing if the type hints become out of date):

```
Python

>>> from typing import NamedTuple

>>> class Car(NamedTuple):
...     color: str
...     mileage: float
...     automatic: bool

>>> car1 = Car("red", 3812.4, True)

>>> # Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

>>> # Accessing fields:
>>> car1.mileage
3812.4

>>> # Fields are immutable:
>>> car1.mileage = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

>>> car1.windshield = "broken"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'windshield'

>>> # Type annotations are not enforced without
>>> # a separate type checking tool like mypy:
>>> Car("red", "NOT_A_FLOAT", 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

struct.Struct: Serialized C Structs

The `struct.Struct` class converts between Python values and C structs serialized into Python bytes objects. For example, it can be used to handle binary data stored in files or coming in from network connections.

Structs are defined using a mini language based on [format strings](#) that allows you to define the arrangement of various C data types like `char`, `int`, and `long` as well as their `unsigned` variants.

Serialized structs are seldom used to represent data objects meant to be handled purely inside Python code. They're intended primarily as a data exchange format rather than as a way of holding data in memory that's only used by Python code.

In some cases, packing primitive data into structs may use less memory than keeping it in other data types. However, in most cases that would be quite an advanced (and probably unnecessary) optimization:

```
Python>>> from struct import Struct
>>> MyStruct = Struct("i?f")
>>> data = MyStruct.pack(23, False, 42.0)

>>> # All you get is a blob of data:
>>> data
b'\x17\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00(B'

>>> # Data blobs can be unpacked again:
>>> MyStruct.unpack(data)
(23, False, 42.0)
```

`types.SimpleNamespace`. Dotted Attribute Access

Here's one more slightly obscure choice for implementing data objects in Python: `types.SimpleNamespace`. This class was added in Python 3.3 and provides **attribute access** to its namespace.

This means `SimpleNamespace` instances expose all of their keys as class attributes. You can use `obj.key` dotted attribute access instead of the `obj['key']` square-bracket indexing syntax that's used by regular dicts. All instances also include a meaningful `__repr__` by default.

As its name proclaims, `SimpleNamespace` is simple! It's basically a dictionary that allows attribute access and prints nicely. Attributes can be added, modified, and deleted freely:

```
Python >>>
>>> from types import SimpleNamespace
>>> car1 = SimpleNamespace(color="red", mileage=3812.4, automatic=True)

>>> # The default repr:
>>> car1
namespace(automatic=True, color='red', mileage=3812.4)

>>> # Instances support attribute access and are mutable:
>>> car1.mileage = 12
>>> car1.windshield = "broken"
>>> del car1.automatic
>>> car1
namespace(color='red', mileage=12, windshield='broken')
```

Records, Structs, and Data Objects in Python: Summary

As you've seen, there's quite a number of different options for implementing records or data objects. Which type should you use for data objects in Python? Generally your decision will depend on your use case:

- If you have only a few fields, then using a plain tuple object may be okay if the field order is easy to remember or field names are superfluous. For example, think of an (x, y, z) point in three-dimensional space.
- If you need immutable fields, then plain tuples, `collections.namedtuple`, and `typing.NamedTuple` are all good options.
- If you need to lock down field names to avoid typos, then `collections.namedtuple` and `typing.NamedTuple` are your friends.
- If you want to keep things simple, then a plain dictionary object might be a good choice due to the convenient syntax that closely resembles [JSON](#).
- If you need full control over your data structure, then it's time to write a custom class with `@property` setters and getters.
- If you need to add behavior (methods) to the object, then you should write a custom class, either from scratch, or using the `dataclass` decorator, or by extending `collections.namedtuple` or `typing.NamedTuple`.
- If you need to pack data tightly to serialize it to disk or to send it over the network, then it's time to read up on `struct.Struct` because this is a great use case for it!

If you're looking for a safe default choice, then my general recommendation for implementing a plain record, struct, or data object in Python would be to use `collections.namedtuple` in Python 2.x and its younger sibling, `typing.NamedTuple` in Python 3.

Sets and Multisets

In this section, you'll see how to implement mutable and immutable set and multiset (bag) data structures in Python using built-in data types and classes from the standard library.

A **set** is an unordered collection of objects that doesn't allow duplicate elements. Typically, sets are used to quickly test a value for membership in the set, to insert or delete new values from a set, and to compute the union or intersection of two sets.

In a proper set implementation, membership tests are expected to run in fast $O(1)$ time. Union, intersection, difference, and subset operations should take $O(n)$ time on average. The set implementations included in Python's standard library [follow these performance characteristics](#).

Just like dictionaries, sets get special treatment in Python and have some syntactic sugar that makes them easy to create. For example, the curly-brace set expression syntax and [set comprehensions](#) allow you to conveniently define new set instances:

Python

```
vowels = {"a", "e", "i", "o", "u"}  
squares = {x * x for x in range(10)}
```

But be careful: To create an empty set you'll need to call the `set()` constructor. Using empty curly-braces `{}` is ambiguous and will create an empty dictionary instead.

Python and its standard library provide several set implementations. Let's have a look at them.

set: Your Go-To Set

The `set` type is the built-in set implementation in Python. It's mutable and allows for the dynamic insertion and deletion of elements.

Python's sets are backed by the `dict` data type and share the same performance characteristics. Any [hashable](#) object can be stored in a set:

Python

>>>

```
>>> vowels = {"a", "e", "i", "o", "u"}  
>>> "e" in vowels  
True  
  
>>> letters = set("alice")  
>>> letters.intersection(vowels)  
{'a', 'e', 'i'}  
  
>>> vowels.add("x")  
>>> vowels  
{'i', 'a', 'u', 'o', 'x', 'e'}  
  
>>> len(vowels)  
6
```

frozenset: Immutable Sets

The `frozenset` class implements an immutable version of `set` that can't be changed after it's been constructed.

`frozenset` objects are static and allow only query operations on their elements, not inserts or deletions. Because `frozenset` objects are static and hashable, they can be used as dictionary keys or as elements of another set, something that isn't possible with regular (mutable) `set` objects:

Python

>>>

```
>>> vowels = frozenset({"a", "e", "i", "o", "u"})  
>>> vowels.add("p")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'frozenset' object has no attribute 'add'  
  
>>> # Frozensets are hashable and can  
>>> # be used as dictionary keys:
```

```
>>> d = { frozenset({1, 2, 3}): "hello" }
>>> d[frozenset({1, 2, 3})]
'hello'
```

`collections.Counter`: Multisets

The [`collections.Counter`](#) class in the Python standard library implements a multiset, or bag, type that allows elements in the set to have more than one occurrence.

This is useful if you need to keep track of not only *if* an element is part of a set, but also *how many times* it's included in the set:

```
Python >>>
>>> from collections import Counter
>>> inventory = Counter()

>>> loot = {"sword": 1, "bread": 3}
>>> inventory.update(loot)
>>> inventory
Counter({'bread': 3, 'sword': 1})

>>> more_loot = {"sword": 1, "apple": 1}
>>> inventory.update(more_loot)
>>> inventory
Counter({'bread': 3, 'sword': 2, 'apple': 1})
```

One caveat for the Counter class is that you'll want to be careful when counting the number of elements in a Counter object. Calling `len()` returns the number of *unique* elements in the multiset, whereas the *total* number of elements can be retrieved using `sum()`:

```
Python >>>
>>> len(inventory)
3 # Unique elements

>>> sum(inventory.values())
6 # Total no. of elements
```

Sets and Multisets in Python: Summary

Sets are another useful and commonly used data structure included with Python and its standard library. Here are a few guidelines for deciding which one to use:

- If you need a mutable set, then use the built-in set type.
- If you need hashable objects that can be used as dictionary or set keys, then use a `frozenset`.
- If you need a multiset, or bag, data structure, then use `collections.Counter`.

Stacks (LIFOs)

A **stack** is a collection of objects that supports fast **Last-In/First-Out** (LIFO) semantics for inserts and deletes. Unlike lists or arrays, stacks typically don't allow for random access to the objects they contain. The insert and delete operations are also often called **push** and **pop**.

A useful real-world analogy for a stack data structure is a stack of plates. New plates are added to the top of the stack, and because the plates are precious and heavy, only the topmost plate can be moved. In other words, the last plate on the stack must be the first one removed (LIFO). To reach the plates that are lower down in the stack, the topmost plates must be removed one by one.

Performance-wise, a proper [stack implementation](#) is expected to take $O(1)$ time for insert and delete operations.

Stacks have a wide range of uses in algorithms. For example, they're used in language parsing as well as runtime

memory management, which relies on a **call stack**. A short and beautiful algorithm using a stack is [depth-first search](#) (DFS) on a tree or graph data structure.

Python ships with several stack implementations that each have slightly different characteristics. Let's take a look at them and compare their characteristics.

list: Simple, Built-In Stacks

Python's built-in `list` type [makes a decent stack data structure](#) as it supports push and pop operations in [amortized O\(1\)](#) time.

Python's lists are implemented as dynamic arrays internally, which means they occasionally need to resize the storage space for elements stored in them when elements are added or removed. The list over-allocates its backing storage so that not every push or pop requires resizing. As a result, you get an amortized $O(1)$ time complexity for these operations.

The downside is that this makes their performance less consistent than the stable $O(1)$ inserts and deletes provided by a linked list-based implementation (as you'll see below with `collections.deque`). On the other hand, lists do provide fast $O(1)$ time random access to elements on the stack, and this can be an added benefit.

There's an important performance caveat that you should be aware of when using lists as stacks: To get the amortized $O(1)$ performance for inserts and deletes, new items must be added to the *end* of the list with the `append()` method and removed again from the end using `pop()`. For optimum performance, stacks based on Python lists should grow towards higher indexes and shrink towards lower ones.

Adding and removing from the front is much slower and takes $O(n)$ time, as the existing elements must be shifted around to make room for the new element. This is a performance [antipattern](#) that you should avoid as much as possible:

```
Python >>>
>>> s = []
>>> s.append("eat")
>>> s.append("sleep")
>>> s.append("code")

>>> s
['eat', 'sleep', 'code']

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

`collections.deque`: Fast and Robust Stacks

The `deque` class implements a double-ended queue that supports adding and removing elements from either end in $O(1)$ time (non-amortized). Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.

Python's deque objects are implemented as [doubly-linked lists](#), which gives them excellent and consistent performance for inserting and deleting elements but poor $O(n)$ performance for randomly accessing elements in the middle of a stack.

Overall, [`collections.deque` is a great choice](#) if you're looking for a stack data structure in Python's standard library that has the performance characteristics of a linked-list implementation:

```
Python >>>
>>> from collections import deque
>>> s = deque()
```

```

>>> s.append("eat")
>>> s.append("sleep")
>>> s.append("code")

>>> s
deque(['eat', 'sleep', 'code'])

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from an empty deque

```

queue.LifoQueue: Locking Semantics for Parallel Computing

The [LifoQueue](#) stack implementation in the Python standard library is synchronized and provides **locking semantics** to support multiple concurrent producers and consumers.

Besides LifoQueue, the queue module contains several other classes that implement multi-producer, multi-consumer queues that are useful for parallel computing.

Depending on your use case, the locking semantics might be helpful, or they might just incur unneeded overhead. In this case, you'd be better off using a list or a deque as a general-purpose stack:

Python

```

>>> from queue import LifoQueue
>>> s = LifoQueue()
>>> s.put("eat")
>>> s.put("sleep")
>>> s.put("code")

>>> s
<queue.LifoQueue object at 0x108298dd8>

>>> s.get()
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'

>>> s.get_nowait()
queue.Empty

>>> s.get() # Blocks/waits forever...

```

Stack Implementations in Python: Summary

As you've seen, Python ships with several implementations for a stack data structure. All of them have slightly different characteristics as well as performance and usage trade-offs.

If you're not looking for parallel processing support (or if you don't want to handle locking and unlocking manually), then your choice comes down to the built-in `list` type or `collections.deque`. The difference lies in the data structure used behind the scenes and overall ease of use.

`list` is backed by a dynamic array, which makes it great for fast random access but requires occasional resizing when elements are added or removed.

The list over-allocates its backing storage so that not every push or pop requires resizing, and you get an amortized $O(1)$ time complexity for these operations. But you do need to be careful to only insert and remove items using `append()` and `pop()`. Otherwise, performance slows down to $O(n)$.

`collections.deque` is backed by a doubly-linked list, which optimizes appends and deletes at both ends and provides consistent $O(1)$ performance for these operations. Not only is its performance more stable, the `deque` class is also easier to use because you don't have to worry about adding or removing items from the wrong end.

In summary, `collections.deque` is an excellent choice for implementing a stack (LIFO queue) in Python.

Queues (FIFOs)

In this section, you'll see how to implement a **First-In/First-Out** (FIFO) queue data structure using only built-in data types and classes from the Python standard library.

A **queue** is a collection of objects that supports fast FIFO semantics for inserts and deletes. The insert and delete operations are sometimes called **enqueue** and **dequeue**. Unlike lists or arrays, queues typically don't allow for random access to the objects they contain.

Here's a real-world analogy for a FIFO queue:

Imagine a line of Pythonistas waiting to pick up their conference badges on day one of PyCon registration. As new people enter the conference venue and queue up to receive their badges, they join the line (enqueue) at the back of the queue. Developers receive their badges and conference swag bags and then exit the line (dequeue) at the front of the queue.

Another way to memorize the characteristics of a queue data structure is to think of it as a pipe. You add ping-pong balls to one end, and they travel to the other end, where you remove them. While the balls are in the queue (a solid metal pipe) you can't get at them. The only way to interact with the balls in the queue is to add new ones at the back of the pipe (enqueue) or to remove them at the front (dequeue).

Queues are similar to stacks. The difference between them lies in how items are removed. With a **queue**, you remove the item *least* recently added (FIFO) but with a **stack**, you remove the item *most* recently added (LIFO).

Performance-wise, a proper queue implementation is expected to take $O(1)$ time for insert and delete operations. These are the two main operations performed on a queue, and in a correct implementation, they should be fast.

Queues have a wide range of applications in algorithms and often help solve scheduling and parallel programming problems. A short and beautiful algorithm using a queue is [breadth-first search](#) (BFS) on a tree or graph data structure.

Scheduling algorithms often use **priority queues** internally. These are specialized queues. Instead of retrieving the next element by insertion time, a [priority queue](#) retrieves the *highest-priority* element. The priority of individual elements is decided by the queue based on the ordering applied to their keys.

A regular queue, however, won't reorder the items it carries. Just like in the pipe example, you get out what you put in, and in exactly that order.

Python ships with several queue implementations that each have slightly different characteristics. Let's review them.

list: Terribly Sloooow Queues

It's possible to [use a regular list as a queue](#), but this is not ideal from a performance perspective. Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all the other elements by one, requiring $O(n)$ time.

Therefore, I would *not* recommend using a list as a makeshift queue in Python unless you're dealing with only a small number of elements:

```
Python >>>
>>> q = []
>>> q.append("eat")
>>> q.append("sleep")
>>> q.append("code")

>>> q
['eat', 'sleep', 'code']

>>> # Careful: This is slow!
>>> q.pop(0)
'eat'
```

collections.deque: Fast and Robust Queues

The deque class implements a double-ended queue that supports adding and removing elements from either end in $O(1)$ time (non-amortized). Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.

Python's deque objects are implemented as doubly-linked lists. This gives them excellent and consistent performance for inserting and deleting elements, but poor $O(n)$ performance for randomly accessing elements in the middle of the stack.

As a result, `collections.deque` is a great default choice if you're looking for a queue data structure in Python's standard library:

```
Python >>>
>>> from collections import deque
>>> q = deque()
>>> q.append("eat")
>>> q.append("sleep")
>>> q.append("code")

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.popleft()
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'

>>> q.popleft()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from an empty deque
```

queue.Queue: Locking Semantics for Parallel Computing

The `queue.Queue` implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

The `queue` module contains several other classes implementing multi-producer/multi-consumer queues that are

The queue module contains several other classes implementing multi-producer, multi-consumer queues that are useful for parallel computing.

Depending on your use case, the locking semantics might be helpful or just incur unneeded overhead. In this case, you'd be better off using collections.deque as a general-purpose queue:

Python

>>>

```
>>> from queue import Queue
>>> q = Queue()
>>> q.put("eat")
>>> q.put("sleep")
>>> q.put("code")

>>> q
<queue.Queue object at 0x1070f5b38>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get_nowait()
queue.Empty

>>> q.get() # Blocks/waits forever...
```

`multiprocessing.Queue`: Shared Job Queues

`multiprocessing.Queue` is a shared job queue implementation that allows queued items to be processed in parallel by multiple concurrent workers. Process-based parallelization is popular in CPython due to the [global interpreter lock](#) (GIL) that prevents some forms of parallel execution on a single interpreter process.

As a specialized queue implementation meant for sharing data between processes, `multiprocessing.Queue` makes it easy to distribute work across multiple processes in order to work around the GIL limitations. This type of queue can store and transfer any [pickleable](#) object across process boundaries:

Python

>>>

```
>>> from multiprocessing import Queue
>>> q = Queue()
>>> q.put("eat")
>>> q.put("sleep")
>>> q.put("code")

>>> q
<multiprocessing.queues.Queue object at 0x1081c12b0>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get() # Blocks/waits forever...
```

Queues in Python: Summary

Python includes several queue implementations as part of the core language and its standard library.

`list` objects can be used as queues, but this is generally not recommended due to slow performance.

If you're not looking for parallel processing support, then the implementation offered by `collections.deque` is an excellent default choice for implementing a FIFO queue data structure in Python. It provides the performance

characteristics you'd expect from a good queue implementation and can also be used as a stack (LIFO queue).

Priority Queues

A **priority queue** is a container data structure that manages a set of records with totally-ordered keys to provide quick access to the record with the smallest or largest key in the set.

You can think of a priority queue as a modified queue. Instead of retrieving the next element by insertion time, it retrieves the *highest-priority* element. The priority of individual elements is decided by the order applied to their keys.

Priority queues are commonly used for dealing with scheduling problems. For example, you might use them to give precedence to tasks with higher urgency.

Think about the job of an operating system task scheduler:

Ideally, higher-priority tasks on the system (such as playing a real-time game) should take precedence over lower-priority tasks (such as downloading updates in the background). By organizing pending tasks in a priority queue that uses task urgency as the key, the task scheduler can quickly select the highest-priority tasks and allow them to run first.

In this section, you'll see a few options for how you can implement priority queues in Python using built-in data structures or data structures included in Python's standard library. Each implementation will have its own upsides and downsides, but in my mind there's a clear winner for most common scenarios. Let's find out which one it is.

list: Manually Sorted Queues

You can use a sorted `list` to quickly identify and delete the smallest or largest element. The downside is that inserting new elements into a list is a slow $O(n)$ operation.

While the insertion point can be found in $O(\log n)$ time using `bisect.insort` in the standard library, this is always dominated by the slow insertion step.

Maintaining the order by appending to the list and re-sorting also takes at least $O(n \log n)$ time. Another downside is that you must manually take care of re-sorting the list when new elements are inserted. It's easy to introduce bugs by missing this step, and the burden is always on you, the developer.

This means sorted lists are only suitable as priority queues when there will be few insertions:

```
Python >>>
>>> q = []
>>> q.append((2, "code"))
>>> q.append((1, "eat"))
>>> q.append((3, "sleep"))
>>> # Remember to re-sort every time a new element is inserted,
>>> # or use bisect.insort()
>>> q.sort(reverse=True)

>>> while q:
...     next_item = q.pop()
...     print(next_item)
...
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

heapq: List-Based Binary Heaps

`heapq` is a binary heap implementation usually backed by a plain `list`, and it supports insertion and extraction of the smallest element in $O(\log n)$ time.

This module is a good choice for implementing priority queues in Python. Since `heapq` technically provides only a min-heap implementation, extra steps must be taken to ensure sort stability and other features typically expected from a practical priority queue:

Python

>>>

```
>>> import heapq
>>> q = []
>>> heapq.heappush(q, (2, "code"))
>>> heapq.heappush(q, (1, "eat"))
>>> heapq.heappush(q, (3, "sleep"))

>>> while q:
...     next_item = heapq.heappop(q)
...     print(next_item)
...
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

queue.PriorityQueue: Beautiful Priority Queues

`queue.PriorityQueue` uses `heapq` internally and shares the same time and space complexities. The difference is that `PriorityQueue` is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

Depending on your use case, this might be helpful, or it might just slow your program down slightly. In any case, you might prefer the class-based interface provided by `PriorityQueue` over the function-based interface provided by `heapq`:

Python

>>>

```
>>> from queue import PriorityQueue
>>> q = PriorityQueue()
>>> q.put((2, "code"))
>>> q.put((1, "eat"))
>>> q.put((3, "sleep"))

>>> while not q.empty():
...     next_item = q.get()
...     print(next_item)
...
(1, 'eat')
(2, 'code')
(3, 'sleep')
```

Priority Queues in Python: Summary

Python includes several priority queue implementations ready for you to use.

`queue.PriorityQueue` stands out from the pack with a nice object-oriented interface and a name that clearly states its intent. It should be your preferred choice.

If you'd like to avoid the locking overhead of `queue.PriorityQueue`, then using the `heapq` module directly is also a good option.

Conclusion: Python Data Structures

That concludes your tour of common data structures in Python. With the knowledge you've gained here, you're ready to implement efficient data structures that are just right for your specific algorithm or use case.

In this tutorial, you've learned:

- Which common **abstract data types** are built into the Python standard library
- How the most common abstract data types map to Python's **naming scheme**
- How to put abstract data types to **practical use** in various algorithms

If you enjoyed what you learned in this sample from [Python Tricks: The Book](#), then be sure to check out [the rest of the](#)

[book.](#)

If you're interested in brushing up on your general data structures knowledge, then I highly recommend [Steven S. Skiena's *The Algorithm Design Manual*](#). It strikes a great balance between teaching you fundamental (and more advanced) data structures and showing you how to implement them in your code. Steve's book was a great help in the writing of this tutorial.

About Dan Bader

Dan Bader is the owner and editor in chief of Real Python and the main developer of the realpython.com learning platform. Dan has been writing code for more than 20 years and holds master's degree in computer science.

[» More about Dan](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[David](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)



Real Python

Sorting Algorithms in Python

by Santiago Valdarrama ⏰ Apr 15, 2020 💬 10 Comments 📁 intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [The Importance of Sorting Algorithms in Python](#)
- [Python's Built-In Sorting Algorithm](#)
- [The Significance of Time Complexity](#)
 - [Timing Your Code](#)
 - [Measuring Efficiency With Big O Notation](#)
- [The Bubble Sort Algorithm in Python](#)
 - [Implementing Bubble Sort in Python](#)
 - [Measuring Bubble Sort's Big O Runtime Complexity](#)
 - [Timing Your Bubble Sort Implementation](#)
 - [Analyzing the Strengths and Weaknesses of Bubble Sort](#)
- [The Insertion Sort Algorithm in Python](#)
 - [Implementing Insertion Sort in Python](#)
 - [Measuring Insertion Sort's Big O Runtime Complexity](#)
 - [Timing Your Insertion Sort Implementation](#)
 - [Analyzing the Strengths and Weaknesses of Insertion Sort](#)
- [The Merge Sort Algorithm in Python](#)
 - [Implementing Merge Sort in Python](#)
 - [Measuring Merge Sort's Big O Complexity](#)
 - [Timing Your Merge Sort Implementation](#)
 - [Analyzing the Strengths and Weaknesses of Merge Sort](#)
- [The Quicksort Algorithm in Python](#)
 - [Implementing Quicksort in Python](#)
 - [Selecting the pivot Element](#)
 - [Measuring Quicksort's Big O Complexity](#)
 - [Timing Your Quicksort Implementation](#)
 - [Analyzing the Strengths and Weaknesses of Quicksort](#)
- [The Timsort Algorithm in Python](#)
 - [Implementing Timsort in Python](#)
 - [Measuring Timsort's Big O Complexity](#)
 - [Timing Your Timsort Implementation](#)
 - [Analyzing the Strengths and Weaknesses of Timsort](#)

- [Conclusion](#)


blackfire.io
 Profile & Optimize Python Apps Performance
 

Now available as
Public Beta
Sign-up for free and
install in minutes!

Sorting is a basic building block that many other algorithms are built upon. It's related to several exciting ideas that you'll see throughout your programming career. Understanding how sorting algorithms in Python work behind the scenes is a fundamental step toward implementing correct and efficient algorithms that solve real-world problems.

In this tutorial, you'll learn:

- How different **sorting algorithms in Python** work and how they compare under different circumstances
- How **Python's built-in sort functionality** works behind the scenes
- How different computer science concepts like **recursion** and **divide and conquer** apply to sorting
- How to measure the efficiency of an algorithm using **Big O notation** and **Python's timeit module**

By the end of this tutorial, you'll understand sorting algorithms from both a theoretical and a practical standpoint. More importantly, you'll have a deeper understanding of different algorithm design techniques that you can apply to other areas of your work. Let's get started!

Free Download: Get a sample chapter from **Python Tricks: The Book** that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

The Importance of Sorting Algorithms in Python

Sorting is one of the most thoroughly studied algorithms in computer science. There are dozens of different sorting implementations and applications that you can use to make your code more efficient and effective.

You can use sorting to solve a wide range of problems:

- **Searching:** Searching for an item on a list works much faster if the list is sorted.
- **Selection:** Selecting items from a list based on their relationship to the rest of the items is easier with sorted data. For example, finding the k^{th} -largest or smallest value, or finding the median value of the list, is much easier when the values are in ascending or descending order.
- **Duplicates:** Finding duplicate values on a list can be done very quickly when the list is sorted.
- **Distribution:** Analyzing the frequency distribution of items on a list is very fast if the list is sorted. For example, finding the element that appears most or least often is relatively straightforward with a sorted list.

From commercial applications to academic research and everywhere in between, there are countless ways you can use sorting to save yourself time and effort.

Python's Built-In Sorting Algorithm

The Python language, like many other high-level programming languages, offers the ability to sort data out of the box using `sorted()`. Here's an example of sorting an integer array:

Python

>>>

```
>>> array = [8, 2, 6, 4, 5]
>>> sorted(array)
[2, 4, 5, 6, 8]
```

You can use `sorted()` to sort any list as long as the values inside are comparable.

Note: For a deeper dive into how Python's built-in sorting functionality works, check out [How to Use `sorted\(\)` and `sort\(\)` in Python](#) and [Sorting Data With Python](#).

The Significance of Time Complexity

This tutorial covers two different ways to measure the **runtime** of sorting algorithms:

1. For a practical point of view, you'll measure the runtime of the implementations using the `timeit` module.
2. For a more theoretical perspective, you'll measure the **runtime complexity** of the algorithms using [Big O notation](#).

Timing Your Code

When comparing two sorting algorithms in Python, it's always informative to look at how long each one takes to run. The specific time each algorithm takes will be partly determined by your hardware, but you can still use the proportional time between executions to help you decide which implementation is more time efficient.

In this section, you'll focus on a practical way to measure the actual time it takes to run to your sorting algorithms using the `timeit` module. For more information on the different ways you can time the execution of code in Python, check out [Python Timer Functions: Three Ways to Monitor Your Code](#).

Here's a function you can use to time your algorithms:

Python

```
1 from random import randint
2 from timeit import repeat
3
4 def run_sorting_algorithm(algorithm, array):
5     # Set up the context and prepare the call to the specified
6     # algorithm using the supplied array. Only import the
7     # algorithm function if it's not the built-in `sorted()`.
8     setup_code = f"from __main__ import {algorithm}" \
9         if algorithm != "sorted" else ""
10
11     stmt = f"{algorithm}({array})"
12
13     # Execute the code ten different times and return the time
14     # in seconds that each execution took
15     times = repeat(setup=setup_code, stmt=stmt, repeat=3, number=10)
16
17     # Finally, display the name of the algorithm and the
18     # minimum time it took to run
19     print(f"Algorithm: {algorithm}. Minimum execution time: {min(times)}")
```

In this example, `run_sorting_algorithm()` receives the name of the algorithm and the input array that needs to be sorted. Here's a line-by-line explanation of how it works:

- **Line 8** imports the name of the algorithm using the magic of [Python's f-strings](#). This is so that `timeit.repeat()` knows where to call the algorithm from. Note that this is only necessary for the custom implementations used in this tutorial. If the algorithm specified is the built-in `sorted()`, then nothing will be imported.
- **Line 11** prepares the call to the algorithm with the supplied array. This is the statement that will be executed and timed.
- **Line 15** calls `timeit.repeat()` with the setup code and the statement. This will call the specified sorting algorithm ten times, returning the number of seconds each one of these executions took.
- **Line 19** identifies the shortest time returned and prints it along with the name of the algorithm.

Note: A common misconception is that you should find the average time of each run of the algorithm instead of selecting the single shortest time. Time measurements are noisy because the system runs other processes.

or selecting the single shortest time. Time measurements are [noisy](#), because the system runs other processes concurrently. The shortest time is always the least noisy, making it the best representation of the algorithm's true runtime.

Here's an example of how to use `run_sorting_algorithm()` to determine the time it takes to sort an array of ten thousand integer values using `sorted()`:

Python

```
21 | ARRAY_LENGTH = 10000
22 |
23 | if __name__ == "__main__":
24 |     # Generate an array of `ARRAY_LENGTH` items consisting
25 |     # of random integer values between 0 and 999
26 |     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
27 |
28 |     # Call the function using the name of the sorting algorithm
29 |     # and the array you just created
30 |     run_sorting_algorithm(algorithm="sorted", array=array)
```

If you save the above code in a `sorting.py` file, then you can run it from the terminal and see its output:

Shell

```
$ python sorting.py
Algorithm: sorted. Minimum execution time: 0.010945824000000007
```

Remember that the time in seconds of every experiment depends in part on the hardware you use, so you'll likely see slightly different results when running the code.

Note: You can learn more about the `timeit` module in the [official Python documentation](#).

Measuring Efficiency With Big O Notation

The specific time an algorithm takes to run isn't enough information to get the full picture of its [time complexity](#). To solve this problem, you can use Big O (pronounced "big oh") notation. Big O is often used to compare different implementations and decide which one is the most efficient, skipping unnecessary details and focusing on what's most important in the runtime of an algorithm.

The time in seconds required to run different algorithms can be influenced by several unrelated factors, including processor speed or available memory. Big O, on the other hand, provides a platform to express runtime complexity in hardware-agnostic terms. With Big O, you express complexity in terms of how quickly your algorithm's runtime grows relative to the size of the input, especially as the input grows arbitrarily large.

Assuming that n is the size of the input to an algorithm, the Big O notation represents the relationship between n and the number of steps the algorithm takes to find a solution. Big O uses a capital letter "O" followed by this relationship inside parentheses. For example, $O(n)$ represents algorithms that execute a number of steps proportional to the size of their input.

Although this tutorial isn't going to dive very deep into the details of Big O notation, here are five examples of the runtime complexity of different algorithms:

Big O	Complexity	Description
$O(1)$	constant	The runtime is constant regardless of the size of the input. Finding an element in a hash table is an example of an operation that can be performed in constant time .
$O(n)$	linear	The runtime grows linearly with the size of the input. A function that checks a condition on every item of a list is an example of an $O(n)$ algorithm.
$O(n^2)$	quadratic	The runtime is a quadratic function of the size of the input. A naive implementation of finding duplicate values in a list, in which each item has to be checked twice, is an example of a quadratic algorithm.

Big O	exponential Complexity	The runtime grows exponentially with the size of the input. These algorithms are considered extremely inefficient. An example of an exponential algorithm is the three-coloring problem .
$O(\log n)$	logarithmic	The runtime grows linearly while the size of the input grows exponentially. For example, it takes one second to process one thousand elements, then it will take two seconds to process ten thousand, three seconds to process one hundred thousand, and so on. Binary search is an example of a logarithmic runtime algorithm.

This tutorial covers the Big O runtime complexity of each of the sorting algorithms discussed. It also includes a brief explanation of how to determine the runtime on each particular case. This will give you a better understanding of how to start using Big O to classify other algorithms.

Note: For a deeper understanding of Big O, together with several practical examples in Python, check out [Big O Notation and Algorithm Analysis with Python Examples](#).

The Bubble Sort Algorithm in Python

Bubble Sort is one of the most straightforward sorting algorithms. Its name comes from the way the algorithm works: With every new pass, the largest element in the list “bubbles up” toward its correct position.

Bubble sort consists of making multiple passes through a list, comparing elements one by one, and swapping adjacent items that are out of order.

Implementing Bubble Sort in Python

Here's an implementation of a bubble sort algorithm in Python:

Python

```

1 def bubble_sort(array):
2     n = len(array)
3
4     for i in range(n):
5         # Create a flag that will allow the function to
6         # terminate early if there's nothing left to sort
7         already_sorted = True
8
9         # Start looking at each item of the list one by one,
10        # comparing it with its adjacent value. With each
11        # iteration, the portion of the array that you look at
12        # shrinks because the remaining items have already been
13        # sorted.
14        for j in range(n - i - 1):
15            if array[j] > array[j + 1]:
16                # If the item you're looking at is greater than its
17                # adjacent value, then swap them
18                array[j], array[j + 1] = array[j + 1], array[j]
19

```

```
20      # Since you had to swap two elements,
21      # set the `already_sorted` flag to `False` so the
22      # algorithm doesn't finish prematurely
23      already_sorted = False
24
25      # If there were no swaps during the last iteration,
26      # the array is already sorted, and you can terminate
27      if already_sorted:
28          break
29
30  return array
```

Since this implementation sorts the array in ascending order, each step “bubbles” the largest element to the end of the array. This means that each iteration takes fewer steps than the previous iteration because a continuously larger portion of the array is sorted.

The loops in **lines 4 and 10** determine the way the algorithm runs through the list. Notice how *j* initially goes from the first element in the list to the element immediately before the last. During the second iteration, *j* runs until two items from the last, then three items from the last, and so on. At the end of each iteration, the end portion of the list will be sorted.

As the loops progress, **line 15** compares each element with its adjacent value, and **line 18** swaps them if they are in the incorrect order. This ensures a sorted list at the end of the function.

Note: The `already_sorted` flag in **lines 13, 23, and 27** of the code above is an optimization to the algorithm, and it's not required in a fully functional bubble sort implementation. However, it allows the function to save unnecessary steps if the list ends up wholly sorted before the loops have finished.

As an exercise, you can remove the use of this flag and compare the runtimes of both implementations.

To properly analyze how the algorithm works, consider a list with values [8, 2, 6, 4, 5]. Assume you're using `bubble_sort()` from above. Here's a figure illustrating what the array looks like at each iteration of the algorithm:

The Bubble Sort Process

Now take a step-by-step look at what's happening with the array as the algorithm progresses:

1. The code starts by comparing the first element, 8, with its adjacent element, 2. Since $8 > 2$, the values are swapped, resulting in the following order: [2, 8, 6, 4, 5].
2. The algorithm then compares the second element, 8, with its adjacent element, 6. Since $8 > 6$, the values are swapped, resulting in the following order: [2, 6, 8, 4, 5].
3. Next, the algorithm compares the third element, 8, with its adjacent element, 4. Since $8 > 4$, it swaps the values as well, resulting in the following order: [2, 6, 4, 8, 5].
4. Finally, the algorithm compares the fourth element, 8, with its adjacent element, 5, and swaps them as well, resulting in [2, 6, 4, 5, 8]. At this point, the algorithm completed the first pass through the list ($i = 0$). Notice how the value 8 bubbled up from its initial location to its correct position at the end of the list.
5. The second pass ($i = 1$) takes into account that the last element of the list is already positioned and focuses on the remaining four elements, [2, 6, 4, 5]. At the end of this pass, the value 6 finds its correct position. The third pass through the list positions the value 5, and so on until the list is sorted.

Measuring Bubble Sort's Big O Runtime Complexity

Your implementation of bubble sort consists of two nested [for loops](#) in which the algorithm performs $n - 1$ comparisons, then $n - 2$ comparisons, and so on until the final comparison is done. This comes at a total of $(n - 1) + (n - 2) + (n - 3) + \dots + 2 + 1 = n(n - 1)/2$ comparisons, which can also be written as $\frac{1}{2}n^2 - \frac{1}{2}n$.

You learned earlier that Big O focuses on how the runtime grows in comparison to the size of the input. That means that, in order to turn the above equation into the Big O complexity of the algorithm, you need to remove the constants because they don't change with the input size.

Doing so simplifies the notation to $n^2 - n$. Since n^2 grows much faster than n , this last term can be dropped as well, leaving bubble sort with an average- and worst-case complexity of $O(n^2)$.

In cases where the algorithm receives an array that's already sorted—and assuming the implementation includes the `already_sorted` flag optimization explained before—the runtime complexity will come down to a much better $O(n)$ because the algorithm will not need to visit any element more than once.

$O(n)$, then, is the best-case runtime complexity of bubble sort. But keep in mind that best cases are an exception, and you should focus on the average case when comparing different algorithms.

Timing Your Bubble Sort Implementation

Using your `run_sorting_algorithm()` from earlier in this tutorial, here's the time it takes for bubble sort to process an array with ten thousand items. **Line 8** replaces the name of the algorithm and everything else stays the same:

Python

```
1 if __name__ == "__main__":
2     # Generate an array of `ARRAY_LENGTH` items consisting
3     # of random integer values between 0 and 999
4     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
5
6     # Call the function using the name of the sorting algorithm
7     # and the array you just created
8     run_sorting_algorithm(algorithm="bubble_sort", array=array)
```

You can now run the script to get the execution time of bubble_sort:

Shell

```
$ python sorting.py
Algorithm: bubble_sort. Minimum execution time: 73.21720498399998
```

It took 73 seconds to sort the array with ten thousand elements. This represents the fastest execution out of the ten repetitions that `run_sorting_algorithm()` runs. Executing this script multiple times will produce similar results.

Note: A single execution of bubble sort took 73 seconds, but the algorithm ran ten times using `timeit.repeat()`. This means that you should expect your code to take around $73 * 10 = 730$ seconds to run, assuming you have similar hardware characteristics. Slower machines may take much longer to finish.

Analyzing the Strengths and Weaknesses of Bubble Sort

The main advantage of the bubble sort algorithm is its **simplicity**. It is straightforward to both implement and understand. This is probably the main reason why most computer science courses introduce the topic of sorting using bubble sort.

As you saw before, the disadvantage of bubble sort is that it is **slow**, with a runtime complexity of $O(n^2)$. Unfortunately, this rules it out as a practical candidate for sorting large arrays.

The Insertion Sort Algorithm in Python

Like bubble sort, the **insertion sort** algorithm is straightforward to implement and understand. But unlike bubble sort, it builds the sorted list one element at a time by comparing each item with the rest of the list and inserting it into its correct position. This “insertion” procedure gives the algorithm its name.

An excellent analogy to explain insertion sort is the way you would sort a deck of cards. Imagine that you’re holding a group of cards in your hands, and you want to arrange them in order. You’d start by comparing a single card step by step with the rest of the cards until you find its correct position. At that point, you’d insert the card in the correct location and start over with a new card, repeating until all the cards in your hand were sorted.

Implementing Insertion Sort in Python

The insertion sort algorithm works exactly like the example with the deck of cards. Here’s the implementation in Python:

Python

```
1 def insertion_sort(array):
2     # Loop from the second element of the array until
3     # the last element
4     for i in range(1, len(array)):
5         # This is the element we want to position in its
6         # correct place
7         key_item = array[i]
8
9         # Initialize the variable that will be used to
10        # find the correct position of the element referenced
11        # by `key_item`
12        j = i - 1
13
14        # Run through the list of items (the left
15        # portion of the array) and find the correct position
```

```

16     # of the element referenced by `key_item`. Do this only
17     # if `key_item` is smaller than its adjacent values.
18     while j >= 0 and array[j] > key_item:
19         # Shift the value one position to the left
20         # and reposition j to point to the next element
21         # (from right to left)
22         array[j + 1] = array[j]
23         j -= 1
24
25     # When you finish shifting the elements, you can position
26     # `key_item` in its correct location
27     array[j + 1] = key_item
28
29 return array

```

Unlike bubble sort, this implementation of insertion sort constructs the sorted list by pushing smaller items to the left. Let's break down `insertion_sort()` line by line:

- **Line 4** sets up the loop that determines the `key_item` that the function will position during each iteration. Notice that the loop starts with the second item on the list and goes all the way to the last item.
- **Line 7** initializes `key_item` with the item that the function is trying to place.
- **Line 12** initializes a variable that will consecutively point to each element to the left of `key_item`. These are the elements that will be consecutively compared with `key_item`.
- **Line 18** compares `key_item` with each value to its left using a `while` loop, shifting the elements to make room to place `key_item`.
- **Line 27** positions `key_item` in its correct place after the algorithm shifts all the larger values to the right.

Here's a figure illustrating the different iterations of the algorithm when sorting the array [8, 2, 6, 4, 5]:

The Insertion Sort Process

Now here's a summary of the steps of the algorithm when sorting the array:

1. The algorithm starts with `key_item = 2` and goes through the subarray to its left to find the correct position

© 2023, Open Data Science, Inc.

for it. In this case, the subarray is [8].

2. Since $2 < 8$, the algorithm shifts element 8 one position to its right. The resultant array at this point is [8, 8, 6, 4, 5].
3. Since there are no more elements in the subarray, the `key_item` is now placed in its new position, and the final array is [2, 8, 6, 4, 5].
4. The second pass starts with `key_item = 6` and goes through the subarray located to its left, in this case [2, 8].
5. Since $6 < 8$, the algorithm shifts 8 to its right. The resultant array at this point is [2, 8, 8, 4, 5].
6. Since $6 > 2$, the algorithm doesn't need to keep going through the subarray, so it positions `key_item` and finishes the second pass. At this time, the resultant array is [2, 6, 8, 4, 5].
7. The third pass through the list puts the element 4 in its correct position, and the fourth pass places element 5 in the correct spot, leaving the array sorted.

Measuring Insertion Sort's Big O Runtime Complexity

Similar to your bubble sort implementation, the insertion sort algorithm has a couple of nested loops that go over the list. The inner loop is pretty efficient because it only goes through the list until it finds the correct position of an element. That said, the algorithm still has an $O(n^2)$ runtime complexity on the average case.

The worst case happens when the supplied array is sorted in reverse order. In this case, the inner loop has to execute every comparison to put every element in its correct position. This still gives you an $O(n^2)$ runtime complexity.

The best case happens when the supplied array is already sorted. Here, the inner loop is never executed, resulting in an $O(n)$ runtime complexity, just like the best case of bubble sort.

Although bubble sort and insertion sort have the same Big O runtime complexity, in practice, insertion sort is considerably more efficient than bubble sort. If you look at the implementation of both algorithms, then you can see how insertion sort has to make fewer comparisons to sort the list.

Timing Your Insertion Sort Implementation

To prove the assertion that insertion sort is more efficient than bubble sort, you can time the insertion sort algorithm and compare it with the results of bubble sort. To do this, you just need to replace the call to `run_sorting_algorithm()` with the name of your insertion sort implementation:

Python

```
1 if __name__ == "__main__":
2     # Generate an array of `ARRAY_LENGTH` items consisting
3     # of random integer values between 0 and 999
4     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
5
6     # Call the function using the name of the sorting algorithm
7     # and the array we just created
8     run_sorting_algorithm(algorithm="insertion_sort", array=array)
```

You can execute the script as before:

Shell

```
$ python sorting.py
Algorithm: insertion_sort. Minimum execution time: 56.71029764299999
```

Notice how the insertion sort implementation took around 17 fewer seconds than the bubble sort implementation to sort the same array. Even though they're both $O(n^2)$ algorithms, insertion sort is more efficient.

Analyzing the Strengths and Weaknesses of Insertion Sort

Just like bubble sort, the insertion sort algorithm is very uncomplicated to implement. Even though insertion sort is

Just like bubble sort, the insertion sort algorithm is very uncomplicated to implement. Even though insertion sort is an $O(n^2)$ algorithm, it's also much more efficient in practice than other quadratic implementations such as bubble sort.

There are more powerful algorithms, including merge sort and quicksort, but these implementations are recursive and usually fail to beat insertion sort when working on small lists. Some quicksort implementations even use insertion sort internally if the list is small enough to provide a faster overall implementation. [Timsort](#) also uses insertion sort internally to sort small portions of the input array.

That said, insertion sort is not practical for large arrays, opening the door to algorithms that can scale in more efficient ways.

The Merge Sort Algorithm in Python

Merge sort is a very efficient sorting algorithm. It's based on the [divide-and-conquer](#) approach, a powerful algorithmic technique used to solve complex problems.

To properly understand divide and conquer, you should first understand the concept of **recursion**. Recursion involves breaking a problem down into smaller subproblems until they're small enough to manage. In programming, recursion is usually expressed by a function calling itself.

Note: This tutorial doesn't explore recursion in depth. To better understand how recursion works and see it in action using Python, check out [Thinking Recursively in Python](#).

Divide-and-conquer algorithms typically follow the same structure:

1. The original input is broken into several parts, each one representing a subproblem that's similar to the original but simpler.
2. Each subproblem is solved recursively.
3. The solutions to all the subproblems are combined into a single overall solution.

In the case of merge sort, the divide-and-conquer approach divides the set of input values into two equal-sized parts, sorts each half recursively, and finally merges these two sorted parts into a single sorted list.

Implementing Merge Sort in Python

The implementation of the merge sort algorithm needs two different pieces:

1. A function that recursively splits the input in half
2. A function that merges both halves, producing a sorted array

Here's the code to merge two different arrays:

Python

```
1 def merge(left, right):
2     # If the first array is empty, then nothing needs
3     # to be merged, and you can return the second array as the result
4     if len(left) == 0:
5         return right
6
7     # If the second array is empty, then nothing needs
8     # to be merged, and you can return the first array as the result
9     if len(right) == 0:
10        return left
11
12    result = []
13    index_left = index_right = 0
14
15    # Now go through both arrays until all the elements
16    # make it into the resultant array
17    while len(result) < len(left) + len(right):
18        # The elements need to be sorted to add them to the
19        # resultant array, so you need to decide whether to get
20        # the next element from the first or the second array
21        if left[index_left] <= right[index_right]:
22            result.append(left[index_left])
```

```

23     index_left += 1
24 else:
25     result.append(right[index_right])
26     index_right += 1
27
28     # If you reach the end of either array, then you can
29     # add the remaining elements from the other array to
30     # the result and break the loop
31     if index_right == len(right):
32         result += left[index_left:]
33         break
34
35     if index_left == len(left):
36         result += right[index_right:]
37         break
38
39 return result

```

`merge()` receives two different sorted arrays that need to be merged together. The process to accomplish this is straightforward:

- **Lines 4 and 9** check whether either of the arrays is empty. If one of them is, then there's nothing to merge, so the function returns the other array.
- **Line 17** starts a `while loop` that ends whenever the result contains all the elements from both of the supplied arrays. The goal is to look into both arrays and combine their items to produce a sorted list.
- **Line 21** compares the elements at the head of both arrays, selects the smaller value, and appends it to the end of the resultant array.
- **Lines 31 and 35** append any remaining items to the result if all the elements from either of the arrays were already used.

With the above function in place, the only missing piece is a function that recursively splits the input array in half and uses `merge()` to produce the final result:

Python

```

41 def merge_sort(array):
42     # If the input array contains fewer than two elements,
43     # then return it as the result of the function
44     if len(array) < 2:
45         return array
46
47     midpoint = len(array) // 2
48
49     # Sort the array by recursively splitting the input
50     # into two equal halves, sorting each half and merging them
51     # together into the final result
52     return merge(
53         left=merge_sort(array[:midpoint]),
54         right=merge_sort(array[midpoint:]))

```

Here's a quick summary of the code:

- **Line 44** acts as the stopping condition for the recursion. If the input array contains fewer than two elements, then the function returns the array. Notice that this condition could be triggered by receiving either a single item or an empty array. In both cases, there's nothing left to sort, so the function should return.
- **Line 47** computes the middle point of the array.
- **Line 52** calls `merge()`, passing both sorted halves as the arrays.

Notice how this function calls itself **recursively**, halving the array each time. Each iteration deals with an ever-shrinking array until fewer than two elements remain, meaning there's nothing left to sort. At this point, `merge()` takes over, merging the two halves and producing a sorted list.

Take a look at a representation of the steps that merge sort will take to sort the array [8, 2, 6, 4, 5]:

The Merge Sort Process

The figure uses yellow arrows to represent halving the array at each recursion level. The green arrows represent merging each subarray back together. The steps can be summarized as follows:

1. The first call to `merge_sort()` with [8, 2, 6, 4, 5] defines midpoint as 2. The midpoint is used to halve the input array into `array[:2]` and `array[2:]`, producing [8, 2] and [6, 4, 5], respectively. `merge_sort()` is then recursively called for each half to sort them separately.
2. The call to `merge_sort()` with [8, 2] produces [8] and [2]. The process repeats for each of these halves.
3. The call to `merge_sort()` with [8] returns [8] since that's the only element. The same happens with the call to `merge_sort()` with [2].
4. At this point, the function starts merging the subarrays back together using `merge()`, starting with [8] and [2] as input arrays, producing [2, 8] as the result.
5. On the other side, [6, 4, 5] is recursively broken down and merged using the same procedure, producing [4, 5, 6] as the result.

6. In the final step, [2, 8] and [4, 5, 6] are merged back together with `merge()`, producing the final result: [2, 4, 5, 6, 8].

Measuring Merge Sort's Big O Complexity

To analyze the complexity of merge sort, you can look at its two steps separately:

1. `merge()` has a linear runtime. It receives two arrays whose combined length is at most n (the length of the original input array), and it combines both arrays by looking at each element at most once. This leads to a runtime complexity of $O(n)$.

2. The second step splits the input array recursively and calls `merge()` for each half. Since the array is halved until a single element remains, the total number of halving operations performed by this function is $\log_2 n$. Since `merge()` is called for each half, we get a total runtime of $O(n \log_2 n)$.

Interestingly, $O(n \log_2 n)$ is the best possible worst-case runtime that can be achieved by a sorting algorithm.

Timing Your Merge Sort Implementation

To compare the speed of merge sort with the previous two implementations, you can use the same mechanism as before and replace the name of the algorithm in [line 8](#):

Python

```
1 if __name__ == "__main__":
2     # Generate an array of `ARRAY_LENGTH` items consisting
3     # of random integer values between 0 and 999
4     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
5
6     # Call the function using the name of the sorting algorithm
7     # and the array you just created
8     run_sorting_algorithm(algorithm="merge_sort", array=array)
```

You can execute the script to get the execution time of `merge_sort`:

Shell

```
$ python sorting.py
Algorithm: merge_sort. Minimum execution time: 0.6195857160000173
```

Compared to bubble sort and insertion sort, the merge sort implementation is extremely fast, sorting the ten-thousand-element array in less than a second!

Analyzing the Strengths and Weaknesses of Merge Sort

Thanks to its runtime complexity of $O(n \log_2 n)$, merge sort is a very efficient algorithm that scales well as the size of the input array grows. It's also straightforward to [parallelize](#) because it breaks the input array into chunks that can be distributed and processed in parallel if necessary.

That said, for small lists, the time cost of the recursion allows algorithms such as bubble sort and insertion sort to be faster. For example, running an experiment with a list of ten elements results in the following times:

Shell

```
Algorithm: bubble_sort. Minimum execution time: 0.00001877499999998654
Algorithm: insertion_sort. Minimum execution time: 0.00002978600000000395
Algorithm: merge_sort. Minimum execution time: 0.00016983000000000276
```

Both bubble sort and insertion sort beat merge sort when sorting a ten-element list.

Another drawback of merge sort is that it creates copies of the array when calling itself recursively. It also creates a new list inside `merge()` to sort and return both input halves. This makes merge sort use much more memory than bubble sort and insertion sort, which are both able to sort the list in place.

Due to this limitation, you may not want to use merge sort to sort large lists in memory-constrained hardware.

The Quicksort Algorithm in Python

Just like merge sort, the **quicksort** algorithm applies the divide-and-conquer principle to divide the input array into two lists, the first with small items and the second with large items. The algorithm then sorts both lists recursively until the resultant list is completely sorted.

Dividing the input list is referred to as **partitioning** the list. Quicksort first selects a pivot element and partitions the list around the pivot, putting every smaller element into a `low` array and every larger element into a `high` array.

Putting every element from the `low` list to the left of the pivot and every element from the `high` list to the right positions the pivot precisely where it needs to be in the final sorted list. This means that the function can now recursively apply the same procedure to `low` and then `high` until the entire list is sorted.

Implementing Quicksort in Python

Here's a fairly compact implementation of quicksort:

Python

```
1 | from random import randint
2 |
3 | def quicksort(array):
4 |     # If the input array contains fewer than two elements,
5 |     # then return it as the result of the function
6 |     if len(array) < 2:
7 |         return array
8 |
9 |     low, same, high = [], [], []
10 |
11 |     # Select your `pivot` element randomly
12 |     pivot = array[randint(0, len(array) - 1)]
13 |
14 |     for item in array:
15 |         # Elements that are smaller than the `pivot` go to
16 |         # the `low` list. Elements that are larger than
17 |         # `pivot` go to the `high` list. Elements that are
18 |         # equal to `pivot` go to the `same` list.
19 |         if item < pivot:
20 |             low.append(item)
21 |         elif item == pivot:
22 |             same.append(item)
23 |         elif item > pivot:
24 |             high.append(item)
25 |
26 |     # The final result combines the sorted `low` list
27 |     # with the `same` list and the sorted `high` list
28 |     return quicksort(low) + same + quicksort(high)
```

Here's a summary of the code:

- **Line 6** stops the recursive function if the array contains fewer than two elements.
- **Line 12** selects the pivot element randomly from the list and proceeds to partition the list.
- **Lines 19 and 20** put every element that's smaller than pivot into the list called `low`.
- **Lines 21 and 22** put every element that's equal to pivot into the list called `same`.
- **Lines 23 and 24** put every element that's larger than pivot into the list called `high`.
- **Line 28** recursively sorts the `low` and `high` lists and combines them along with the contents of the `same` list.

Here's an illustration of the steps that quicksort takes to sort the array [8, 2, 6, 4, 5]:

The Quicksort Process

The yellow lines represent the partitioning of the array into three lists: `low`, `same`, and `high`. The green lines represent sorting and putting these lists back together. Here's a brief explanation of the steps:

1. The pivot element is selected randomly. In this case, pivot is 6.
2. The first pass partitions the input array so that `low` contains [2, 4, 5], `same` contains [6], and `high` contains [8].
3. `quicksort()` is then called recursively with `low` as its input. This selects a random pivot and breaks the array into [2] as `low`, [4] as `same`, and [5] as `high`.
4. The process continues, but at this point, both `low` and `high` have fewer than two items each. This ends the recursion, and the function puts the array back together. Adding the sorted `low` and `high` to either side of the `same` list produces [2, 4, 5].
5. On the other side, the `high` list containing [8] has fewer than two elements, so the algorithm returns the sorted `low` array, which is now [2, 4, 5]. Merging it with `same` ([6]) and `high` ([8]) produces the final sorted list.

Selecting the pivot Element

Why does the implementation above select the pivot element randomly? Wouldn't it be the same to consistently select the first or last element of the input list?

Because of how the quicksort algorithm works, the number of recursion levels depends on where pivot ends up in each partition. In the best-case scenario, the algorithm consistently picks the **median** element as the pivot. That would make each generated subproblem exactly half the size of the previous problem, leading to at most $\log_2 n$ levels.

On the other hand, if the algorithm consistently picks either the smallest or largest element of the array as the pivot, then the generated partitions will be as unequal as possible, leading to $n-1$ recursion levels. That would be the worst-case scenario for quicksort.

As you can see, quicksort's efficiency often depends on the pivot selection. If the input array is unsorted, then using

the first or last element as the pivot will work the same as a random element. But if the input array is sorted or almost sorted, using the first or last element as the pivot could lead to a worst-case scenario. Selecting the pivot at random makes it more likely quicksort will select a value closer to the median and finish faster.

Another option for selecting the pivot is to [find the median value of the array](#) and force the algorithm to use it as the pivot. This can be done in $O(n)$ time. Although the process is little bit more involved, using the median value as the pivot for quicksort guarantees you will have the best-case Big O scenario.

Measuring Quicksort's Big O Complexity

With quicksort, the input list is partitioned in linear time, $O(n)$, and this process repeats recursively an average of $\log_2 n$ times. This leads to a final complexity of $O(n \log_2 n)$.

That said, remember the discussion about how the selection of the pivot affects the runtime of the algorithm. The $O(n)$ best-case scenario happens when the selected pivot is close to the median of the array, and an $O(n^2)$ scenario happens when the pivot is the smallest or largest value of the array.

Theoretically, if the algorithm focuses first on finding the median value and then uses it as the pivot element, then the worst-case complexity will come down to $O(n \log_2 n)$. The median of an array can be found in linear time, and using it as the pivot guarantees the quicksort portion of the code will perform in $O(n \log_2 n)$.

By using the median value as the pivot, you end up with a final runtime of $O(n) + O(n \log_2 n)$. You can simplify this down to $O(n \log_2 n)$ because the logarithmic portion grows much faster than the linear portion.

Note: Although achieving $O(n \log_2 n)$ is possible in quicksort's worst-case scenario, this approach is seldom used in practice. Lists have to be quite large for the implementation to be faster than a simple randomized selection of the pivot.

Randomly selecting the pivot makes the worst case very unlikely. That makes random pivot selection good enough for most implementations of the algorithm.

Timing Your Quicksort Implementation

By now, you're familiar with the process for timing the runtime of the algorithm. Just change the name of the algorithm in [line 8](#):

Python

```
1 if __name__ == "__main__":
2     # Generate an array of `ARRAY_LENGTH` items consisting
3     # of random integer values between 0 and 999
4     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
5
6     # Call the function using the name of the sorting algorithm
7     # and the array you just created
8     run_sorting_algorithm(algorithm="quicksort", array=array)
```

You can execute the script as you have before:

Shell

```
$ python sorting.py
Algorithm: quicksort. Minimum execution time: 0.11675417600002902
```

Not only does quicksort finish in less than one second, but it's also much faster than merge sort (0.11 seconds versus 0.61 seconds). Increasing the number of elements specified by `ARRAY_LENGTH` from $10,000$ to $1,000,000$ and running the script again ends up with merge sort finishing in 97 seconds, whereas quicksort sorts the list in a mere 10 seconds.

Analyzing the Strengths and Weaknesses of Quicksort

True to its name, quicksort is *very fast*. Although its worst-case scenario is theoretically $O(n^2)$, in practice, a good implementation of quicksort beats most other sorting implementations. Also, just like merge sort, quicksort is straightforward to [parallelize](#).

One of quicksort's main disadvantages is the lack of a guarantee that it will achieve the average runtime complexity.

Although worst-case scenarios are rare, certain applications can't afford to risk poor performance, so they opt for algorithms that stay within $O(n \log_2 n)$ regardless of the input.

Just like merge sort, quicksort also trades off memory space for speed. This may become a limitation for sorting larger lists.

A quick experiment sorting a list of ten elements leads to the following results:

Shell

```
Algorithm: bubble_sort. Minimum execution time: 0.000090900000000014
Algorithm: insertion_sort. Minimum execution time: 0.0000668190000000268
Algorithm: quicksort. Minimum execution time: 0.0001319930000000004
```

The results show that quicksort also pays the price of recursion when the list is sufficiently small, taking longer to complete than both insertion sort and bubble sort.

The Timsort Algorithm in Python

The **Timsort** algorithm is considered a **hybrid** sorting algorithm because it employs a best-of-both-worlds combination of insertion sort and merge sort. Timsort is near and dear to the Python community because it was created by Tim Peters in 2002 to be used as the [standard sorting algorithm of the Python language](#).

The main characteristic of Timsort is that it takes advantage of already-sorted elements that exist in most real-world datasets. These are called **natural runs**. The algorithm then iterates over the list, collecting the elements into runs and merging them into a single sorted list.

Implementing Timsort in Python

In this section, you'll create a barebones Python implementation that illustrates all the pieces of the Timsort algorithm. If you're interested, you can also check out the [original C implementation of Timsort](#).

The first step in implementing Timsort is modifying the implementation of `insertion_sort()` from before:

Python

```
1 | def insertion_sort(array, left=0, right=None):
2 |     if right is None:
3 |         right = len(array) - 1
```

```

4
5     # Loop from the element indicated by
6     # `left` until the element indicated by `right`
7     for i in range(left + 1, right + 1):
8         # This is the element we want to position in its
9         # correct place
10        key_item = array[i]
11
12        # Initialize the variable that will be used to
13        # find the correct position of the element referenced
14        # by `key_item`
15        j = i - 1
16
17        # Run through the list of items (the left
18        # portion of the array) and find the correct position
19        # of the element referenced by `key_item`. Do this only
20        # if the `key_item` is smaller than its adjacent values.
21        while j >= left and array[j] > key_item:
22            # Shift the value one position to the left
23            # and reposition `j` to point to the next element
24            # (from right to left)
25            array[j + 1] = array[j]
26            j -= 1
27
28        # When you finish shifting the elements, position
29        # the `key_item` in its correct location
30        array[j + 1] = key_item
31
32    return array

```

This modified implementation adds a couple of parameters, `left` and `right`, that indicate which portion of the array should be sorted. This allows the Timsort algorithm to sort a portion of the array in place. Modifying the function instead of creating a new one means that it can be reused for both insertion sort and Timsort.

Now take a look at the implementation of Timsort:

Python

```

1 def timsort(array):
2     min_run = 32
3     n = len(array)
4
5     # Start by slicing and sorting small portions of the
6     # input array. The size of these slices is defined by
7     # your `min_run` size.
8     for i in range(0, n, min_run):

```

```

9      insertion_sort(array, i, min((i + min_run - 1), n - 1))
10
11  # Now you can start merging the sorted slices.
12  # Start from `min_run`, doubling the size on
13  # each iteration until you surpass the length of
14  # the array.
15  size = min_run
16  while size < n:
17      # Determine the arrays that will
18      # be merged together
19      for start in range(0, n, size * 2):
20          # Compute the `midpoint` (where the first array ends
21          # and the second starts) and the `endpoint` (where
22          # the second array ends)
23          midpoint = start + size - 1
24          end = min((start + size * 2 - 1), (n-1))
25
26          # Merge the two subarrays.
27          # The `left` array should go from `start` to
28          # `midpoint + 1`, while the `right` array should
29          # go from `midpoint + 1` to `end + 1`.
30          merged_array = merge(
31              left=array[start:midpoint + 1],
32              right=array[midpoint + 1:end + 1])
33
34          # Finally, put the merged array back into
35          # your array
36          array[start:start + len(merged_array)] = merged_array
37
38          # Each iteration should double the size of your arrays
39          size *= 2
40
41  return array

```

Although the implementation is a bit more complex than the previous algorithms, we can summarize it quickly in the following way:

- **Lines 8 and 9** create small slices, or runs, of the array and sort them using insertion sort. You learned previously that insertion sort is speedy on small lists, and Timsort takes advantage of this. Timsort uses the newly introduced `left` and `right` parameters in `insertion_sort()` to sort the list in place without having to create new arrays like merge sort and quicksort do.
- **Line 16** merges these smaller runs, with each run being of size 32 initially. With each iteration, the size of the runs is doubled, and the algorithm continues merging these larger runs until a single sorted run remains.

Notice how, unlike merge sort, Timsort merges subarrays that were previously sorted. Doing so decreases the total number of comparisons required to produce a sorted list. This advantage over merge sort will become apparent when running experiments using different arrays.

Finally, **line 2** defines `min_run = 32`. There are two reasons for using 32 as the value here:

1. Sorting small arrays using insertion sort is very fast, and `min_run` has a small value to take advantage of this characteristic. Initializing `min_run` with a value that's too large will defeat the purpose of using insertion sort and will make the algorithm slower.
2. Merging two balanced lists is much more efficient than merging lists of disproportionate size. Picking a `min_run` value that's a power of two ensures better performance when merging all the different runs that the algorithm creates.

Combining both conditions above offers several options for `min_run`. The implementation in this tutorial uses `min_run = 32` as one of the possibilities.

Note: In practice, Timsort does something a little more complicated to compute `min_run`. It picks a value between 32 and 64 inclusive, such that the length of the list divided by `min_run` is exactly a power of 2. If that's not possible, it chooses a value that's close to, but strictly less than, a power of 2.

If you're curious, you can read the [complete analysis on how to pick min_run](#) under the *Computing minrun* section.

Measuring Timsort's Big O Complexity

On average, the complexity of Timsort is $O(n \log_2 n)$, just like merge sort and quicksort. The logarithmic part comes from doubling the size of the run to perform each linear merge operation.

However, Timsort performs exceptionally well on already-sorted or close-to-sorted lists, leading to a best-case scenario of $O(n)$. In this case, Timsort clearly beats merge sort and matches the best-case scenario for quicksort. But the worst case for Timsort is also $O(n \log_2 n)$, which surpasses quicksort's $O(n^2)$.

Timing Your Timsort Implementation

You can use `run_sorting_algorithm()` to see how Timsort performs sorting the ten-thousand-element array:

Python

```
1 if __name__ == "__main__":
2     # Generate an array of `ARRAY_LENGTH` items consisting
3     # of random integer values between 0 and 999
4     array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
5
6     # Call the function using the name of the sorting algorithm
7     # and the array you just created
8     run_sorting_algorithm(algorithm="timsort", array=array)
```

Now execute the script to get the execution time of `timsort`:

Shell

```
$ python sorting.py
Algorithm: timsort. Minimum execution time: 0.5121690789999998
```

At 0.51 seconds, this Timsort implementation is a full 0.1 seconds, or 17 percent, faster than merge sort, though it doesn't match the 0.11 of quicksort. It's also a ridiculous 11,000 percent faster than insertion sort!

Now try to sort an already-sorted list using these four algorithms and see what happens. You can modify your `__main__` section as follows:

Python

```
1 if __name__ == "__main__":
2     # Generate a sorted array of ARRAY_LENGTH items
3     array = [i for i in range(ARRAY_LENGTH)]
4
5     # Call each of the functions
6     run_sorting_algorithm(algorithm="insertion_sort", array=array)
7     run_sorting_algorithm(algorithm="merge_sort", array=array)
8     run_sorting_algorithm(algorithm="quicksort", array=array)
9     run_sorting_algorithm(algorithm="timsort", array=array)
```

If you execute the script now, then all the algorithms will run and output their corresponding execution time:

Shell

```
Algorithm: insertion_sort. Minimum execution time: 53.5485634999991
Algorithm: merge sort. Minimum execution time: 0.372304601
```

```
Algorithm: quicksort. Minimum execution time: 0.24626494199999982
Algorithm: timsort. Minimum execution time: 0.23350277099999994
```

This time, Timsort comes in at a whopping thirty-seven percent faster than merge sort and five percent faster than quicksort, flexing its ability to take advantage of the already-sorted runs.

Notice how Timsort benefits from two algorithms that are much slower when used by themselves. The genius of Timsort is in combining these algorithms and playing to their strengths to achieve impressive results.

Analyzing the Strengths and Weaknesses of Timsort

The main disadvantage of Timsort is its complexity. Despite implementing a very simplified version of the original algorithm, it still requires much more code because it relies on both `insertion_sort()` and `merge()`.

One of Timsort's advantages is its ability to predictably perform in $O(n \log_2 n)$ regardless of the structure of the input array. Contrast that with quicksort, which can degrade down to $O(n^2)$. Timsort is also very fast for small arrays because the algorithm turns into a single insertion sort.

For real-world usage, in which it's common to sort arrays that already have some preexisting order, Timsort is a great option. Its adaptability makes it an excellent choice for sorting arrays of any length.

Conclusion

Sorting is an essential tool in any Pythonista's toolkit. With knowledge of the different sorting algorithms in Python and how to maximize their potential, you're ready to implement faster, more efficient apps and programs!

In this tutorial, you learned:

- How Python's built-in `sort()` works behind the scenes
- What **Big O notation** is and how to use it to compare the efficiency of different algorithms
- How to measure the **actual time spent** running your code
- How to implement five different **sorting algorithms** in Python
- What the **pros and cons** are of using different algorithms

You also learned about different techniques such as **recursion, divide and conquer**, and **randomization**. These are fundamental building blocks for solving a long list of different algorithms, and they'll come up again and again as you keep researching.

Take the code presented in this tutorial, create new experiments, and explore these algorithms further. Better yet, try implementing other sorting algorithms in Python. The list is vast, but **selection sort**, **heapsort**, and **tree sort** are three excellent options to start with.

About Santiago Valdarrama

Santiago is a software and machine learning engineer who specializes in building enterprise software applications.

[» More about Santiago](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

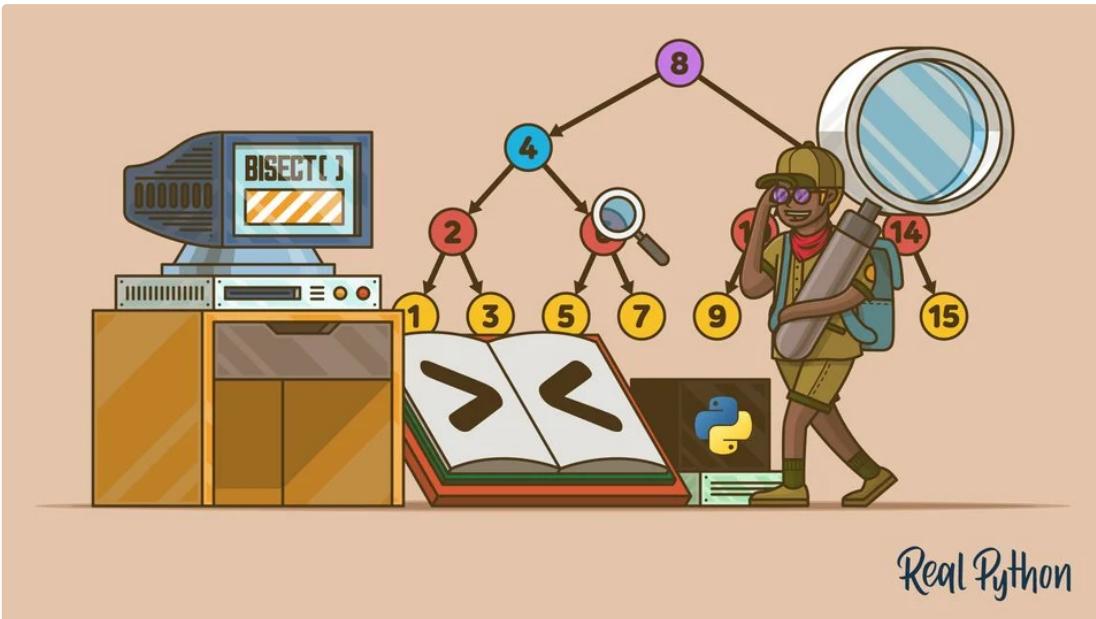
[Jon](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Real Python

How to Do a Binary Search in Python

by Bartosz Zaczyński ⌂ Mar 16, 2020 🗣 6 Comments 🏷 [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Benchmarking](#)
 - [Download IMDb](#)
 - [Read Tab-Separated Values](#)
 - [Measure the Execution Time](#)
- [Understanding Search Algorithms](#)
 - [Random Search](#)
 - [Linear Search](#)
 - [Binary Search](#)
 - [Hash-Based Search](#)
- [Using the bisect Module](#)
 - [Finding an Element](#)
 - [Inserting a New Element](#)
- [Implementing Binary Search in Python](#)
 - [Iteratively](#)
 - [Recursively](#)
- [Covering Tricky Details](#)
 - [Integer Overflow](#)
 - [Stack Overflow](#)
 - [Duplicate Elements](#)
 - [Floating-Point Rounding](#)
- [Analyzing the Time-Space Complexity of Binary Search](#)
 - [Time-Space Complexity](#)
 - [The Big O Notation](#)
 - [The Complexity of Binary Search](#)
- [Conclusion](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

Binary search is a classic algorithm in computer science. It often comes up in programming contests and [technical](#)

[interviews](#). Implementing binary search turns out to be a challenging task, even when you understand the concept. Unless you're curious or have a specific assignment, you should always leverage existing libraries to do a binary search in Python or any other language.

In this tutorial, you'll learn how to:

- Use the `bisect` module to do a binary search in Python
- Implement a binary search in Python both **recursively** and **iteratively**
- Recognize and fix **defects** in a binary search Python implementation
- Analyze the **time-space complexity** of the binary search algorithm
- Search even **faster** than binary search

This tutorial assumes you're a student or an [intermediate programmer](#) with an interest in algorithms and [data structures](#). At the very least, you should be familiar with Python's built-in data types, such as [lists and tuples](#). In addition, some familiarity with [recursion](#), [classes](#), [data classes](#), and [lambdas](#) will help you better understand the concepts you'll see in this tutorial.

Below you'll find a link to the sample code you'll see throughout this tutorial, which requires Python 3.7 or later to run:

Get Sample Code: [Click here to get the sample code you'll use](#) to learn about binary search in Python in this tutorial.

Benchmarking

In the next section of this tutorial, you'll be using a subset of the [Internet Movie Database \(IMDb\)](#) to benchmark the performance of a few search algorithms. This dataset is free of charge for personal and non-commercial use. It's distributed as a bunch of compressed [tab-separated values \(TSV\)](#) files, which get daily updates.

To make your life easier, you can use a Python script included in the sample code. It'll automatically fetch the relevant file from IMDb, decompress it, and extract the interesting pieces:

Shell

```
$ python download_imdb.py
Fetching data from IMDb...
Created "names.txt" and "sorted_names.txt"
```

Be warned that this will download and extract approximately 600 MB of data, as well as produce two additional files, which are about half of that in size. The download, as well as the processing of this data, might take a minute or two to complete.

Download IMDb

To manually obtain the data, navigate your web browser to <https://datasets.imdbws.com/> and grab the file called `name.basics.tsv.gz`, which contains the records of actors, directors, writers, and so on. When you decompress the file, you'll see the following content:

Text

nconst	primaryName	birthYear	deathYear	(...)
nm0000001	Fred Astaire	1899	1987	(...)
nm0000002	Lauren Bacall	1924	2014	(...)
nm0000003	Brigitte Bardot	1934	\N	(...)
nm0000004	John Belushi	1949	1982	(...)

It has a **header** with the column names in the first line, followed by **data records** in each of the subsequent lines. Each record contains a unique identifier, a full name, birth year, and a few other attributes. These are all delimited with a tab character.

There are millions of records, so don't try to open the file with a regular text editor to avoid crashing your computer. Even specialized software such as spreadsheets can have problems opening it. Instead, you might take advantage of

the high-performance data grid viewer included in [JupyterLab](#), for example.

Read Tab-Separated Values

There are a few ways to parse a TSV file. For example, you can read it with [Pandas](#), use a dedicated application, or leverage a few command-line tools. However, it's recommended that you use the hassle-free Python script included in the sample code.

Note: As a rule of thumb, you should avoid parsing files manually because you might overlook **edge cases**. For example, in one of the fields, the delimiting tab character could be used literally inside quotation marks, which would break the number of columns. Whenever possible, try to find a relevant module in the standard library or a trustworthy third-party one.

Ultimately, you want to end up with two text files at your disposal:

1. names.txt
2. sorted_names.txt

One will contain a [list](#) of names obtained by cutting out the second column from the original TSV file:

Text

```
Fred Astaire
Lauren Bacall
Brigitte Bardot
John Belushi
Ingmar Bergman
...
```

The second one will be the sorted version of this.

Once both files are ready, you can load them into Python using this function:

Python

```
def load_names(path):
    with open(path) as text_file:
        return text_file.read().splitlines()

names = load_names('names.txt')
sorted_names = load_names('sorted_names.txt')
```

This code returns a list of names pulled from the given file. Note that calling `.splitlines()` on the resulting [string](#) removes the trailing newline character from each line. As an alternative, you could call `text_file.readlines()`, but that would keep the unwanted newlines.

Measure the Execution Time

To evaluate the performance of a particular algorithm, you can measure its execution time against the IMDb dataset. This is usually done with the help of the built-in [time](#) or [timeit](#) modules, which are useful for timing a block of code.

You could also define a custom [decorator](#) to time a function if you wanted to. The sample code provided uses [`time.perf_counter_ns\(\)`](#), introduced in Python 3.7, because it offers high precision in nanoseconds.

Understanding Search Algorithms

Searching is ubiquitous and lies at the heart of computer science. You probably did several web searches today alone, but have you ever wondered what **searching** really means?

Search algorithms take many different forms. For example, you can:

- Do a [full-text search](#)
- Match strings with [fuzzy searching](#)
- Find the shortest path in a graph

- Query a database
- Look for a minimum or maximum value

In this tutorial, you'll learn about searching for an element in a sorted list of items, like a phone book. When you search for such an element, you might be asking one of the following questions:

Question	Answer
Is it there?	Yes
Where is it?	On the 42nd page
Which one is it?	A person named John Doe

The answer to the first question tells you whether an element is **present** in the collection. It always holds either *true* or *false*. The second answer is the **location** of an element within the collection, which may be unavailable if that element was missing. Finally, the third answer is the **element** itself, or a lack of it.

Note: Sometimes there might be more than one correct answer due to **duplicate or similar items**. For example, if you have a few contacts with the same name, then they will all fit your search criteria. At other times, there might only be an approximate answer or no answer at all.

In the most common case, you'll be **searching by value**, which compares elements in the collection against the exact one you provide as a reference. In other words, your search criteria are the entire element, such as a number, a string, or an object like a person. Even the tiniest difference between the two compared elements won't result in a match.

On the other hand, you can be more granular with your search criteria by choosing some property of an element, such as a person's last name. This is called **searching by key** because you pick one or more attributes to compare. Before you dive into binary search in Python, let's take a quick look at other search algorithms to get a bigger picture and understand how they work.

Random Search

How might you look for something in your backpack? You might just dig your hand into it, pick an item at random, and see if it's the one you wanted. If you're out of luck, then you put the item back, rinse, and repeat. This example is a good way to understand **random search**, which is one of the least efficient search algorithms. The inefficiency of this approach stems from the fact that you're running the risk of picking the same wrong thing multiple times.

Note: Funnily enough, this strategy *could* be the most efficient one, in theory, if you were very lucky or had a small number of elements in the collection.

The fundamental principle of this algorithm can be expressed with the following snippet of Python code:

Python

```
import random

def find(elements, value):
    while True:
        random_element = random.choice(elements)
        if random_element == value:
            return random_element
```

The function loops until some element chosen at `random` matches the value given as input. However, this isn't very useful because the function returns either `None` implicitly or the same value it already received in a parameter. You can find the full implementation in the sample code available for download at the link below:

Get Sample Code: [Click here to get the sample code you'll use](#) to learn about binary search in Python in this tutorial.

For microscopic datasets, the random search algorithm appears to be doing its job reasonably fast:

```
Python >>>
>>> from search.random import * # Sample code to download
>>> fruits = ['orange', 'plum', 'banana', 'apple']
>>> contains(fruits, 'banana')
True
>>> find_index(fruits, 'banana')
2
>>> find(fruits, key=len, value=4)
'plum'
```

However, imagine having to search like that through *millions* of elements! Here's a quick rundown of a performance test that was done against the IMDb dataset:

Search Term	Element Index	Best Time	Average Time	Worst Time
<i>Fred Astaire</i>	0	0.74s	21.69s	43.16s
<i>Alicia Monica</i>	4,500,000	1.02s	26.17s	66.34s
<i>Baoyin Liu</i>	9,500,000	0.11s	17.41s	51.03s
<i>missing</i>	N/A	5m 16s	5m 40s	5m 54s

Unique elements at different memory locations were specifically chosen to avoid bias. Each term was searched for ten times to account for the randomness of the algorithm and other factors such as garbage collection or system processes running in the background.

Note: If you'd like to conduct this experiment yourself, then refer back to the instructions in the introduction to this tutorial. To measure the performance of your code, you can use the built-in [time](#) and [timeit](#) modules, or you can time functions with a custom [decorator](#).

The algorithm has a **non-deterministic performance**. While the average time to find an element doesn't depend on its whereabouts, the best and worst times are two to three orders of magnitude apart. It also suffers from inconsistent behavior. Consider having a collection of elements containing some duplicates. Because the algorithm picks elements at random, it'll inevitably return different copies upon subsequent runs.

How can you improve on this? One way to address both issues at once is by using a **linear search**.

Linear Search

When you're deciding what to have for lunch, you may be looking around the menu chaotically until something catches your eye. Alternatively, you can take a more systematic approach by scanning the menu from top to bottom and scrutinizing every item in a **sequence**. That's linear search in a nutshell. To implement it in Python, you could [enumerate\(\)](#) elements to keep track of the current element's index:

```
Python
def find_index(elements, value):
    for index, element in enumerate(elements):
        if element == value:
            return index
```

The function loops over a collection of elements in a predefined and consistent order. It stops when the element is found, or when there are no more elements to check. This strategy guarantees that no element is visited more than once because you're traversing them in order by index.

Let's see how well linear search copes with the IMDb dataset you used before:

Search Term	Element Index	Best Time	Average Time	Worst Time
<i>Fred Astaire</i>	0	491ns	1.17µs	6.1µs
<i>Alicia Monica</i>	4,500,000	0.37s	0.38s	0.39s
<i>Baoyin Liu</i>	9,500,000	0.77s	0.79s	0.82s
<i>missing</i>	N/A	0.79s	0.81s	0.83s

There's hardly any variance in the lookup time of an individual element. The average time is virtually the same as the best and the worst one. Since the elements are always browsed in the same order, the number of comparisons required to find the same element doesn't change.

However, the lookup time grows with the increasing index of an element in the collection. The further the element is from the beginning of the list, the more comparisons have to run. In the worst case, when an element is missing, the whole collection has to be checked to give a definite answer.

When you project experimental data onto a plot and connect the dots, then you'll immediately see the relationship between element location and the time it takes to find it:

All samples lie on a straight line and can be described by a **linear function**, which is where the name of the algorithm comes from. You can assume that, on average, the time required to find any element using a linear search will be proportional to the number of all elements in the collection. They don't scale well as the amount of data to search increases.

For example, biometric scanners available at some airports wouldn't recognize passengers in a matter of seconds, had they been implemented using linear search. On the other hand, the linear search algorithm may be a good choice for smaller datasets, because it doesn't require **preprocessing** the data. In such a case, the benefits of preprocessing wouldn't pay back its cost.

Python already ships with linear search, so there's no point in writing it yourself. The `list` data structure, for example, exposes a method that will return the index of an element or raise an exception otherwise:

```
Python >>>
>>> fruits = ['orange', 'plum', 'banana', 'apple']
>>> fruits.index('banana')
2
>>> fruits.index('blueberry')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'blueberry' is not in list
```

This can also tell you if the element is present in the collection, but a more [Pythonic](#) way would involve using the versatile `in` operator:

```
Python >>>
>>> 'banana' in fruits
True
>>> 'blueberry' in fruits
False
```

It's worth noting that despite using linear search under the hood, these built-in functions and operators will blow your implementation out of the water. That's because they were written in pure [C](#), which compiles to native machine code. The standard Python interpreter is no match for it, no matter how hard you try.

A quick test with the `timeit` module reveals that the Python implementation might run almost ten times slower than the equivalent native one:

```
Python >>>
>>> import timeit
>>> from search.linear import contains
>>> fruits = ['orange', 'plum', 'banana', 'apple']
>>> timeit.timeit(lambda: contains(fruits, 'blueberry'))
1.8904765040024358
>>> timeit.timeit(lambda: 'blueberry' in fruits)
0.22473459799948614
```

However, for sufficiently large datasets, even the native code will hit its limits, and the only solution will be to rethink the algorithm.

Note: The `in` operator doesn't always do a linear search. When you use it on a `set`, for example, it does a hash-based search instead. The operator can work with any [iterable](#), including `tuple`, `list`, `set`, `dict`, and `str`. You can even support your custom classes with it by implementing the [magic method](#) `__contains__()` to define the underlying logic.

In real-life scenarios, the linear search algorithm should usually be avoided. For example, there was a time I wasn't able to register my cat at the vet clinic because their system kept crashing. The doctor told me he must eventually upgrade his computer because adding more records into the database makes it run slower and slower.

I remember thinking to myself at that point that the person who wrote that software clearly didn't know about the [binary search](#) algorithm!

Binary Search

The word **binary** is generally associated with the number 2. In this context, it refers to dividing a collection of elements into two halves and throwing away one of them at each step of the algorithm. This can dramatically reduce the number of comparisons required to find an element. But there's a catch—elements in the collection must be [sorted](#) first.

The idea behind it resembles the steps for finding a page in a book. At first, you typically open the book to a completely random page or at least one that's close to where you think your desired page might be.

Occasionally, you'll be fortunate enough to find that page on the first try. However, if the page number is too low, then you know the page must be to the right. If you overshoot on the next try, and the current page number is higher than the page you're looking for, then you know for sure that it must be somewhere in between.

You repeat the process, but rather than choosing a page at random, you check the page located right in the **middle** of that new range. This minimizes the number of tries. A similar approach can be used in the [number guessing game](#). If you haven't heard of that game, then you can look it up on the Internet to get a plethora of examples implemented in Python.

Note: Sometimes, if the values are uniformly distributed, you can calculate the middle index with [linear interpolation](#) rather than taking the average. This variation of the algorithm will require even fewer steps.

The page numbers that restrict the range of pages to search through are known as the **lower bound** and the **upper bound**. In binary search, you commonly start with the first page as the lower bound and the last page as the upper bound. You must update both bounds as you go. For example, if the page you turn to is lower than the one you're looking for, then that's your new lower bound.

Let's say you were looking for a strawberry in a collection of fruits sorted in ascending order by size:

On the first attempt, the element in the middle happens to be a lemon. Since it's bigger than a strawberry, you can discard all elements to the right, including the lemon. You'll move the upper bound to a new position and update the middle index:

Now, you're left with only half of the fruits you began with. The current middle element is indeed the strawberry you were looking for, which concludes the search. If it wasn't, then you'd just update the bounds accordingly and continue until they pass each other. For example, looking for a missing plum, which would go between the strawberry and a kiwi, will end with the following result:

Notice there weren't that many comparisons that had to be made in order to find the desired element. That's the magic of binary search. Even if you're dealing with a million elements, you'd only require at most a handful of checks. This number won't exceed the **logarithm** base two of the total number of elements due to halving. In other words, the number of remaining elements is reduced by half at each step.

This is possible because the elements are already sorted by size. However, if you wanted to find fruits by another key, such as a color, then you'd have to sort the entire collection once again. To avoid the costly overhead of sorting, you might try to compute different views of the same collection in advance. This is somewhat similar to creating a **database index**.

Consider what happens if you add, delete or update an element in a collection. For a binary search to continue working, you'd need to maintain the proper sort order. This can be done with the `bisect` module, which you'll read about in the upcoming section.

You'll see how to implement the binary search algorithm in Python later on in this tutorial. For now, let's confront it with the IMDb dataset. Notice that there are different people to search for than before. That's because the dataset must be sorted for binary search, which reorders the elements. The new elements are located roughly at the same indices as before, to keep the measurements comparable:

Search Term Search Term	Element Index Element Index	Average Time Average Time	Comparisons Comparisons
(...) Berendse	0	6.52µs	23
Jonathan Samuagte	4, 499, 997	6.99µs	24
Yorgos Rahmatoulin	9, 500, 001	6.5µs	23
missing	N/A	7.2µs	23

The answers are nearly instantaneous. In the average case, it takes only a few microseconds for the binary search to find one element among all nine million! Other than that, the number of comparisons for the chosen elements remains almost constant, which coincides with the following formula:

Finding most elements will require the highest number of comparisons, which can be derived from a logarithm of the size of the collection. Conversely, there's just one element in the middle that can be found on the first try with one comparison.

Binary search is a great example of a **divide-and-conquer** technique, which partitions one problem into a bunch of smaller problems of the same kind. The individual solutions are then combined to form the final answer. Another well-known example of this technique is the [quicksort](#) algorithm.

Note: Don't confuse divide-and-conquer with [dynamic programming](#), which is a somewhat similar technique.

Unlike other search algorithms, binary search can be used beyond just searching. For example, it allows for set membership testing, finding the largest or smallest value, finding the nearest neighbor of the target value, performing range queries, and more.

If speed is a top priority, then binary search is not always the best choice. There are even faster algorithms that can take advantage of hash-based data structures. However, those algorithms require a lot of additional memory, whereas binary search offers a good [space-time tradeoff](#).

Hash-Based Search

To search faster, you need to narrow down the **problem space**. Binary search achieves that goal by halving the number of candidates at each step. That means that even if you have one million elements, it takes at most twenty comparisons to determine if the element is present, provided that all elements are sorted.

The fastest way to search is to know where to find what you're looking for. If you knew the exact memory location of an element, then you'd access it directly without the need for searching in the first place. Mapping an element or (more commonly) one of its keys to the element location in memory is referred to as **hashing**.

You can think of hashing not as searching for the specific element, but instead computing the index based on the element itself. That's the job of a [hash function](#), which needs to hold certain mathematical properties. A good hash function should:

- Take arbitrary input and turn it into a fixed-size output.
- Have a uniform value distribution to mitigate [hash collisions](#).
- Produce deterministic results.
- Be a [one-way function](#).
- Amplify input changes to achieve the [avalanche effect](#).

At the same time, it shouldn't be too computationally expensive, or else its cost would outweigh the gains. A hash function is also used for data integrity verification as well as in cryptography.

A data structure that uses this concept to map keys into values is called a **map**, a **hash table**, a **dictionary**, or an **associative array**.

Note: Python has two built-in data structures, namely `set` and `dict`, which rely on the `hash` function to find elements. While a set hashes its elements, a dict uses the hash function against element keys. To find out exactly how a dict is implemented in Python, check out Raymond Hettinger's conference talk on [Modern Python Dictionaries](#).

Another way to visualize hashing is to imagine so-called **buckets** of similar elements grouped under their respective keys. For example, you may be harvesting fruits into different buckets based on color:

The coconut and a kiwi fruit go to the bucket labeled *brown*, while an apple ends up in a bucket with the *red* label, and so on. This allows you to glance through a fraction of the elements quickly. Ideally, you want to have only one fruit in each bucket. Otherwise, you get what's known as a **collision**, which leads to extra work.

Note: The buckets, as well as their contents, are typically in no particular order.

Let's put the names from the IMDb dataset into a dictionary, so that each name becomes a key, and the corresponding value becomes its index:

```
Python >>>
>>> from benchmark import load_names # Sample code to download
>>> names = load_names('names.txt')
>>> index_by_name = {
...     name: index for index, name in enumerate(names)
... }
```

After loading textual names into a flat list, you can `enumerate()` it inside a [dictionary comprehension](#) to create the mapping. Now, checking the element's presence as well as getting its index is straightforward:

```
Python >>>
>>> 'Guido van Rossum' in index_by_name
False
>>> 'Arnold Schwarzenegger' in index_by_name
True
>>> index_by_name['Arnold Schwarzenegger']
215
```

Thanks to the hash function used behind the scenes, you don't have to implement any search at all!

Here's how the hash-based search algorithm performs against the IMDb dataset:

Search Term	Element Index	Best Time	Average Time	Worst Time
<i>Fred Astaire</i>	0	0.18µs	0.4µs	1.9µs
<i>Alicia Monica</i>	4,500,000	0.17µs	0.4µs	2.4µs
<i>Baoyin Liu</i>	9,500,000	0.17µs	0.4µs	2.6µs
<i>missing</i>	N/A	0.19µs	0.4µs	1.7µs

Not only is the average time an order of magnitude faster than the already fast binary search Python implementation, but the speed is also sustained across all elements regardless of where they are.

The price for that gain is approximately 0.5 GB of more memory consumed by the Python process, slower load time, and the need to keep that additional data consistent with [dictionary](#) contents. In turn, the lookup is very quick, while updates and insertions are slightly slower when compared to a list.

Another constraint that dictionaries impose on their keys is that they must be **hashable**, and their **hash values** can't change over time. You can check if a particular data type is hashable in Python by calling `hash()` on it:

```
Python >>>
>>> key = ['round', 'juicy']
>>> hash(key)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Mutable collections—such as a list, set, and dict—are not hashable. In practice, dictionary keys should be **immutable** because their hash value often depends on some attributes of the key. If a mutable collection was hashable and could be used as a key, then its hash value would be different every time the contents changed. Consider what would happen if a particular fruit changed color due to ripening. You'd be looking for it in the wrong bucket!

The `hash` function has many other uses. For example, it's used in cryptography to avoid storing passwords in plain text form, as well as for data integrity verification.

Using the `bisect` Module

Binary search in Python can be performed using the built-in `bisect` module, which also helps with preserving a list in sorted order. It's based on the [bisection method](#) for finding roots of functions. This module comes with six functions divided into two categories:

Find Index	Insert Element
<code>bisect()</code>	<code>insort()</code>
<code>bisect_left()</code>	<code>insort_left()</code>
<code>bisect_right()</code>	<code>insort_right()</code>

These functions allow you to either find an index of an element or add a new element in the right position. Those in the first row are just aliases for `bisect_right()` and `insort_right()`, respectively. In reality, you're dealing with only four functions.

Note: It's your responsibility to sort the list before passing it to one of the functions. If the elements aren't sorted, then you'll most likely get incorrect results.

Without further ado, let's see the `bisect` module in action.

Finding an Element

To find the index of an existing element in a sorted list, you want to `bisect_left()`:

```
Python >>>
>>> import bisect
>>> sorted_fruits = ['apple', 'banana', 'orange', 'plum']
>>> bisect.bisect_left(sorted_fruits, 'banana')
1
```

The output tells you that a banana is the second fruit on the list because it was found at index 1. However, if an element was missing, then you'd still get its expected position:

```
Python >>> bisect.bisect_left(sorted_fruits, 'apricot')
1
>>> bisect.bisect_left(sorted_fruits, 'watermelon')
4
```

Even though these fruits aren't on the list yet, you can get an idea of where to put them. For example, an apricot should come between the apple and the banana, whereas a watermelon should become the last element. You'll know if an element was found by evaluating two conditions:

1. Is the **index** within the size of the list?
2. Is the **value** of the element the desired one?

This can be translated to a universal function for finding elements by value:

```
Python
def find_index(elements, value):
    index = bisect.bisect_left(elements, value)
    if index < len(elements) and elements[index] == value:
        return index
```

When there's a match, the function will return the corresponding element index. Otherwise, it'll return None implicitly.

To search by key, you have to maintain a separate list of keys. Since this incurs an additional cost, it's worthwhile to calculate the keys upfront and reuse them as much as possible. You can define a helper class to be able to search by different keys without introducing much code duplication:

```
Python
class SearchBy:
    def __init__(self, key, elements):
        self.elements_by_key = sorted([(key(x), x) for x in elements])
        self.keys = [x[0] for x in self.elements_by_key]
```

The key is a function passed as the first parameter to `__init__()`. Once you have it, you make a sorted list of key-value pairs to be able to retrieve an element from its key at a later time. Representing pairs with [tuples](#) guarantees that the first element of each pair will be sorted. In the next step, you extract the keys to make a flat list that's suitable for your binary search Python implementation.

Then there's the actual method for finding elements by key:

```
Python
class SearchBy:
    def __init__(self, key, elements):
        ...
        self.elements_by_key = sorted([(key(x), x) for x in elements])
        self.keys = [x[0] for x in self.elements_by_key]

    def find(self, value):
        index = bisect.bisect_left(self.keys, value)
        if index < len(self.keys) and self.keys[index] == value:
            return self.elements_by_key[index][1]
```

This code bisects the list of sorted keys to get the index of an element by key. If such a key exists, then its index can be used to get the corresponding pair from the previously computed list of key-value pairs. The second element of that pair is the desired value.

Note: This is just an illustrative example. You'll be better off using the [recommended recipe](#), which is mentioned in the official documentation.

If you had multiple bananas, then `bisect_left()` would return the leftmost instance:

```
Python >>>
>>> sorted_fruits = [
...     'apple',
...     'banana', 'banana', 'banana',
...     'orange',
...     'plum'
... ]
>>> bisect.bisect_left(sorted_fruits, 'banana')
1
```

Predictably, to get the rightmost banana, you'd need to call `bisect_right()` or its `bisect()` alias. However, those two functions return one index further from the actual rightmost banana, which is useful for finding the insertion point of a new element:

```
Python >>>
>>> bisect.bisect_right(sorted_fruits, 'banana')
4
>>> bisect.bisect(sorted_fruits, 'banana')
4
>>> sorted_fruits[4]
'orange'
```

When you combine the code, you can see how many bananas you have:

```
Python >>>
>>> l = bisect.bisect_left(sorted_fruits, 'banana')
>>> r = bisect.bisect_right(sorted_fruits, 'banana')
>>> r - l
3
```

If an element were missing, then both `bisect_left()` and `bisect_right()` would return the same index yielding zero bananas.

Inserting a New Element

Another practical application of the `bisect` module is maintaining the order of elements in an already sorted list. After all, you wouldn't want to sort the whole list every time you had to insert something into it. In most cases, all three functions can be used interchangeably:

```
Python >>>
>>> import bisect
>>> sorted_fruits = ['apple', 'banana', 'orange']
>>> bisect.insort(sorted_fruits, 'apricot')
>>> bisect.insort_left(sorted_fruits, 'watermelon')
>>> bisect.insort_right(sorted_fruits, 'plum')
>>> sorted_fruits
['apple', 'apricot', 'banana', 'orange', 'plum', 'watermelon']
```

You won't see any difference until there are **duplicates** in your list. But even then, it won't become apparent as long as those duplicates are simple values. Adding another banana to the left will have the same effect as adding it to the right.

To notice the difference, you need a data type whose objects can have **unique identities** despite having **equal values**. Let's define a `Person` type using the [@dataclass decorator](#), which was introduced in Python 3.7:

```
Python >>>
from dataclasses import dataclass, field

@dataclass
class Person:
```

```

name: str
number: int = field(compare=False)

def __repr__(self):
    return f'{self.name}({self.number})'

```

A person has a name and an arbitrary number assigned to it. By excluding the number field from the equality test, you make two people equal even if they have different values of that attribute:

Python

```

>>> p1 = Person('John', 1)
>>> p2 = Person('John', 2)
>>> p1 == p2
True

```

>>>

On the other hand, those two variables refer to completely separate entities, which allows you to make a distinction between them:

Python

```

>>> p1 is p2
False
>>> p1
John(1)
>>> p2
John(2)

```

>>>

The variables p1 and p2 are indeed different objects.

Note that instances of a data class aren't comparable by default, which prevents you from using the bisection algorithm on them:

Python

```

>>> alice, bob = Person('Alice', 1), Person('Bob', 1)
>>> alice < bob
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Person' and 'Person'

```

>>>

Python doesn't know to order alice and bob, because they're objects of a custom class. Traditionally, you'd implement the magic method `__lt__()` in your class, which stands for **less than**, to tell the interpreter how to compare such elements. However, the `@dataclass` decorator accepts a few optional Boolean flags. One of them is `order`, which results in an automatic generation of the magic methods for comparison when set to `True`:

Python

```

@dataclass(order=True)
class Person:
    ...

```

In turn, this allows you to compare two people and decide which one comes first:

Python

```

>>> alice < bob
True
>>> bob < alice
False

```

>>>

Finally, you can take advantage of the `name` and `number` properties to observe where various functions insert new people to the list:

people to the list.

Python

>>>

```
>>> sorted_people = [Person('John', 1)]
>>> bisect.insort_left(sorted_people, Person('John', 2))
>>> bisect.insort_right(sorted_people, Person('John', 3))
>>> sorted_people
[John(2), John(1), John(3)]
```

The numbers in parentheses after the names indicate the insertion order. In the beginning, there was just one John, who got the number 1. Then, you added its duplicate to the left, and later one more to the right.

Implementing Binary Search in Python

Keep in mind that you probably shouldn't implement the algorithm unless you have a strong reason to. You'll save time and won't need to reinvent the wheel. The chances are that the library code is mature, already tested by real users in a production environment, and has extensive functionality delivered by multiple contributors.

That said, there are times when it makes sense to roll up your sleeves and do it yourself. Your company might have a policy banning certain open source libraries due to licensing or security matters. Maybe you can't afford another dependency due to memory or network bandwidth constraints. Lastly, writing code yourself might be a great learning tool!

You can implement most algorithms in two ways:

1. **Iteratively**

2. **Recursively**

However, there are exceptions to that rule. One notable example is the [Ackermann function](#), which can only be expressed in terms of recursion.

Before you go any further, make sure that you have a good grasp of the binary search algorithm. You can refer to an [earlier](#) part of this tutorial for a quick refresher.

Iteratively

The iterative version of the algorithm involves a loop, which will repeat some steps until the stopping condition is met. Let's begin by implementing a function that will **search elements by value** and return their index:

Python

```
def find_index(elements, value):
    ...
```

You're going to reuse this function later.

Assuming that all elements are *sorted*, you can set the lower and the upper boundaries at the opposite ends of the sequence:

Python

```
def find_index(elements, value):
    left, right = 0, len(elements) - 1
```

Now, you want to identify the middle element to see if it has the desired value. Calculating the middle index can be done by taking the average of both boundaries:

Python

```
def find_index(elements, value):
    left, right = 0, len(elements) - 1
    middle = (left + right) // 2
```

Notice how an **integer division** helps to handle both an odd and even number of elements in the bounded range by

flooring the result. Depending on how you're going to update the boundaries and define the stopping condition, you could also use a [ceiling function](#).

Next, you either finish or split the sequence in two and continue searching in one of the resultant halves:

Python

```
def find_index(elements, value):
    left, right = 0, len(elements) - 1
    middle = (left + right) // 2

    if elements[middle] == value:
        return middle

    if elements[middle] < value:
        left = middle + 1
    elif elements[middle] > value:
        right = middle - 1
```

If the element in the middle was a match, then you return its index. Otherwise, if it was too small, then you need to move the lower boundary up. If it was too big, then you need to move the upper boundary down.

To keep going, you have to enclose most of the steps in a loop, which will stop when the lower boundary overtakes the upper one:

Python

```
def find_index(elements, value):
    left, right = 0, len(elements) - 1

    while left <= right:
        middle = (left + right) // 2

        if elements[middle] == value:
            return middle

        if elements[middle] < value:
            left = middle + 1
        elif elements[middle] > value:
            right = middle - 1
```

In other words, you want to iterate as long as the lower boundary is below or equal to the upper one. Otherwise, there was no match, and the function returns None implicitly.

Searching by key boils down to looking at an object's attributes instead of its literal value. A key could be the number of characters in a fruit's name, for example. You can adapt `find_index()` to accept and use a `key` parameter:

Python

```
def find_index(elements, value, key):
    left, right = 0, len(elements) - 1

    while left <= right:
        middle = (left + right) // 2
        middle_element = key(elements[middle])

        if middle_element == value:
            return middle

        if middle_element < value:
            left = middle + 1
        elif middle_element > value:
            right = middle - 1
```

However, you must also remember to sort the list using the same key that you're going to search with:

Python

>>>

```
>>> fruits = ['orange', 'plum', 'watermelon', 'apple']
>>> fruits.sort(key=len)
>>> fruits
['plum', 'apple', 'orange', 'watermelon']
>>> fruits[find_index(fruits, key=len, value=10)]
'watermelon'
>>> print(find_index(fruits, key=len, value=3))
None
```

In the example above, watermelon was chosen because its name is precisely ten characters long, while no fruits on the list have names made up of three letters.

That's great, but at the same time, you've just lost the ability to search by value. To remedy this, you could assign the key a default value of None and then check if it was given or not. However, in a more streamlined solution, you'd always want to call the key. By default, it would be an [identity function](#) returning the element itself:

Python

```
def identity(element):
    return element

def find_index(elements, value, key=identity):
    ...
```

Alternatively, you might define the identity function inline with an anonymous [lambda expression](#):

Python

```
def find_index(elements, value, key=lambda x: x):
    ...
```

`find_index()` answers only one question. There are still two others, which are “Is it there?” and “What is it?” To answer these two, you can build on top of it:

Python

```
def find_index(elements, value, key):
    ...

def contains(elements, value, key=identity):
    return find_index(elements, value, key) is not None

def find(elements, value, key=identity):
    index = find_index(elements, value, key)
    return None if index is None else elements[index]
```

With these three functions, you can tell almost everything about an element. However, you still haven't addressed **duplicates** in your implementation. What if you had a collection of people, and some of them shared a common name or surname? For example, there might be a Smith family or a few guys going by the name of John among the people:

Python

```
people = [
    Person('Bob', 'Williams'),
    Person('John', 'Doe'),
    Person('Paul', 'Brown'),
    Person('Alice', 'Smith'),
    Person('John', 'Smith'),
]
```

To model the `Person` type, you can modify a data class defined earlier:

Python

```
from dataclasses import dataclass

@dataclass(order=True)
class Person:
    name: str
    surname: str
```

Notice the use of the `order` attribute to enable automatic generation of magic methods for comparing instances of the class by all fields. Alternatively, you might prefer to take advantage of the [namedtuple](#), which has a shorter syntax:

Python

```
from collections import namedtuple
Person = namedtuple('Person', 'name surname')
```

Both definitions are fine and interchangeable. Each person has a `name` and a `surname` attribute. To sort and search by one of them, you can conveniently define the key function with an `attrgetter()` available in the built-in operator module:

Python

>>>

```
>>> from operator import attrgetter
>>> by_surname = attrgetter('surname')
>>> people.sort(key=by_surname)
>>> people
[Person(name='Paul', surname='Brown'),
 Person(name='John', surname='Doe'),
 Person(name='Alice', surname='Smith'),
 Person(name='John', surname='Smith'),
 Person(name='Bob', surname='Williams')]
```

Notice how people are now sorted by surname in ascending order. There's John Smith and Alice Smith, but binary searching for the Smith surname currently gives you only one arbitrary result:

Python

>>>

```
>>> find(people, key=by_surname, value='Smith')
Person(name='Alice', surname='Smith')
```

To mimic the features of the `bisect` module shown before, you can write your own version of `bisect_left()` and `bisect_right()`. Before finding the **leftmost** instance of a duplicate element, you want to determine if there's such an element at all:

Python

```
def find_leftmost_index(elements, value, key=identity):
    index = find_index(elements, value, key)
    if index is not None:
        ...
    return index
```

If some index has been found, then you can look to the left and keep moving until you come across an element with a different key or there are no more elements:

Python

```
def find_leftmost_index(elements, value, key=identity):
    index = find_index(elements, value, key)
    if index is not None:
        while index > -1 and key(elements[index]) == value:
```

```

        while index >= 0 and key(elements[index]) == value:
            index -= 1
        index += 1
    return index

```

Once you go past the leftmost element, you need to move the index back by one position to the right.

Finding the **rightmost** instance is quite similar, but you need to flip the conditions:

Python

```

def find_rightmost_index(elements, value, key=identity):
    index = find_index(elements, value, key)
    if index is not None:
        while index < len(elements) and key(elements[index]) == value:
            index += 1
        index -= 1
    return index

```

Instead of going left, now you're going to the right until the end of the list. Using both functions allows you to find **all occurrences** of duplicate items:

Python

```

def find_all_indices(elements, value, key=identity):
    left = find_leftmost_index(elements, value, key)
    right = find_rightmost_index(elements, value, key)
    if left and right:
        return set(range(left, right + 1))
    return set()

```

This function always returns a `set`. If the element isn't found, then the set will be empty. If the element is unique, then the set will be made up of only a single index. Otherwise, there will be multiple indices in the set.

To wrap up, you can define even more abstract functions to complete your binary search Python library:

Python

```

def find_leftmost(elements, value, key=identity):
    index = find_leftmost_index(elements, value, key)
    return None if index is None else elements[index]

def find_rightmost(elements, value, key=identity):
    index = find_rightmost_index(elements, value, key)
    return None if index is None else elements[index]

def find_all(elements, value, key=identity):
    return {elements[i] for i in find_all_indices(elements, value, key)}

```

Not only does this allow you to pinpoint the exact location of elements on the list, but also to retrieve those elements. You're able to ask very specific questions:

Is it there?	Where is it?	What is it?
<code>contains()</code>	<code>find_index()</code>	<code>find()</code>
	<code>find_leftmost_index()</code>	<code>find_leftmost()</code>
	<code>find_rightmost_index()</code>	<code>find_rightmost()</code>
	<code>find_all_indices()</code>	<code>find_all()</code>

The complete code of this binary search Python library can be found at the link below:

Get Sample Code: [Click here to get the sample code you'll use](#) to learn about binary search in Python in this

Recursively

For the sake of simplicity, you're only going to consider the **recursive** version of `contains()`, which tells you if an element was found.

Note: My favorite definition of [recursion](#) was given in an [episode](#) of the *Fun Fun Function* series about functional programming in JavaScript:

“Recursion is when a function calls itself until it doesn’t.”

— Mattias Petter Johansson

The most straightforward approach would be to take the iterative version of binary search and use the [slicing operator](#) to chop the list:

Python

```
def contains(elements, value):
    left, right = 0, len(elements) - 1

    if left <= right:
        middle = (left + right) // 2

        if elements[middle] == value:
            return True

        if elements[middle] < value:
            return contains(elements[middle + 1:], value)
        elif elements[middle] > value:
            return contains(elements[:middle], value)

    return False
```

Instead of looping, you check the condition once and sometimes call the same function on a smaller list. What could go wrong with that? Well, it turns out that slicing generates **copies** of element references, which can have noticeable memory and computational overhead.

To avoid copying, you might reuse the same list but pass different boundaries into the function whenever necessary:

Python

```
def contains(elements, value, left, right):
    if left <= right:
        middle = (left + right) // 2

        if elements[middle] == value:
            return True

        if elements[middle] < value:
            return contains(elements, value, middle + 1, right)
        elif elements[middle] > value:
            return contains(elements, value, left, middle - 1)
```

```
    return False
```

The downside is that every time you want to call that function, you have to pass initial boundaries, making sure they're correct:

Python

>>>

```
>>> sorted_fruits = ['apple', 'banana', 'orange', 'plum']
>>> contains(sorted_fruits, 'apple', 0, len(sorted_fruits) - 1)
True
```

If you were to make a mistake, then it would potentially not find that element. You can improve this by using default function arguments or by introducing a helper function that delegates to the recursive one:

Python

```
def contains(elements, value):
    return recursive(elements, value, 0, len(elements) - 1)

def recursive(elements, value, left, right):
    ...
```

Going further, you might prefer to [nest](#) one function in another to hide the technical details and to take advantage of variable reuse from outer scope:

Python

```
def contains(elements, value):
    def recursive(left, right):
        if left <= right:
            middle = (left + right) // 2
            if elements[middle] == value:
                return True
            if elements[middle] < value:
                return recursive(middle + 1, right)
            elif elements[middle] > value:
                return recursive(left, middle - 1)
        return False
    return recursive(0, len(elements) - 1)
```

The `recursive()` inner function can access both `elements` and `value` parameters even though they're defined in the enclosing scope. The life cycle and visibility of variables in Python is dictated by the so-called **LEGB** rule, which tells the interpreter to look for symbols in the following order:

1. **Local** scope
2. **Enclosing** scope
3. **Global** scope
4. **Built-in** symbols

This allows variables that are defined in outer scope to be accessed from within nested blocks of code.

The choice between an iterative and a recursive implementation is often the net result of performance considerations, convenience, as well as personal taste. However, there are also certain risks involved with recursion, which is one of the subjects of the next section.

Covering Tricky Details

Here's what the author of [*The Art of Computer Programming*](#) has to say about implementing the binary search algorithm:

“Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky, and many good programmers have done it wrong the first few times they tried.”

— Donald Knuth

If that doesn't deter you enough from the idea of writing the algorithm yourself, then maybe this will. The standard library in [Java](#) had a subtle [bug](#) in their implementation of binary search, which remained undiscovered for a decade! But the bug itself traces its roots much earlier than that.

Note: I once fell victim to the binary search algorithm during a technical screening. There were a couple of coding puzzles to solve, including a binary search one. Guess which one I failed to complete? Yeah.

The following list isn't exhaustive, but at the same time, it doesn't talk about common mistakes like forgetting to sort the list.

Integer Overflow

This is the Java bug that was just mentioned. If you recall, the binary search Python algorithm inspects the middle element of a bounded range in a sorted collection. But how is that middle element chosen exactly? Usually, you take the average of the lower and upper boundary to find the middle index:

Python

```
middle = (left + right) // 2
```

This method of calculating the average works just fine in the overwhelming majority of cases. However, once the collection of elements becomes sufficiently large, the sum of both boundaries won't fit the integer data type. It'll be larger than the maximum value allowed for integers.

Some programming languages might raise an error in such situations, which would immediately stop program execution. Unfortunately, that's not always the case. For example, Java silently ignores this problem, letting the value flip around and become some seemingly random number. You'll only know about the problem as long as the resulting number happens to be negative, which throws an `IndexOutOfBoundsException`.

Here's an example that demonstrates this behavior in [jshell](#), which is kind of like an interactive interpreter for Java:

Java

```
jshell> var a = Integer.MAX_VALUE
a ==> 2147483647

jshell> a + 1
$2 ==> -2147483648
```

A safer way to find the middle index could be calculating the offset first and then adding it to the lower boundary:

Python

```
middle = left + (right - left) // 2
```

Even if both values are maxed out, the sum in the formula above will never be. There are a few more ways, but the good news is that you don't need to worry about any of these, because Python is free from the integer overflow error. There's no upper limit on how big integers can be other than your memory:

Python

>>>

```
>>> 2147483647**7
210624582650556372047028295576838759252690170086892944262392971263
```

However, there's a catch. When you call functions from a library, that code might be subject to the C language constraints and still cause an overflow. There are plenty of libraries based on the C language in Python. You could even build your own [C extension module](#) or load a dynamically-linked library into Python using [ctypes](#).

Stack Overflow

The **stack overflow** problem may, theoretically, concern the recursive implementation of binary search. Most programming languages impose a limit on the number of nested function calls. Each call is associated with a return address stored on a stack. In Python, the default limit is a few thousand levels of such calls:

Python

>>>

```
>>> import sys
>>> sys.getrecursionlimit()
3000
```

This won't be enough for a lot of recursive functions. However, it's very unlikely that a binary search in Python would ever need more due to its logarithmic nature. You'd need a collection of two to the power of three thousand elements. That's a number with over *nine hundred* digits!

Nevertheless, it's still possible for the **infinite recursion error** to arise if the stopping condition is stated incorrectly due to a bug. In such a case, the infinite recursion will eventually cause a stack overflow.

Note: The **stack overflow error** is very common among languages with manual memory management. People would often google those errors to see if someone else already had similar issues, which gave the name to a popular [Q&A site](#) for programmers.

You can temporarily lift or decrease the recursion limit to simulate a stack overflow error. Note that the effective limit will be smaller because of the functions that the Python runtime environment has to call:

Python

>>>

```
>>> def countup(limit, n=1):
...     print(n)
...     if n < limit:
...         countup(limit, n + 1)
...
>>> import sys
>>> sys.setrecursionlimit(7) # Actual limit is 3
>>> countup(10)
1
2
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in countup
  File "<stdin>", line 4, in countup
  File "<stdin>", line 2, in countup
RecursionError: maximum recursion depth exceeded while calling a Python object
```

The recursive function was called three times before saturating the stack. The remaining four calls must have been made by the interactive interpreter. If you run that same code in [PyCharm](#) or an alternative Python shell, then you might get a different result.

Duplicate Elements

You're aware of the possibility of having duplicate elements in the list and you know how to deal with them. This is just to emphasize that a conventional binary search in Python might not produce deterministic results. Depending on how the list was sorted or how many elements it has, you'll get a different answer:

Python

>>>

```
>>> from search.binary import *
>>> sorted_fruits = ['apple', 'banana', 'banana', 'orange']
>>> find_index(sorted_fruits, 'banana')
1
>>> sorted_fruits.append('plum')
>>> find_index(sorted_fruits, 'banana')
2
```

There are two bananas on the list. At first, the call to `find_index()` returns the left one. However, adding a completely unrelated element at the end of the list makes the same call give you a different banana.

The same principle, known as **algorithm stability**, applies to [sorting algorithms](#). Some are stable, meaning they don't change the relative positions of equivalent elements. Others don't make such guarantees. If you ever need to sort elements by multiple criteria, then you should always start from the least significant key to retain stability.

Floating-Point Rounding

So far you've only searched for fruits or people, but what about numbers? They should be no different, right? Let's make a list of floating-point numbers at 0.1 increments using a [list comprehension](#):

Python

>>>

```
>>> sorted_numbers = [0.1*i for i in range(1, 4)]
```

The list should contain numbers the *one-tenth*, *two-tenths*, and *three-tenths*. Surprisingly, only two of those three numbers can be found:

Python

>>>

```
>>> from search.binary import contains  
>>> contains(sorted_numbers, 0.1)  
True  
>>> contains(sorted_numbers, 0.2)  
True  
>>> contains(sorted_numbers, 0.3)  
False
```

This isn't a problem strictly related to binary search in Python, as the built-in linear search is consistent with it:

Python

>>>

```
>>> 0.1 in sorted_numbers  
True  
>>> 0.2 in sorted_numbers  
True  
>>> 0.3 in sorted_numbers  
False
```

It's not even a problem related to Python but rather to how floating-point numbers are [represented](#) in computer memory. This is defined by the IEEE 754 standard for floating-point arithmetic. Without going into much detail, some decimal numbers don't have a finite representation in binary form. Because of limited memory, those numbers get [rounded](#), causing a **floating-point rounding error**.

Note: If you require maximum precision, then steer away from floating-point numbers. They're great for engineering purposes. However, for monetary operations, you don't want rounding errors to accumulate. It's recommended to scale down all prices and amounts to the smallest unit, such as cents or pennies, and treat them as integers.

Alternatively, many programming languages have support for **fixed-point numbers**, such as the [decimal](#) type in Python. This puts you in control of when and how rounding is taking place.

If you do need to work with floating-point numbers, then you should replace exact matching with an **approximate comparison**. Let's consider two variables with slightly different values:

Python

>>>

```
>>> a = 0.3  
>>> b = 0.1 * 3  
>>> b  
0.3000000000000004  
>>> a == b  
False
```

Regular comparison gives a negative result, although both values are nearly identical. Fortunately, Python comes with a function that will test if two values are close to each other within some small neighborhood:

Python

>>>

```
>>> import math  
>>> math.isclose(a, b)  
True
```

True

That neighborhood, which is the maximum distance between the values, can be adjusted if needed:

Python

>>>

```
>>> math.isclose(a, b, rel_tol=1e-16)
False
```

You can use that function to do a binary search in Python in the following way:

Python

```
import math

def find_index(elements, value):
    left, right = 0, len(elements) - 1

    while left <= right:
        middle = (left + right) // 2

        if math.isclose(elements[middle], value):
            return middle

        if elements[middle] < value:
            left = middle + 1
        elif elements[middle] > value:
            right = middle - 1
```

On the other hand, this implementation of binary search in Python is specific to floating-point numbers only. You couldn't use it to search for anything else without getting an error.

Analyzing the Time-Space Complexity of Binary Search

The following section will contain no code and some math concepts.

In computing, you can optimize the performance of pretty much any algorithm at the expense of increased memory use. For instance, you saw that a hash-based search of the IMDb dataset required an extra 0.5 GB of memory to achieve unparalleled speed.

Conversely, to save bandwidth, you'd compress a video stream before sending it over the network, increasing the amount of work to be done. This phenomenon is known as the **space-time tradeoff** and is useful in evaluating an algorithm's **complexity**.

Time-Space Complexity

The computational complexity is a *relative* measure of how many resources an algorithm needs to do its job. The resources include **computation time** as well as the amount of **memory** it uses. Comparing the complexity of various algorithms allows you to make an informed decision about which is better in a given situation.

Note: Algorithms that don't need to allocate more memory than their input data already consumes are called **in-place**, or **in-situ**, algorithms. This results in mutating the original data, which sometimes may have unwanted side-effects.

You looked at a few search algorithms and their average performance against a large dataset. It's clear from those measurements that a binary search is faster than a linear search. You can even tell by what factor.

However, if you took the same measurements in a different environment, you'd probably get slightly or perhaps entirely different results. There are invisible factors at play that can be influencing your test. Besides, such measurements aren't always feasible. So, how can you compare time complexities quickly and objectively?

The first step is to break down the algorithm into smaller pieces and find the one that is doing the most work. It's likely going to be some **elementary operation** that gets called a lot and consistently takes about the same time to run. For search algorithms, such an operation might be the comparison of two elements.

Having established that, you can now analyze the algorithm. To find the time complexity, you want to describe the **relationship** between the number of elementary operations executed versus the size of the input. Formally, such a relationship is a mathematical function. However, you're not interested in looking for its exact algebraic formula but rather **estimating** its overall shape.

There are a few well-known classes of functions that most algorithms fit in. Once you classify an algorithm according to one of them, you can put it on a scale:

Common Classes of Time Complexity

These classes tell you how the number of elementary operations increases with the growing size of the input. They are, from left to right:

- Constant
- Logarithmic
- Linear
- Quasilinear
- Quadratic
- Exponential
- Factorial

This can give you an idea about the performance of the algorithm you're considering. A constant complexity, regardless of the input size, is the most desired one. A logarithmic complexity is still pretty good, indicating a divide-and-conquer technique at use. The further to the right on this scale, the worse the complexity of the algorithm, because it has more work to do.

When you're talking about the time complexity, what you typically mean is the **asymptotic complexity**, which describes the behavior under very large data sets. This simplifies the function formula by eliminating all terms and coefficients but the one that grows at the fastest rate (for example, n squared).

However, a single function doesn't provide enough information to compare two algorithms accurately. The time complexity may vary depending on the volume of data. For example, the binary search algorithm is like a turbocharged engine, which builds pressure before it's ready to deliver power. On the other hand, the linear search algorithm is fast from the start but quickly reaches its peak power and ultimately loses the race:

In terms of speed, the binary search algorithm starts to overtake the linear search when there's a certain number of elements in the collection. For smaller collections, a linear search might be a better choice.

Note: Note that the same algorithm may have different **optimistic**, **pessimistic**, and **average** time complexity.

For example, in the best-case scenario, a linear search algorithm will find the element at the first index, after running just one comparison.

On the other end of the spectrum, it'll have to compare a reference value to all elements in the collection. In practice, you want to know the pessimistic complexity of an algorithm.

There are a few mathematical notations of the asymptotic complexity, which are used to compare algorithms. By far the most popular one is the **Big O notation**.

The Big O Notation

The **Big O notation** represents the worst-case scenario of asymptotic complexity. Although this might sound rather intimidating, you don't need to know the formal definition. Intuitively, it's a very rough measure of the rate of growth at the tail of the function that describes the complexity. You pronounce it as "*big oh*" of something:

That "something" is usually a function of data size or just the digit "one" that stands for a constant. For example, the linear search algorithm has a time complexity of $O(n)$, while a hash-based search has $O(1)$ complexity.

Note: When you say that some algorithm has complexity $O(f(n))$, where n is the size of the input data, then it means that the function $f(n)$ is an upper bound of the graph of that complexity. In other words, the actual complexity of that algorithm won't grow faster than $f(n)$ multiplied by some constant, when n approaches infinity.

In real-life, the Big O notation is used less formally as both an upper and a lower bound. This is useful for the classification and comparison of algorithms without having to worry about the exact function formulas.

The Complexity of Binary Search

You'll estimate the asymptotic time complexity of binary search by determining the number of comparisons in the worst-case scenario—when an element is missing—as a function of input size. You can approach this problem in three different ways:

1. Tabular
2. Graphical
3. Analytical

The **tabular** method is about collecting empirical data, putting it in a table, and trying to guess the formula by eyeballing sampled values:

Number of Elements	Number of Comparisons
0	0
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4

The number of comparisons grows as you increase the number of elements in the collection, but the rate of growth is slower than if it was a linear function. That's an indication of a good algorithm that can scale with data.

If that doesn't help you, you can try the **graphical** method, which visualizes the sampled data by drawing a graph:

The data points seem to overlay with a curve, but you don't have enough information to provide a conclusive answer. It could be a [polynomial](#), whose graph turns up and down for larger inputs.

Taking the **analytical** approach, you can choose some relationship and look for patterns. For example, you might study how the number of elements shrinks in each step of the algorithm:

Comparison	Number of Elements
-	n
1st	$n/2$
2nd	$n/4$
3rd	$n/8$
:	:
k-th	$n/2^k$

In the beginning, you start with the whole collection of n elements. After the first comparison, you're left with only half of them. Next, you have a quarter, and so on. The pattern that arises from this observation is that after k -th comparison, there are $n/2^k$ elements. Variable k is the expected number of elementary operations.

After all k comparisons, there will be no more elements left. However, when you take one step back, that is $k - 1$, there will be exactly one element left. This gives you a convenient equation:

Multiply both sides of the equation by the denominator, then take the logarithm base two of the result, and move the remaining constant to the right. You've just found the formula for the binary search complexity, which is on the order of $O(\log(n))$.

Conclusion

Now you know the binary search algorithm inside and out. You can flawlessly implement it yourself, or take advantage of the standard library in Python. Having tapped into the concept of time-space complexity, you're able to choose the best search algorithm for the given situation.

Now you can:

- Use the `bisect` module to do a binary search in Python
- Implement binary search in Python **recursively** and **iteratively**
- Recognize and fix **defects** in a binary search Python implementation
- Analyze the **time-space complexity** of the binary search algorithm
- Search even **faster** than binary search

With all this knowledge, you'll rock your [programming interview](#)! Whether the binary search algorithm is an optimal solution to a particular problem, you have the tools to figure it out on your own. You don't need a computer science degree to do so.

You can grab all of the code you've seen in this tutorial at the link below:

Get Sample Code: [Click here to get the sample code you'll use](#) to learn about binary search in Python in this tutorial.

About Bartosz Zaczyński

Bartosz is a bootcamp instructor, author, and polyglot programmer in love with Python. He helps his students get into software engineering by sharing over a decade of commercial experience in the IT industry.

[» More about Bartosz](#)

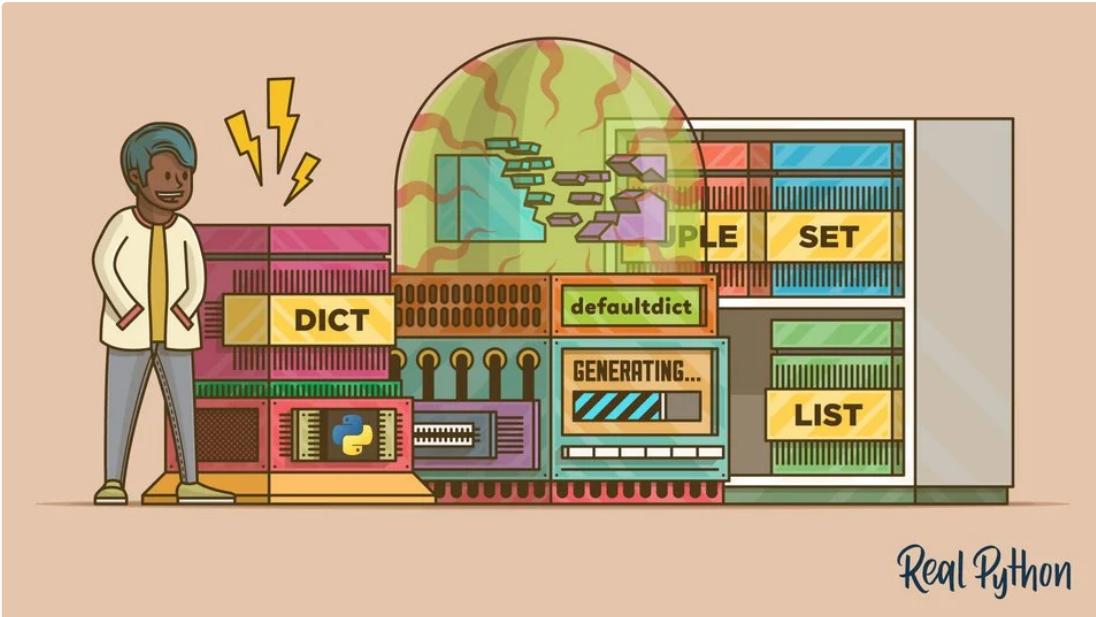
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Jon](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Real Python

Using the Python defaultdict Type for Handling Missing Keys

by Leodanis Pozo Ramos 🕒 Mar 11, 2020 💬 4 Comments 🔖 basics python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Handling Missing Keys in Dictionaries](#)
- [Understanding the Python defaultdict Type](#)
- [Using the Python defaultdict Type](#)
 - [Grouping Items](#)
 - [Grouping Unique Items](#)
 - [Counting Items](#)
 - [Accumulating Values](#)
- [Diving Deeper Into defaultdict](#)
 - [defaultdict vs dict](#)
 - [defaultdict.default_factory](#)
 - [defaultdict vs dict.setdefault\(\)](#)
 - [defaultdict.__missing__\(\)](#)
- [Emulating the Python defaultdict Type](#)
- [Passing Arguments to .default_factory](#)
 - [Using lambda](#)
 - [Using functools.partial\(\)](#)
- [Conclusion](#)



A common problem that you can face when working with Python [dictionaries](#) is to try to access or modify keys that don't exist in the dictionary. This will raise a [KeyError](#) and break up your code execution. To handle these kinds of situations, the [standard library](#) provides the Python **defaultdict** type, a dictionary-like class that's available for you in [collections](#).

The Python **defaultdict** type behaves almost exactly like a regular Python dictionary, but if you try to access or modify a missing key, then **defaultdict** will automatically create the key and generate a default value for it. This

makes defaultdict a valuable option for handling missing keys in dictionaries.

In this tutorial, you'll learn:

- How to use the Python defaultdict type for **handling missing keys** in a dictionary
- When and why to use a Python defaultdict rather than a regular [dict](#)
- How to use a defaultdict for **grouping, counting, and accumulating** operations

With this knowledge under your belt, you'll be in a better condition to effectively use the Python defaultdict type in your day-to-day programming challenges.

To get the most out of this tutorial, you should have some previous understanding of what Python [dictionaries](#) are and how to work with them. If you need to freshen up, then check out the following resources:

- [Dictionaries in Python](#) (Tutorial)
- [Dictionaries in Python](#) (Course)
- [How to Iterate Through a Dictionary in Python](#)

Free Bonus: [Click here to get a Python Cheat Sheet](#) and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Handling Missing Keys in Dictionaries

A common issue that you can face when working with Python dictionaries is **how to handle missing keys**. If your code is heavily based on dictionaries, or if you're creating dictionaries on the fly all the time, then you'll soon notice that dealing with frequent [KeyError exceptions](#) can be quite annoying and can add extra complexity to your code. With Python dictionaries, you have at least four available ways to handle missing keys:

1. Use `.setdefault()`
2. Use `.get()`
3. Use the `key in dict` idiom
4. Use a `try` and `except` block

The [Python docs](#) explain `.setdefault()` and `.get()` as follows:

`setdefault(key[, default])`

If `key` is in the dictionary, return its value. If not, insert `key` with a value of `default` and return `default`. `default` defaults to `None`.

`get(key[, default])`

Return the value for `key` if `key` is in the dictionary, else `default`. If `default` is not given, it defaults to `None`, so that this method never raises a [KeyError](#).

[\(Source\)](#)

Here's an example of how you can use `.setdefault()` to handle missing keys in a dictionary:

```
Python >>> a_dict = {}
>>> a_dict['missing_key']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
    a_dict['missing_key']
NameError: name 'a_dict' is not defined
```

```
KeyError: 'missing_key'
>>> a_dict.setdefault('missing_key', 'default value')
'default value'
>>> a_dict['missing_key']
'default value'
>>> a_dict.setdefault('missing_key', 'another default value')
'default value'
>>> a_dict
{'missing_key': 'default value'}
```

In the above code, you use `.setdefault()` to generate a default value for `missing_key`. Notice that your dictionary, `a_dict`, now has a new key called `missing_key` whose value is `'default value'`. This key didn't exist before you called `.setdefault()`. Finally, if you call `.setdefault()` on an existing key, then the call won't have any effect on the dictionary. Your key will hold the original value instead of the new default value.

Note: In the above code example, you get an exception, and Python shows you a **traceback** message, which tells you that you're trying to access a missing key in `a_dict`. If you want to dive deeper into how to decipher and understand a Python traceback, then check out [Understanding the Python Traceback](#) and [Getting the Most out of a Python Traceback](#).

On the other hand, if you use `.get()`, then you can code something like this:

```
Python >>>
>>> a_dict = {}
>>> a_dict.get('missing_key', 'default value')
'default value'
>>> a_dict
{}
```

Here, you use `.get()` to generate a default value for `missing_key`, but this time, your dictionary stays empty. This is because `.get()` returns the default value, but this value isn't added to the underlying dictionary. For example, if you have a dictionary called `D`, then you can assume that `.get()` works something like this:

```
Text
D.get(key, default) -> D[key] if key in D, else default
```

With this pseudo-code, you can understand how `.get()` works internally. If the key exists, then `.get()` returns the value mapped to that key. Otherwise, the default value is returned. Your code never creates or assigns a value to key. In this example, `default` defaults to [None](#).

You can also use [conditional statements](#) to handle missing keys in dictionaries. Take a look at the following example, which uses the `key in dict` idiom:

```
Python >>>
>>> a_dict = {}
>>> if 'key' in a_dict:
...     # Do something with 'key'...
...     a_dict['key']
... else:
...     a_dict['key'] = 'default value'
...
>>> a_dict
{'key': 'default value'}
```

In this code, you use an `if` statement along with the [in operator](#) to check if key is present in `a_dict`. If so, then you can perform any action with key or with its value. Otherwise, you create the new key, key, and assign it a 'default value'. Note that the above code works similar to `.setdefault()` but takes four lines of code, while `.setdefault()` would only take one line (in addition to being more readable).

You can also walk around the `KeyError` by using a `try` and `except` block to handle the exception. Consider the following piece of code:

Python

```
>>> a_dict = {}
>>> try:
...     # Do something with 'key'...
...     a_dict['key']
... except KeyError:
...     a_dict['key'] = 'default value'
...
>>> a_dict
{'key': 'default value'}
```

>>>

The `try` and `except` block in the above example catches the `KeyError` whenever you try to get access to a missing key. In the `except` clause, you create the key and assign it a 'default value'.

Note: If missing keys are uncommon in your code, then you might prefer to use a `try` and `except` block ([EAFP coding style](#)) to catch the `KeyError` exception. This is because the code doesn't check the existence of every key and only handles a few exceptions, if any.

On the other hand, if missing keys are quite common in your code, then the conditional statement ([LBYL coding style](#)) can be a better choice because checking for keys can be less costly than handling frequent exceptions.

So far, you've learned how to handle missing keys using the tools that `dict` and Python offer you. However, the examples you saw here are quite verbose and hard to read. They might not be as straightforward as you might want. That's why the [Python standard library](#) provides a more elegant, [Pythonic](#), and efficient solution. That solution is `collections.defaultdict`, and that's what you'll be covering from now on.

Understanding the Python defaultdict Type

The Python standard library provides [collections](#), which is a module that implements specialized container types. One of those is the Python `defaultdict` type, which is an alternative to `dict` that's specifically designed to help you out with missing keys. `defaultdict` is a Python type that inherits from `dict`:

Python

```
>>> from collections import defaultdict
>>> issubclass(defaultdict, dict)
True
```

>>>

The above code shows that the Python `defaultdict` type is a **subclass** of `dict`. This means that `defaultdict` inherits most of the behavior of `dict`. So, you can say that `defaultdict` is much like an ordinary dictionary. The main difference between `defaultdict` and `dict` is that when you try to access or modify a key that's not present in the dictionary, a default value is automatically given to that key. In order to provide this functionality, the Python `defaultdict` type does two things:

1. It overrides `__missing__()`.
2. It adds `.default_factory`, a writable instance variable that needs to be provided at the time of instantiation.

The instance variable `.default_factory` will hold the first argument passed into `defaultdict.__init__()`. This argument can take a valid Python callable or `None`. If a callable is provided, then it'll automatically be called by `defaultdict` whenever you try to access or modify the value associated with a missing key.

Note: All the remaining arguments to the class initializer are treated as if they were passed to the initializer of regular `dict`, including the keyword arguments.

Take a look at how you can create and properly initialize a defaultdict:

Python

>>>

```
>>> # Correct instantiation
>>> def_dict = defaultdict(list) # Pass list to .default_factory
>>> def_dict['one'] = 1 # Add a key-value pair
>>> def_dict['missing'] # Access a missing key returns an empty list
[]
>>> def_dict['another_missing'].append(4) # Modify a missing key
>>> def_dict
defaultdict(<class 'list'>, {'one': 1, 'missing': [], 'another_missing': [4]})
```

Here, you pass `list` to `.default_factory` when you create the dictionary. Then, you use `def_dict` just like a regular dictionary. Note that when you try to access or modify the value mapped to a non-existent key, the dictionary assigns it the default value that results from calling `list()`.

Keep in mind that you must pass a valid Python callable object to `.default_factory`, so remember not to call it using the parentheses at initialization time. This can be a common issue when you start using the Python `defaultdict` type. Take a look at the following code:

Python

>>>

```
>>> # Wrong instantiation
>>> def_dict = defaultdict(list())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    def_dict = defaultdict(list())
TypeError: first argument must be callable or None
```

Here, you try to create a `defaultdict` by passing `list()` to `.default_factory`. The call to `list()` raises a `TypeError`, which tells you that the first argument must be callable or `None`.

With this introduction to the Python `defaultdict` type, you can get start coding with practical examples. The next few sections will walk you through some common use cases where you can rely on a `defaultdict` to provide an elegant, efficient, and Pythonic solution.

Using the Python `defaultdict` Type

Sometimes, you'll use a mutable built-in collection (a `list`, `dict`, or `set`) as values in your Python dictionaries. In these cases, you'll need to **initialize the keys** before first use, or you'll get a `KeyError`. You can either do this process manually or automate it using a Python `defaultdict`. In this section, you'll learn how to use the Python `defaultdict` type for solving some common programming problems:

- **Grouping** the items in a collection
- **Counting** the items in a collection
- **Accumulating** the values in a collection

You'll be covering some examples that use `list`, `set`, `int`, and `float` to perform grouping, counting, and accumulating operations in a user-friendly and efficient way.

Grouping Items

A typical use of the Python `defaultdict` type is to set `.default_factory` to `list` and then build a dictionary that maps keys to lists of values. With this `defaultdict`, if you try to get access to any missing key, then the dictionary runs the following steps:

1. **Call** `list()` to create a new empty list
2. **Insert** the empty list into the dictionary using the missing key as key
3. **Return** a reference to that list

This allows you to write code like this:

Python

>>>

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
>>> dd['key'].append(1)
>>> dd
defaultdict(<class 'list'>, {'key': [1]})
>>> dd['key'].append(2)
>>> dd
defaultdict(<class 'list'>, {'key': [1, 2]})
>>> dd['key'].append(3)
>>> dd
defaultdict(<class 'list'>, {'key': [1, 2, 3]})
```

Here, you create a Python defaultdict called dd and pass list to .default_factory. Notice that even when key isn't defined, you can append values to it without getting a KeyError. That's because dd automatically calls .default_factory to generate a default value for the missing key.

You can use defaultdict along with list to group the items in a sequence or a collection. Suppose that you've retrieved the following data from your company's database:

Department	Employee Name
Sales	John Doe
Sales	Martin Smith
Accounting	Jane Doe
Marketing	Elizabeth Smith
Marketing	Adam Doe
...	...

With this data, you create an initial list of [tuple](#) objects like the following:

Python

```
dep = [('Sales', 'John Doe'),
        ('Sales', 'Martin Smith'),
        ('Accounting', 'Jane Doe'),
        ('Marketing', 'Elizabeth Smith'),
        ('Marketing', 'Adam Doe')]
```

Now, you need to create a dictionary that groups the employees by department. To do this, you can use a defaultdict as follows:

Python

```
from collections import defaultdict

dep_dd = defaultdict(list)
for department, employee in dep:
    dep_dd[department].append(employee)
```

Here, you create a defaultdict called dep_dd and use a [for loop](#) to iterate through your dep list. The statement dep_dd[department].append(employee) creates the keys for the departments, initializes them to an empty list, and then appends the employees to each department. Once you run this code, your dep_dd will look something like this:

Python

>>>

```
defaultdict(<class 'list'>, {'Sales': ['John Doe', 'Martin Smith'],
                             'Accounting': ['Jane Doe'],
                             'Marketing': ['Elizabeth Smith', 'Adam Doe']})
```

In this example, you group the employees by their department using a defaultdict with `.default_factory` set to `list`. To do this with a regular dictionary, you can use `dict.setdefault()` as follows:

Python

```
dep_d = dict()
for department, employee in dep:
    dep_d.setdefault(department, []).append(employee)
```

This code is straightforward, and you'll find similar code quite often in your work as a Python coder. However, the defaultdict version is arguably more readable, and for large datasets, it can also be a lot [faster and more efficient](#). So, if speed is a concern for you, then you should consider using a defaultdict instead of a standard dict.

Grouping Unique Items

Continue working with the data of departments and employees from the previous section. After some processing, you realize that a few employees have been **duplicated** in the database by mistake. You need to clean up the data and remove the duplicated employees from your `dep_dd` dictionary. To do this, you can use a set as the `.default_factory` and rewrite your code as follows:

Python

```
dep = [('Sales', 'John Doe'),
        ('Sales', 'Martin Smith'),
        ('Accounting', 'Jane Doe'),
        ('Marketing', 'Elizabeth Smith'),
        ('Marketing', 'Elizabeth Smith'),
        ('Marketing', 'Adam Doe'),
        ('Marketing', 'Adam Doe'),
        ('Marketing', 'Adam Doe'])

dep_dd = defaultdict(set)
for department, employee in dep:
    dep_dd[department].add(employee)
```

In this example, you set `.default_factory` to `set`. **Sets** are **collections of unique objects**, which means that you can't create a set with repeated items. This is a really interesting feature of sets, which guarantees that you won't have repeated items in your final dictionary.

Counting Items

If you set `.default_factory` to `int`, then your defaultdict will be useful for **counting the items** in a sequence or collection. When you call `int()` with no arguments, the function returns `0`, which is the typical value you'd use to initialize a counter.

To continue with the example of the company database, suppose you want to build a dictionary that counts the number of employees per department. In this case, you can code something like this:

Python

>>>

```
>>> from collections import defaultdict
>>> dep = [('Sales', 'John Doe'),
...           ('Sales', 'Martin Smith'),
...           ('Accounting', 'Jane Doe'),
...           ('Marketing', 'Elizabeth Smith'),
...           ('Marketing', 'Adam Doe')]
>>> dd = defaultdict(int)
>>> for department, _ in dep:
...     dd[department] += 1
>>> dd
defaultdict(<class 'int'>, {'Sales': 2, 'Accounting': 1, 'Marketing': 2})
```

Here, you set `.default_factory` to `int`. When you call `int()` with no argument, the returned value is `0`. You can use this default value to start counting the employees that work in each department. For this code to work correctly, you need a clean dataset. There must be no repeated data. Otherwise, you'll need to filter out the repeated employees.

Another example of counting items is the `mississippi` example, where you count the number of times each letter in a word is repeated. Take a look at the following code:

```
Python >>>
>>> from collections import defaultdict
>>> s = 'mississippi'
>>> dd = defaultdict(int)
>>> for letter in s:
...     dd[letter] += 1
...
>>> dd
defaultdict(<class 'int'>, {'m': 1, 'i': 4, 's': 4, 'p': 2})
```

In the above code, you create a `defaultdict` with `.default_factory` set to `int`. This sets the default value for any given key to `0`. Then, you use a `for` loop to traverse the `string` `s` and use an [augmented assignment operation](#) to add `1` to the counter in every iteration. The keys of `dd` will be the letters in `mississippi`.

Note: Python's [augmented assignment operators](#) are a handy shortcut to common operations.

Take a look at the following examples:

- `var += 1` is equivalent to `var = var + 1`
- `var -= 1` is equivalent to `var = var - 1`
- `var *= 1` is equivalent to `var = var * 1`

This is just a sample of how the augmented assignment operators work. You can take a look at the [official documentation](#) to learn more about this feature.

As counting is a relatively common task in programming, the Python dictionary-like class [`collections.Counter`](#) is specially designed for counting items in a sequence. With `Counter`, you can write the `mississippi` example as follows:

```
Python >>>
>>> from collections import Counter
>>> counter = Counter('mississippi')
>>> counter
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

In this case, `Counter` does all the work for you! You only need to pass in a sequence, and the dictionary will count its items, storing them as keys and the counts as values. Note that this example works because Python strings are also a sequence type.

Accumulating Values

Sometimes you'll need to calculate the **total sum** of the values in a sequence or collection. Let's say you have the following [Excel sheet](#) with data about the sales of your Python website:

Products	July	August	September
Books	1250.00	1300.00	1420.00
Tutorials	560.00	630.00	750.00
Courses	2500.00	2430.00	2750.00

Next, you process the data using Python and get the following list of tuple objects:

Python

```
incomes = [('Books', 1250.00),
           ('Books', 1300.00),
           ('Books', 1420.00),
           ('Tutorials', 560.00),
           ('Tutorials', 630.00),
           ('Tutorials', 750.00),
           ('Courses', 2500.00),
           ('Courses', 2430.00),
           ('Courses', 2750.00),]
```

With this data, you want to calculate the total income per product. To do that, you can use a Python `defaultdict` with `float` as `.default_factory` and then code something like this:

Python

```
1 from collections import defaultdict
2
3 dd = defaultdict(float)
4 for product, income in incomes:
5     dd[product] += income
6
7 for product, income in dd.items():
8     print(f'Total income for {product}: ${income:.2f}')
```

Here's what this code does:

- In **line 1**, you import the Python `defaultdict` type.
- In **line 3**, you create a `defaultdict` object with `.default_factory` set to `float`.
- In **line 4**, you define a `for` loop to iterate through the items of `incomes`.
- In **line 5**, you use an augmented assignment operation (`+=`) to accumulate the incomes per product in the dictionary.

The second loop iterates through the items of `dd` and prints the incomes to your screen.

Note: If you want to dive deeper into dictionary iteration, check out [How to Iterate Through a Dictionary in Python](#).

If you put all this code into a file called `incomes.py` and run it from your command line, then you'll get the following output:

Shell

```
$ python3 incomes.py
Total income for Books: $3,970.00
Total income for Tutorials: $1,940.00
Total income for Courses: $7,680.00
```

You now have a summary of incomes per product, so you can make decisions on which strategy to follow for increasing the total income of your site.

Diving Deeper Into `defaultdict`

So far, you've learned how to use the Python `defaultdict` type by coding some practical examples. At this point, you can dive deeper into **type implementation** and other working details. That's what you'll be covering in the next few sections.

`defaultdict` vs `dict`

For you to better understand the Python `defaultdict` type, a good exercise would be to compare it with its superclass, `dict`. If you want to know the methods and attributes that are specific to the Python `defaultdict` type,

then you can run the following line of code:

```
Python >>>
>>> set(dir(defaulddict)) - set(dir(dict))
{'__copy__', 'default_factory', '__missing__'}
```

In the above code, you use `dir()` to get the list of valid attributes for `dict` and `defaulddict`. Then, you use a `set` difference to get the set of methods and attributes that you can only find in `defaulddict`. As you can see, the differences between these two classes are. You have two methods and one instance attribute. The following table shows what the methods and the attribute are for:

Method or Attribute	Description
<code>.__copy__()</code>	Provides support for <code>copy.copy()</code>
<code>.default_factory</code>	Holds the callable invoked by <code>.__missing__()</code> to automatically provide default values for missing keys
<code>.__missing__(key)</code>	Gets called when <code>.__getitem__()</code> can't find key

In the above table, you can see the methods and the attribute that make a `defaulddict` different from a regular `dict`. The rest of the methods are the same in both classes.

Note: If you initialize a `defaulddict` using a valid callable, then you won't get a `KeyError` when you try to get access to a missing key. Any key that doesn't exist gets the value returned by `.default_factory`.

Additionally, you might notice that a `defaulddict` is equal to a `dict` with the same items:

```
Python >>>
>>> std_dict = dict(numbers=[1, 2, 3], letters=['a', 'b', 'c'])
>>> std_dict
{'numbers': [1, 2, 3], 'letters': ['a', 'b', 'c']}
>>> def_dict = defaulddict(list, numbers=[1, 2, 3], letters=['a', 'b', 'c'])
>>> def_dict
defaulddict(<class 'list'>, {'numbers': [1, 2, 3], 'letters': ['a', 'b', 'c']})
>>> std_dict == def_dict
True
```

Here, you create a regular dictionary `std_dict` with some arbitrary items. Then, you create a `defaulddict` with the same items. If you test both dictionaries for content equality, then you'll see that they're equal.

defaulddict.default_factory

The first argument to the Python `defaulddict` type must be a **callable** that takes no arguments and returns a value. This argument is assigned to the instance attribute, `.default_factory`. For this, you can use any callable, including functions, methods, classes, type objects, or any other valid callable. The default value of `.default_factory` is `None`.

If you instantiate `defaulddict` without passing a value to `.default_factory`, then the dictionary will behave like a regular `dict` and the usual `KeyError` will be raised for missing key lookup or modification attempts:

```
Python >>>
>>> from collections import defaulddict
>>> dd = defaulddict()
>>> dd['missing_key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    dd['missing_key']
KeyError: 'missing_key'
```

Here, you instantiate the Python `defaulddict` type with no arguments. In this case, the instance behaves like a

standard dictionary. So, if you try to access or modify a missing key, then you'll get the usual `KeyError`. From this point on, you can use `dd` as a normal Python dictionary and, unless you assign a new callable to `.default_factory`, you won't be able to use the ability of `defaultdict` to handle missing keys automatically.

If you pass `None` to the first argument of `defaultdict`, then the instance will behave the same way you saw in the above example. That's because `.default_factory` defaults to `None`, so both initializations are equivalent. On the other hand, if you pass a valid callable object to `.default_factory`, then you can use it to handle missing keys in a user-friendly way. Here's an example where you pass `list` to `.default_factory`:

```
Python >>>
>>> dd = defaultdict(list, letters=['a', 'b', 'c'])
>>> dd.default_factory
<class 'list'>
>>> dd
defaultdict(<class 'list'>, {'letters': ['a', 'b', 'c']})
>>> dd['numbers']
[]
>>> dd
defaultdict(<class 'list'>, {'letters': ['a', 'b', 'c'], 'numbers': []})
>>> dd['numbers'].append(1)
>>> dd
defaultdict(<class 'list'>, {'letters': ['a', 'b', 'c'], 'numbers': [1]})
>>> dd['numbers'] += [2, 3]
>>> dd
defaultdict(<class 'list'>, {'letters': ['a', 'b', 'c'], 'numbers': [1, 2, 3]})
```

In this example, you create a Python `defaultdict` called `dd`, then you use `list` for its first argument. The second argument is called `letters` and holds a list of letters. You see that `.default_factory` now holds a `list` object that will be called when you need to supply a default value for any missing key.

Notice that when you try to access `numbers`, `dd` tests if `numbers` is in the dictionary. If it's not, then it calls `.default_factory()`. Since `.default_factory` holds a `list` object, the returned value is an empty list `[]`.

Now that `dd['numbers']` is initialized with an empty `list`, you can use `.append()` to add elements to the `list`. You can also use an augmented assignment operator `(+=)` to concatenate the lists `[1]` and `[2, 3]`. This way, you can handle missing keys in a more Pythonic and more efficient way.

On the other hand, if you pass a **non-callable** object to the initializer of the Python `defaultdict` type, then you'll get a `TypeError` like in the following code:

```
Python >>>
>>> defaultdict(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    defaultdict(0)
TypeError: first argument must be callable or None
```

Here, you pass `0` to `.default_factory`. Since `0` is not a callable object, you get a `TypeError` telling you that the first argument must be callable or `None`. Otherwise, `defaultdict` doesn't work.

Keep in mind that `.default_factory` is only called from `__getitem__()`, and not from other methods. This means that if `dd` is a `defaultdict` and `key` is a missing key, then `dd[key]` will call `.default_factory` to provide a default value, but `dd.get(key)` still returns `None` instead of the value that `.default_factory` would provide. That's because `.get()` doesn't call `__getitem__()` to retrieve the key.

Take a look at the following code:

```
Python >>>
>>> dd = defaultdict(list)
>>> # Calls dd.__getitem__('missing')
>>> dd['missing']
[]
>>> # Don't call dd.__getitem__('another_missing')
>>> print(dd.get('another_missing'))
None
>>> dd
```

```
defaultdict(<class 'list'>, {'missing': []})
```

In this code fragment, you can see that `dd.get()` returns `None` rather than the default value that `.default_factory` would provide. That's because `.default_factory` is only called from `__missing__()`, which is not called by `.get()`.

Notice that you can also add **arbitrary values** to a Python `defaultdict`. This means that you're not limited to values with the same type as the values generated by `.default_factory`. Here's an example:

Python

>>>

```
>>> dd = defaultdict(list)
>>> dd
defaultdict(<class 'list'>, {})
>>> dd['string'] = 'some string'
>>> dd
defaultdict(<class 'list'>, {'string': 'some string'})
>>> dd['list']
[]
>>> dd
defaultdict(<class 'list'>, {'string': 'some string', 'list': []})
```

Here, you create a `defaultdict` and pass in a `list` object to `.default_factory`. This sets your default values to be empty lists. However, you can freely add a new key that holds values of a different type. That's the case with the key `string`, which holds a `str` object instead of a `list` object.

Finally, you can always **change or update the callable** you initially assign to `.default_factory` in the same way you would do with any instance attribute:

Python

>>>

```
>>> dd.default_factory = str
>>> dd['missing_key']
''
```

In the above code, you change `.default_factory` from `list` to `str`. Now, whenever you try to get access to a missing key, your default value will be an empty string ('').

Depending on your use cases for the Python `defaultdict` type, you might need to freeze the dictionary once you finish creating it and make it read-only. To do this, you can set `.default_factory` to `None` after you finish populating the dictionary. This way, your dictionary will behave like a standard `dict`, which means you won't have more automatically generated default values.

defaultdict vs dict.setdefault()

As you saw before, `dict` provides `.setdefault()`, which will allow you to assign values to missing keys on the fly. In contrast, with a `defaultdict` you can specify the default value up front when you initialize the container. You can use `.setdefault()` to assign default values as follows:

Python

>>>

```
>>> d = dict()
>>> d.setdefault('missing_key', [])
[]
>>> d
{'missing_key': []}
```

In this code, you create a regular dictionary and then use `.setdefault()` to assign a value (`[]`) to the key `missing_key`, which wasn't defined yet.

Note: You can assign any type of Python object using `.setdefault()`. This is an important difference compared to `defaultdict` if you consider that `defaultdict` only accepts a callable or `None`.

On the other hand, if you use a `defaultdict` to accomplish the same task, then the default value is generated on demand whenever you try to access or modify a missing key. Notice that, with `defaultdict`, the default value is

generated by the callable you pass upfront to the initializer of the class. Here's how it works:

Python

>>>

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
>>> dd['missing_key']
[]
>>> dd
defaultdict(<class 'list'>, {'missing_key': []})
```

Here, you first import the Python defaultdict type from collections. Then, you create a defaultdict and pass list to .default_factory. When you try to get access to a missing key, defaultdict internally calls .default_factory(), which holds a reference to list, and assigns the resulting value (an empty list) to missing_key.

The code in the above two examples does the same work, but the defaultdict version is arguably more readable, user-friendly, Pythonic, and straightforward.

Note: A call to a built-in type like list, set, dict, str, int, or float will return an empty object or zero for numeric types.

Take a look at the following code examples:

Python

>>>

```
>>> list()
[]
>>> set()
set([])
>>> dict()
{}
>>> str()
''
>>> float()
0.0
>>> int()
0
```

In this code, you call some built-in types with no arguments and get an empty object or zero for the numeric types.

Finally, using a defaultdict to handle missing keys can be faster than using dict.setdefault(). Take a look at the following example:

Python

```
# Filename: exec_time.py

from collections import defaultdict
from timeit import timeit

animals = [('cat', 1), ('rabbit', 2), ('cat', 3), ('dog', 4), ('dog', 1)]

std_dict = dict()
def_dict = defaultdict(list)

def group_with_dict():
    for animal, count in animals:
        std_dict.setdefault(animal, []).append(count)
    return std_dict

def group_with_defaultdict():
    for animal, count in animals:
        def_dict[animal].append(count)
    return def_dict

print(f'dict.setdefault() takes {timeit(group_with_dict)} seconds.')
print(f'defaultdict takes {timeit(group_with_defaultdict)} seconds.')
```

If you [run the script](#) from your system's command line, then you'll get something like this:

Shell

```
$ python3 exec_time.py
dict.setdefault() takes 1.0281260240008123 seconds.
defaultdict takes 0.6704721650003194 seconds.
```

Here, you use `timeit.timeit()` to measure the execution time of `group_with_dict()` and `group_with defaultdict()`. These functions perform equivalent actions, but the first uses `dict.setdefault()`, and the second uses a `defaultdict`. The time measure will depend on your current hardware, but you can see here that `defaultdict` is faster than `dict.setdefault()`. This difference can become more important as the dataset gets larger.

Additionally, you need to consider that creating a regular `dict` can be faster than creating a `defaultdict`. Take a look at this code:

Python

>>>

```
>>> from timeit import timeit
>>> from collections import defaultdict
>>> print(f'dict() takes {timeit(dict)} seconds.')
dict() takes 0.08921320698573254 seconds.
>>> print(f'defaultdict() takes {timeit(defaultdict)} seconds.')
defaultdict() takes 0.14101867799763568 seconds.
```

This time, you use `timeit.timeit()` to measure the execution time of `dict` and `defaultdict` instantiation. Notice that creating a `dict` takes almost half the time of creating a `defaultdict`. This might not be a problem if you consider that, in real-world code, you normally instantiate `defaultdict` only once.

Also notice that, by default, `timeit.timeit()` will run your code a million times. That's the reason for defining `std_dict` and `def_dict` out of the scope of `group_with_dict()` and `group_with defaultdict()` in `exec_time.py`. Otherwise, the time measure will be affected by the instantiation time of `dict` and `defaultdict`.

At this point, you may have an idea of when to use a `defaultdict` rather than a regular `dict`. Here are three things to take into account:

1. **If your code is heavily base on dictionaries** and you're dealing with missing keys all the time, then you should consider using a `defaultdict` rather than a regular `dict`.
2. **If your dictionary items need to be initialized** with a constant default value, then you should consider using a `defaultdict` instead of a `dict`.
3. **If your code relies on dictionaries** for aggregating, accumulating, counting, or grouping values, and performance is a concern, then you should consider using a `defaultdict`.

You can consider the above guidelines when deciding whether to use a `dict` or a `defaultdict`.

defaultdict.__missing__()

Behind the scenes, the Python `defaultdict` type works by calling `.default_factory` to supply default values to missing keys. The mechanism that makes this possible is `.__missing__()`, a special method supported by all the standard mapping types, including `dict` and `defaultdict`.

Note: Note that `.__missing__()` is automatically called by `.__getitem__()` to handle missing keys and that `.__getitem__()` is automatically called by Python at the same time for subscription operations like `d[key]`.

So, how does `.__missing__()` work? If you set `.default_factory` to `None`, then `.__missing__()` raises a `KeyError` with the key as an argument. Otherwise, `.default_factory` is called without arguments to provide a default value for the given key. This value is inserted into the dictionary and finally returned. If calling `.default_factory` raises an exception, then the exception is propagated unchanged.

The following code shows a viable Python implementation for `.__missing__()`:

Python

```

1 | def __missing__(self, key):
2 |     if self.default_factory is None:
3 |         raise KeyError(key)
4 |     if key not in self:
5 |         self[key] = self.default_factory()
6 |     return self[key]

```

Here's what this code does:

- In **line 1**, you define the method and its signature.
- In **lines 2 and 3**, you test to see if `.default_factory` is `None`. If so, then you raise a `KeyError` with the key as an argument.
- In **lines 4 and 5**, you check if the key is not in the dictionary. If it's not, then you call `.default_factory` and assign its return value to the key.
- In **line 6**, you return the key as expected.

Keep in mind that the presence of `.__missing__()` in a mapping has no effect on the behavior of other methods that look up keys, such as `.get()` or `.__contains__()`, which implements the `in` operator. That's because `.__missing__()` is only called by `.__getitem__()` when the requested key is not found in the dictionary. Whatever `.__missing__()` returns or raises is then returned or raised by `.__getitem__()`.

Now that you've covered an alternative Python implementation for `.__missing__()`, it would be a good exercise to try to emulate `defaultdict` with some Python code. That's what you'll be doing in the next section.

Emulating the Python `defaultdict` Type

In this section, you'll be coding a Python class that will behave much like a `defaultdict`. To do that, you'll subclass `collections.UserDict` and then add `.__missing__()`. Also, you need to add an instance attribute called `.default_factory`, which will hold the callable for generating default values on demand. Here's a piece of code that emulates most of the behavior of the Python `defaultdict` type:

Python

```

1 | import collections
2 |
3 | class my defaultdict(collections.UserDict):
4 |     def __init__(self, default_factory=None, *args, **kwargs):
5 |         super().__init__(*args, **kwargs)
6 |         if not callable(default_factory) and default_factory is not None:
7 |             raise TypeError('first argument must be callable or None')
8 |         self.default_factory = default_factory
9 |
10    def __missing__(self, key):
11        if self.default_factory is None:
12            raise KeyError(key)
13        if key not in self:
14            self[key] = self.default_factory()
15        return self[key]

```

Here's how this code works:

- In **line 1**, you import `collections` to get access to `UserDict`.
- In **line 3**, you create a class that subclasses `UserDict`.
- In **line 4**, you define the class initializer `.__init__()`. This method takes an argument called `default_factory` to hold the callable that you'll use to generate the default values. Notice that `default_factory` defaults to `None`, just like in a `defaultdict`. You also need the `*args` and `**kwargs` for emulating the normal behavior of a regular `dict`.
- In **line 5**, you call the superclass `.__init__()`. This means that you're calling `UserDict.__init__()` and passing `*args` and `**kwargs` to it.
- In **line 6**, you first check if `default_factory` is a valid callable object. In this case, you use `callable\(object\)`, which is a built-in function that returns `True` if `object` appears to be a callable and otherwise returns `False`. This check ensures that you can call `default_factory()` if you need to generate a default value for any

This check ensures that you can call `.default_factory()` if you need to generate a default value for any missing key. Then, you check if `.default_factory` is not `None`.

- In line 7, you raise a `TypeError` just like a regular dict would do if `default_factory` is `None`.
- In line 8, you initialize `.default_factory`.
- In line 10, you define `__missing__()`, which is implemented as you saw before. Recall that `__missing__()` is automatically called by `__getitem__()` when a given key is not in a dictionary.

If you feel in the mood to read some C code, then you can take a look at the [full code](#) for the Python `defaultdict`. Type in the [CPython source code](#).

Now that you've finished coding this class, you can test it by putting the code into a Python script called `my_dd.py` and importing it from an interactive session. Here's an example:

```
Python >>>
>>> from my_dd import my defaultdict
>>> dd_one = my defaultdict(list)
>>> dd_one
{}
>>> dd_one['missing']
[]
>>> dd_one
{'missing': []}
>>> dd_one.default_factory = int
>>> dd_one['another_missing']
0
>>> dd_one
{'missing': [], 'another_missing': 0}
>>> dd_two = my defaultdict(None)
>>> dd_two['missing']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    dd_two['missing']
  File "/home/user/my_dd.py", line 10,
    in __missing__
      raise KeyError(key)
KeyError: 'missing'
```

Here, you first import `my defaultdict` from `my_dd`. Then, you create an instance of `my defaultdict` and pass `list` to `.default_factory`. If you try to get access to a key with a subscription operation, like `dd_one['missing']`, then `__getitem__()` is automatically called by Python. If the key is not in the dictionary, then `__missing__()` is called, which generates a default value by calling `.default_factory()`.

You can also change the callable assigned to `.default_factory` using a normal assignment operation like in `dd_one.default_factory = int`. Finally, if you pass `None` to `.default_factory`, then you'll get a `KeyError` when trying to retrieve a missing key.

Note: The behavior of a `defaultdict` is essentially the same as this Python equivalent. However, you'll soon note that your Python implementation doesn't print as a real `defaultdict` but as a standard `dict`. You can modify this detail by overriding `__str__()` and `__repr__()`.

You may be wondering why you subclass `collections.UserDict` instead of a regular `dict` for this example. The main reason for this is that subclassing built-in types can be error-prone because the C code of the built-ins doesn't seem to consistently call special methods overridden by the user.

Here's an example that shows some issues that you can face when subclassing `dict`:

Python

>>>

```
>>> class MyDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, None)
...
>>> my_dict = MyDict(first=1)
>>> my_dict
{'first': 1}
>>> my_dict['second'] = 2
>>> my_dict
{'first': 1, 'second': None}
>>> my_dict.setdefault('third', 3)
3
>>> my_dict
{'first': 1, 'second': None, 'third': 3}
```

In this example, you create `MyDict`, which is a class that subclasses `dict`. Your implementation of `__setitem__()` always sets values to `None`. If you create an instance of `MyDict` and pass a keyword argument to its initializer, then you'll notice the class is not calling your `__setitem__()` to handle the assignment. You know that because the key `first` wasn't assigned `None`.

By contrast, if you run a subscription operation like `my_dict['second'] = 2`, then you'll notice that `second` is set to `None` rather than to `2`. So, this time you can say that subscription operations call your custom `__setitem__()`. Finally, notice that `.setdefault()` doesn't call `__setitem__()` either, because your `third` key ends up with a value of `3`.

`UserDict` doesn't inherit from `dict` but simulates the behavior of a standard dictionary. The class has an internal `dict` instance called `.data`, which is used to store the content of the dictionary. `UserDict` is a more reliable class when it comes to creating **custom mappings**. If you use `UserDict`, then you'll be avoiding the issues you saw before. To prove this, go back to the code for `my defaultdict` and add the following method:

Python

```
1 class my defaultdict(collections.UserDict):
2     # Snip
3     def __setitem__(self, key, value):
4         print('__setitem__() gets called')
5         super().__setitem__(key, None)
```

Here, you add a custom `__setitem__()` that calls the superclass `__setitem__()`, which always sets the value to `None`. Update this code in your script `my_dd.py` and import it from an interactive session as follows:

Python

>>>

```
>>> from my_dd import my defaultdict
>>> my_dict = my defaultdict(list, first=1)
__setitem__() gets called
>>> my_dict
{'first': None}
>>> my_dict['second'] = 2
__setitem__() gets called
>>> my_dict
{'first': None, 'second': None}
```

In this case, when you instantiate `my defaultdict` and pass `first` to the class initializer, your custom `__setitem__()` gets called. Also, when you assign a value to the key `second`, `__setitem__()` gets called as well. You now have a `my defaultdict` that consistently calls your custom special methods. Notice that all the values in the dictionary are equal to `None` now.

Passing Arguments to `.default_factory`

As you saw earlier, `.default_factory` must be set to a callable object that takes no argument and returns a value.

This value will be used to supply a default value for any missing key in the dictionary. Even when `.default_factory` shouldn't take arguments, Python offers some tricks that you can use if you need to supply arguments to it. In this section, you'll cover two Python tools that can serve this purpose:

1. [lambda](#)
2. [functools.partial\(\)](#)

With these two tools, you can add extra flexibility to the Python `defaultdict` type. For example, you can initialize a `defaultdict` with a callable that takes an argument and, after some processing, you can update the callable with a new argument to change the default value for the keys you'll create from this point on.

Using lambda

A flexible way to pass arguments to `.default_factory` is to use `lambda`. Suppose you want to create a function to generate default values in a `defaultdict`. The function does some processing and returns a value, but you need to pass an argument for the function to work correctly. Here's an example:

```
Python >>>
>>> def factory(arg):
...     # Do some processing here...
...     result = arg.upper()
...     return result
...
>>> def_dict = defaultdict(lambda: factory('default value'))
>>> def_dict['missing']
'DEFAULT VALUE'
```

In the above code, you create a function called `factory()`. The function takes an argument, does some processing, and returns the final result. Then, you create a `defaultdict` and use `lambda` to pass the string 'default value' to `factory()`. When you try to get access to a missing key, the following steps are run:

1. **The dictionary `def_dict` calls its `.default_factory`, which holds a reference to a `lambda` function.**
2. **The `lambda` function gets called and returns the value that results from calling `factory()` with 'default value' as an argument.**

If you're working with `def_dict` and suddenly need to change the argument to `factory()`, then you can do something like this:

```
Python >>>
>>> def_dict.default_factory = lambda: factory('another default value')
>>> def_dict['another_missing']
'ANOTHER DEFAULT VALUE'
```

This time, `factory()` takes a new string argument ('another default value'). From now on, if you try to access or modify a missing key, then you'll get a new default value, which is the string 'ANOTHER DEFAULT VALUE'.

Finally, you can possibly face a situation where you need a default value that's different from `0` or `[]`. In this case, you can also use `lambda` to **generate a different default value**. For example, suppose you have a list of integer numbers, and you need to calculate the cumulative product of each number. Then, you can use a `defaultdict` along with `lambda` as follows:

```
Python >>>
>>> from collections import defaultdict
>>> lst = [1, 1, 2, 1, 2, 2, 3, 4, 3, 3, 4, 4]
>>> def_dict = defaultdict(lambda: 1)
~~~ for number in lst:
```

```
>>> for number in lst:
...     def_dict[number] *= number
...
>>> def_dict
defaultdict(<function <lambda> at 0x...70>, {1: 1, 2: 8, 3: 27, 4: 64})
```

Here, you use `lambda` to supply a default value of `1`. With this initial value, you can calculate the cumulative product of each number in `lst`. Notice that you can't get the same result using `int` because the default value returned by `int` is always `0`, which is not a good initial value for the multiplication operations you need to perform here.

Using `functools.partial()`

`functools.partial(func, *args, **keywords)` is a function that returns a `partial` object. When you call this object with the positional arguments (`args`) and keyword arguments (`keywords`), it behaves similar to when you call `func(*args, **keywords)`. You can take advantage of this behavior of `partial()` and use it to pass arguments to `.default_factory` in a Python `defaultdict`. Here's an example:

```
Python >>>
>>> def factory(arg):
...     # Do some processing here...
...     result = arg.upper()
...     return result
...
>>> from functools import partial
>>> def_dict = defaultdict(partial(factory, 'default value'))
>>> def_dict['missing']
'DEFAULT VALUE'
>>> def_dict.default_factory = partial(factory, 'another default value')
>>> def_dict['another_missing']
'ANOTHER DEFAULT VALUE'
```

Here, you create a Python `defaultdict` and use `partial()` to supply an argument to `.default_factory`. Notice that you can also update `.default_factory` to use another argument for the callable `factory()`. This kind of behavior can add a lot of flexibility to your `defaultdict` objects.

Conclusion

The Python `defaultdict` type is a dictionary-like data structure provided by the Python standard library in a module called `collections`. The class inherits from `dict`, and its main added functionality is to supply default values for missing keys. In this tutorial, you've learned how to use the Python `defaultdict` type for handling the missing keys in a dictionary.

You're now able to:

- **Create and use** a Python `defaultdict` to handle missing keys
- **Solve** real-world problems related to grouping, counting, and accumulating operations
- **Know** the implementation differences between `defaultdict` and `dict`
- **Decide** when and why to use a Python `defaultdict` rather than a standard `dict`

The Python `defaultdict` type is a convenient and efficient data structure that's designed to help you out when you're dealing with missing keys in a dictionary. Give it a try and make your code faster, more readable, and more Pythonic!

About Leodanis Pozo Ramos

Leodanis is an industrial engineer who loves Python and software development. He is a self-taught Python programmer with 5+ years of experience building desktop applications.

[» More about Leodanis](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Bryan](#)

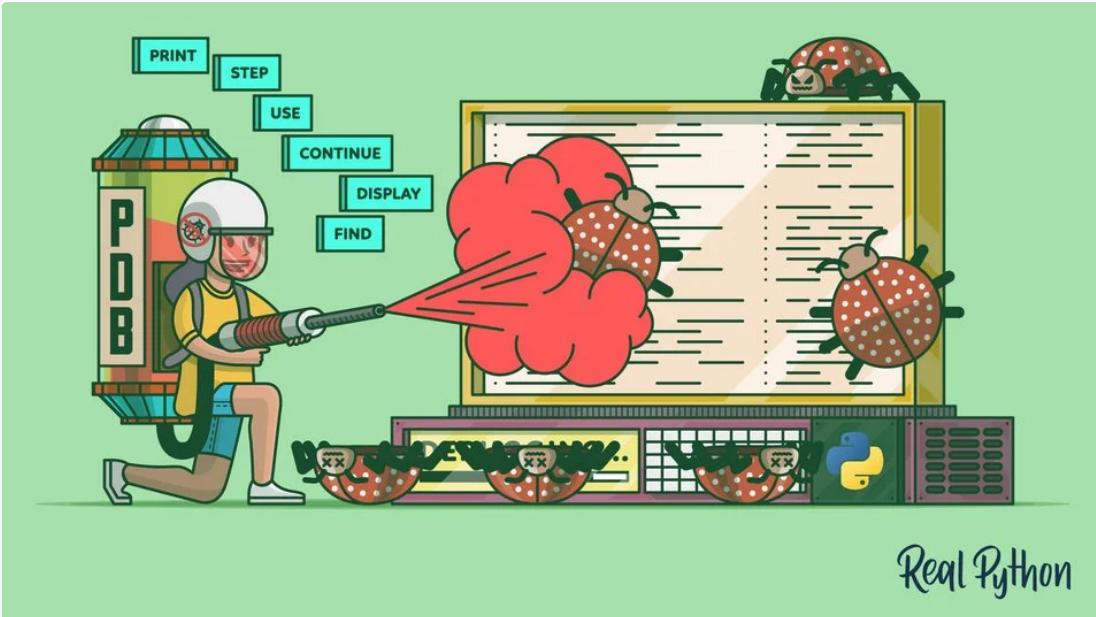
[Geir Arne](#)

[Jaya](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [basics](#) [python](#)



Real Python

Python Debugging With Pdb

by [Nathan Jennings](#) 9 Comments [intermediate](#) [python](#) [tools](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Getting Started: Printing a Variable's Value](#)
- [Printing Expressions](#)
- [Stepping Through Code](#)
 - [Listing Source Code](#)
- [Using Breakpoints](#)
- [Continuing Execution](#)
- [Displaying Expressions](#)
- [Python Caller ID](#)
- [Essential pdb Commands](#)
- [Python Debugging With pdb: Conclusion](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Debugging With pdb](#)

Debugging applications can sometimes be an unwelcome activity. You're busy working under a time crunch and you just want it to work. However, at other times, you might be learning a new language feature or experimenting with a new approach and want to understand more deeply how something is working.

Regardless of the situation, debugging code is a necessity, so it's a good idea to be comfortable working in a debugger. In this tutorial, I'll show you the basics of using pdb, Python's interactive source code debugger.

I'll walk you through a few common uses of pdb. You may want to bookmark this tutorial for quick reference later when you might really need it. pdb, and other debuggers, are indispensable tools. When you need a debugger, there's no substitute. You really need it.

By the end of this tutorial, you'll know how to use the debugger to see the state of any variable in your application. You'll also be able to stop and resume your application's flow of execution at any moment, so you can see exactly

how each line of code affects its internal state.

This is great for tracking down hard-to-find bugs and allows you to fix faulty code more quickly and reliably. Sometimes, stepping through code in pdb and seeing how values change can be a real eye-opener and lead to “aha” moments, along with the occasional “face palm”.

pdb is part of Python’s standard library, so it’s always there and available for use. This can be a life saver if you need to debug code in an environment where you don’t have access to the GUI debugger you’re familiar with.

The example code in this tutorial uses Python 3.6. You can find the source code for these examples on [GitHub](#).

At the end of this tutorial, there is a quick reference for [Essential pdb Commands](#).

There’s also a printable pdb Command Reference you can use as a cheat sheet while debugging:

Free Bonus: [Click here to get a printable "pdb Command Reference" \(PDF\)](#) that you can keep on your desk and refer to while debugging.

Getting Started: Printing a Variable’s Value

In this first example, we’ll look at using pdb in its simplest form: checking the value of a variable.

Insert the following code at the location where you want to break into the debugger:

Python

```
import pdb; pdb.set_trace()
```

When the line above is executed, Python stops and waits for you to tell it what to do next. You’ll see a (Pdb) prompt. This means that you’re now paused in the interactive debugger and can enter a command.

Starting in Python 3.7, [there’s another way to enter the debugger](#). PEP 553 describes the built-in function `breakpoint()`, which makes entering the debugger easy and consistent:

Python

```
breakpoint()
```

By default, `breakpoint()` will [import](#) pdb and call `pdb.set_trace()`, as shown above. However, using `breakpoint()` is more flexible and allows you to control debugging behavior via its API and use of the environment variable `PYTHONBREAKPOINT`. For example, setting `PYTHONBREAKPOINT=0` in your environment will completely disable `breakpoint()`, thus disabling debugging. If you’re using Python 3.7 or later, I encourage you to use `breakpoint()` instead of `pdb.set_trace()`.

You can also break into the debugger, without modifying the source and using `pdb.set_trace()` or `breakpoint()`, by running Python directly from the command-line and passing the option `-m pdb`. If your application accepts command-line arguments, pass them as you normally would after the filename. For example:

Shell

```
$ python3 -m pdb app.py arg1 arg2
```

There are a lot of pdb commands available. At the end of this tutorial, there is a list of [Essential pdb Commands](#). For now, let’s use the `p` command to print a variable’s value. Enter `p variable_name` at the (Pdb) prompt to print its value.

Let’s look at the example. Here’s the `example1.py` source:

Python

```
#!/usr/bin/env python3

filename = __file__
import pdb; pdb.set_trace()
print(f'path = {filename}')
```

If you run this from your shell, you should get the following output:

Shell

```
$ ./example1.py
> /code/example1.py(5)<module>()
-> print(f'path = {filename}')
(Pdb)
```

If you're having trouble getting the examples or your own code to run from the command line, read [How Do I Make My Own Command-Line Commands Using Python?](#) If you're on Windows, check the [Python Windows FAQ](#).

Now enter `p filename`. You should see:

Shell

```
(Pdb) p filename
'./example1.py'
(Pdb)
```

Since you're in a shell and using a CLI (command-line interface), pay attention to the characters and formatting. They'll give you the context you need:

- `>` starts the 1st line and tells you which source file you're in. After the filename, there is the current line number in parentheses. Next is the name of the function. In this example, since we're not paused inside a function and at module level, we see `<module>()`.
- `->` starts the 2nd line and is the current source line where Python is paused. This line hasn't been executed yet. In this example, this is line 5 in `example1.py`, from the `>` line above.
- `(Pdb)` is pdb's prompt. It's waiting for a command.

Use the command `q` to quit debugging and exit.

Printing Expressions

When using the `print` command `p`, you're passing an expression to be evaluated by Python. If you pass a variable name, pdb prints its current value. However, you can do much more to investigate the state of your running application.

In this example, the function `get_path()` is called. To inspect what's happening in this function, I've inserted a call to `pdb.set_trace()` to pause execution just before it returns:

Python

```
#!/usr/bin/env python3

import os

def get_path(filename):
    """Return file's path or empty string if no path."""
    head, tail = os.path.split(filename)
    import pdb; pdb.set_trace()
    return head
```

```
filename = __file__
print(f'path = {get_path(filename)}')
```

If you run this from your shell, you should get the output:

Shell

```
$ ./example2.py
> /code/example2.py(10)get_path()
-> return head
(Pdb)
```

Where are we?

- >: We're in the source file example2.py on line 10 in the function get_path(). This is the frame of reference the p command will use to resolve variable names, i.e. the current scope or context.
- -: Execution has paused at return head. This line hasn't been executed yet. This is line 10 in example2.py in the function get_path(), from the > line above.

Let's print some expressions to look at the current state of the application. I use the command ll (longlist) initially to list the function's source:

Shell

```
(Pdb) ll
 6     def get_path(filename):
 7         """Return file's path or empty string if no path."""
 8         head, tail = os.path.split(filename)
 9         import pdb; pdb.set_trace()
10    ->    return head
(Pdb) p filename
'./example2.py'
(Pdb) p head, tail
('.', 'example2.py')
(Pdb) p 'filename: ' + filename
'filename: ./example2.py'
(Pdb) p get_path
<function get_path at 0x100760e18>
(Pdb) p getattr(get_path, '__doc__')
"Return file's path or empty string if no path."
(Pdb) p [os.path.split(p)[1] for p in os.path.sys.path]
['pdb-basics', 'python36.zip', 'python3.6', 'lib-dynload', 'site-packages']
(Pdb)
```

You can pass any valid Python expression to p for evaluation.

This is especially helpful when you are debugging and want to test an alternative implementation directly in the application at runtime.

You can also use the command pp (pretty-print) to pretty-print expressions. This is helpful if you want to print a variable or expression with a large amount of output, e.g. lists and dictionaries. Pretty-printing keeps objects on a single line if it can or breaks them onto multiple lines if they don't fit within the allowed width.

Stepping Through Code

There are two commands you can use to step through code when debugging:

Command	Description
n (next)	Continue execution until the next line in the current function is reached or it returns.
s (step)	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).

There's a 3rd command named unt (until). It is related to n (next). We'll look at it later in this tutorial in the

section [Continuing Execution](#).

The difference between n (next) and s (step) is where pdb stops.

Use n (next) to continue execution until the next line and stay within the current function, i.e. not stop in a foreign function if one is called. Think of next as “staying local” or “step over”.

Use s (step) to execute the current line and stop in a foreign function if one is called. Think of step as “step into”. If execution is stopped in another function, s will print --Call--.

Both n and s will stop execution when the end of the current function is reached and print --Return-- along with the return value at the end of the next line after ->.

Let's look at an example using both commands. Here's the `example3.py` source:

Python

```
#!/usr/bin/env python3

import os

def get_path(filename):
    """Return file's path or empty string if no path."""
    head, tail = os.path.split(filename)
    return head

filename = __file__
import pdb; pdb.set_trace()
filename_path = get_path(filename)
print(f'path = {filename_path}')
```

If you run this from your shell and enter n, you should get the output:

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) n
> /code/example3.py(15)<module>()
-> print(f'path = {filename_path}')
(Pdb)
```

With n (next), we stopped on line 15, the next line. We “stayed local” in <module>() and “stepped over” the call to `get_path()`. The function is <module>() since we're currently at module level and not paused inside another function.

Let's try s:

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) s
--Call--
> /code/example3.py(6)get_path()
-> def get_path(filename):
(Pdb)
```

With s (step), we stopped on line 6 in the function `get_path()` since it was called on line 14. Notice the line --Call-- after the s command.

Conveniently, pdb remembers your last command. If you're stepping through a lot of code, you can just press Enter ↵ to repeat the last command.

Below is an example of using both s and n to step through the code. I enter s initially because I want to “step into”

the function `get_path()` and stop. Then I enter `n` once to “stay local” or “step over” any other function calls and just press `Enter ↵` to repeat the `n` command until I get to the last source line.

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) s
--Call--
> /code/example3.py(6)get_path()
-> def get_path(filename):
(Pdb) n
> /code/example3.py(8)get_path()
-> head, tail = os.path.split(filename)
(Pdb)
> /code/example3.py(9)get_path()
-> return head
(Pdb)
--Return--
> /code/example3.py(9)get_path()-> '..'
-> return head
(Pdb)
> /code/example3.py(15)<module>()
-> print(f'path = {filename_path}')
(Pdb)
path =
--Return--
> /code/example3.py(15)<module>()->None
-> print(f'path = {filename_path}')
(Pdb)
```

Note the lines `--Call--` and `--Return--`. This is pdb letting you know why execution was stopped. `n` (next) and `s` (step) will stop before a function returns. That's why you see the `--Return--` lines above.

Also note `-> '..'` at the end of the line after the first `--Return--` above:

Shell

```
--Return--
> /code/example3.py(9)get_path()-> '..'
-> return head
(Pdb)
```

When pdb stops at the end of a function before it returns, it also prints the return value for you. In this example it's `'..'`.

Listing Source Code

Don't forget the command `ll` (longlist: list the whole source code for the current function or frame). It's really helpful when you're stepping through unfamiliar code or you just want to see the entire function for context.

Here's an example:

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) s
--Call--
> /code/example3.py(6)get_path()
-> def get_path(filename):
(Pdb) ll
   6  -> def get_path(filename):
   7      """Return file's path or empty string if no path."""
   8      head, tail = os.path.split(filename)
   9      return head
(Pdb)
```

To see a shorter snippet of code, use the command `l (list)`. Without arguments, it will print 11 lines around the current line or continue the previous listing. Pass the argument `.` to always list 11 lines around the current line: `l .`

Shell

```
$ ./example3.py
> /code/example3.py(14)<module>()
-> filename_path = get_path(filename)
(Pdb) 1
    9         return head
  10
  11
  12     filename = __file__
  13     import pdb; pdb.set_trace()
  14 -> filename_path = get_path(filename)
  15     print(f'path = {filename_path}')
[EOF]
(Pdb) 1
[EOF]
(Pdb) 1 .
    9         return head
  10
  11
  12     filename = __file__
  13     import pdb; pdb.set_trace()
  14 -> filename_path = get_path(filename)
  15     print(f'path = {filename_path}')
[EOF]
(Pdb)
```

Using Breakpoints

Breakpoints are very convenient and can save you a lot of time. Instead of stepping through dozens of lines you're not interested in, simply create a breakpoint where you want to investigate. Optionally, you can also tell pdb to break only when a certain condition is true.

Use the command `b (break)` to set a breakpoint. You can specify a line number or a function name where execution is stopped.

The syntax for break is:

Shell

```
b(reak) [ ([filename:]lineno | function) [, condition] ]
```

If `filename:` is not specified before the line number `lineno`, then the current source file is used.

Note the optional 2nd argument to `b: condition`. This is very powerful. Imagine a situation where you wanted to break only if a certain condition existed. If you pass a Python expression as the 2nd argument, pdb will break when the expression evaluates to true. We'll do this in an example below.

In this example, there's a utility module `util.py`. Let's set a breakpoint to stop execution in the function `get_path()`.

Here's the source for the main script `example4.py`:

Python

```
#!/usr/bin/env python3

import util

filename = __file__
import pdb; pdb.set_trace()
filename_path = util.get_path(filename)
print(f'path = {filename_path}')
```

Here's the source for the utility module util.py:

Python

```
def get_path(filename):
    """Return file's path or empty string if no path."""
    import os
    head, tail = os.path.split(filename)
    return head
```

First, let's set a breakpoint using the source filename and line number:

Shell

```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util:5
Breakpoint 1 at /code/util.py:5
(Pdb) c
> /code/util.py(5)get_path()
-> return head
(Pdb) p filename, head, tail
('~/example4.py', '.', 'example4.py')
(Pdb)
```

The command c (continue) continues execution until a breakpoint is found.

Next, let's set a breakpoint using the function name:

Shell

```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util.get_path
Breakpoint 1 at /code/util.py:1
(Pdb) c
> /code/util.py(3)get_path()
-> import os
(Pdb) p filename
'~/example4.py'
(Pdb)
```

Enter b with no arguments to see a list of all breakpoints:

Shell

```
(Pdb) b
Num Type      Disp Enb  Where
1  breakpoint  keep yes  at /code/util.py:1
(Pdb)
```

You can disable and re-enable breakpoints using the command `disable bpnumber` and `enable bpnumber`. `bpnumber` is the breakpoint number from the breakpoints list's 1st column `Num`. Notice the `Enb` column's value change:

Shell

```
(Pdb) disable 1
Disabled breakpoint 1 at /code/util.py:1
(Pdb) b
Num Type      Disp Enb  Where
1  breakpoint  keep no   at /code/util.py:1
(Pdb)
```

```
1 breakpoint keep no at /code/util.py:1
(Pdb) enable 1
Enabled breakpoint 1 at /code/util.py:1
(Pdb) b
Num Type Disp Enb Where
1 breakpoint keep yes at /code/util.py:1
(Pdb)
```

To delete a breakpoint, use the command `c1` (clear):

Shell

```
cl(ear) filename:lineno
cl(ear) [bpnumber [bpnumber...]]
```

Now let's use a Python expression to set a breakpoint. Imagine a situation where you wanted to break only if your troubled function received a certain input.

In this example scenario, the `get_path()` function is failing when it receives a relative path, i.e. the file's path doesn't start with `/`. I'll create an expression that evaluates to true in this case and pass it to `b` as the 2nd argument:

Shell

```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util.get_path, not filename.startswith('/')
Breakpoint 1 at /code/util.py:1
(Pdb) c
> /code/util.py(3)get_path()
-> import os
(Pdb) a
filename = './example4.py'
(Pdb)
```

After you create the breakpoint above and enter `c` to continue execution, `pdb` stops when the expression evaluates to true. The command `a (args)` prints the argument list of the current function.

In the example above, when you're setting the breakpoint with a function name rather than a line number, note that the expression should use only function arguments or global variables that are available at the time the function is entered. Otherwise, the breakpoint will stop execution in the function regardless of the expression's value.

If you need to break using an expression with a variable name located inside a function, i.e. a variable name not in the function's argument list, specify the line number:

Shell

```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util:5, not head.startswith('/')
Breakpoint 1 at /code/util.py:5
(Pdb) c
> /code/util.py(5)get_path()
-> return head
(Pdb) p head
'.'
```

```
(Pdb) a
filename = './example4.py'
(Pdb)
```

You can also set a temporary breakpoint using the command `tbreak`. It's removed automatically when it's first hit. It uses the same arguments as `b`.

Continuing Execution

So far, we've looked at stepping through code with `n` (next) and `s` (step) and using breakpoints with `b` (break) and `c` (continue).

There's also a related command: `unt` (until).

Use `unt` to continue execution like `c`, but stop at the next line greater than the current line. Sometimes `unt` is more convenient and quicker to use and is exactly what you want. I'll demonstrate this with an example below.

Let's first look at the syntax and description for `unt`:

Command	Syntax	Description
unt	unt([lineno])	Without <code>lineno</code> , continue execution until the line with a number greater than the current one is reached. With <code>lineno</code> , continue execution until a line with a number greater or equal to that is reached. In both cases, also stop when the current frame returns.

Depending on whether or not you pass the line number argument `lineno`, `unt` can behave in two ways:

- Without `lineno`, continue execution until the line with a number greater than the current one is reached. This is similar to `n` (next). It's an alternate way to execute and "step over" code. The difference between `n` and `unt` is that `unt` stops only when a line with a number greater than the current one is reached. `n` will stop at the next logically executed line.
- With `lineno`, continue execution until a line with a number greater or equal to that is reached. This is like `c` (continue) with a line number argument.

In both cases, `unt` stops when the current frame (function) returns, just like `n` (next) and `s` (step).

The primary behavior to note with `unt` is that it will stop when a line number **greater or equal** to the current or specified line is reached.

Use `unt` when you want to continue execution and stop farther down in the current source file. You can treat it like a hybrid of `n` (next) and `b` (break), depending on whether you pass a line number argument or not.

In the example below, there is a function with a loop. Here, you want to continue execution of the code and stop after the loop, without stepping through each iteration of the loop or setting a breakpoint:

Here's the example source for `example4unt.py`:

```
Python
#!/usr/bin/env python3

import os

def get_path(fname):
    """Return file's path or empty string if no path."""
    import pdb; pdb.set_trace()
    head, tail = os.path.split(fname)
    for char in tail:
        pass # Check filename char
    return head


filename = __file__
filename_path = get_path(filename)
```

```
print(f'path = {filename_path}')
```

And the console output using unt:

Shell

```
$ ./example4unt.py
> /code/example4unt.py(9)get_path()
-> head, tail = os.path.split(fname)
(Pdb) l1
 6     def get_path(fname):
 7         """Return file's path or empty string if no path."""
 8         import pdb; pdb.set_trace()
 9 ->     head, tail = os.path.split(fname)
10        for char in tail:
11            pass # Check filename char
12        return head
(Pdb) unt
> /code/example4unt.py(10)get_path()
-> for char in tail:
(Pdb)
> /code/example4unt.py(11)get_path()
-> pass # Check filename char
(Pdb)
> /code/example4unt.py(12)get_path()
-> return head
(Pdb) p char, tail
('y', 'example4unt.py')
```

The l1 command was used first to print the function's source, followed by unt. pdb remembers the last command entered, so I just pressed to repeat the unt command. This continued execution through the code until a source line greater than the current line was reached.

Note in the console output above that pdb stopped only once on lines 10 and 11. Since unt was used, execution was stopped only in the 1st iteration of the loop. However, each iteration of the loop was executed. This can be verified in the last line of output. The char variable's value 'y' is equal to the last character in tail's value 'example4unt.py'.

Displaying Expressions

Similar to printing expressions with p and pp, you can use the command `display [expression]` to tell pdb to automatically display the value of an expression, if it changed, when execution stops. Use the command `undisplay [expression]` to clear a display expression.

Here's the syntax and description for both commands:

Command	Syntax	Description
display	display [expression]	Display the value of expression if it changed, each time execution stops in the current frame. Without expression, list all display expressions for the current frame.
undisplay	undisplay [expression]	Do not display expression any more in the current frame. Without expression, clear all display expressions for the current frame.

Below is an example, `example4display.py`, demonstrating its use with a loop:

Shell

```
$ ./example4display.py
> /code/example4display.py(9)get_path()
-> head, tail = os.path.split(fname)
(Pdb) l1
 6     def get_path(fname):
 7         """Return file's path or empty string if no path."""
 8         import pdb; pdb.set_trace()
 9 ->     head, tail = os.path.split(fname)
10     for char in tail:
11         pass # Check filename char
12     return head
(Pdb) b 11
Breakpoint 1 at /code/example4display.py:11
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
(Pdb) display char
display char: 'e'
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'x' [old: 'e']
(Pdb)
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'a' [old: 'x']
(Pdb)
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'm' [old: 'a']
```

In the output above, `pdb` automatically displayed the value of the `char` variable because each time the breakpoint was hit its value had changed. Sometimes this is helpful and exactly what you want, but there's another way to use `display`.

You can enter `display` multiple times to build a watch list of expressions. This can be easier to use than `p`. After adding all of the expressions you're interested in, simply enter `display` to see the current values:

Shell

```
$ ./example4display.py
> /code/example4display.py(9)get_path()
-> head, tail = os.path.split(fname)
(Pdb) l1
 6     def get_path(fname):
 7         """Return file's path or empty string if no path."""
 8         import pdb; pdb.set_trace()
 9 ->     head, tail = os.path.split(fname)
10     for char in tail:
11         pass # Check filename char
12     return head
(Pdb) b 11
Breakpoint 1 at /code/example4display.py:11
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
(Pdb) display char
display char: 'e'
(Pdb) display fname
display fname: './example4display.py'
(Pdb) display head
display head: '.'
(Pdb) display tail
display tail: 'example4display.py'
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass # Check filename char
display char: 'x' [old: 'e']
(Pdb) display
Currently displaying:
char: 'x'
fname: './example4display.py'
head: '.'
tail: 'example4display.py'
```

Python Caller ID

In this last section, we'll build upon what we've learned so far and finish with a nice payoff. I use the name "caller ID" in reference to the phone system's caller identification feature. That is exactly what this example demonstrates, except it's applied to Python.

Here's the source for the main script `example5.py`:

Python

```
#!/usr/bin/env python3

import fileutil
```

```

def get_file_info(full_fname):
    file_path = fileutil.get_path(full_fname)
    return file_path

filename = __file__
filename_path = get_file_info(filename)
print(f'path = {filename_path}')

```

Here's the utility module `fileutil.py`:

Python

```

def get_path(fname):
    """Return file's path or empty string if no path."""
    import os
    import pdb; pdb.set_trace()
    head, tail = os.path.split(fname)
    return head

```

In this scenario, imagine there's a large code base with a function in a utility module, `get_path()`, that's being called with invalid input. However, it's being called from many places in different packages.

How do you find who the caller is?

Use the command `w` (where) to print a stack trace, with the most recent frame at the bottom:

Shell

```

$ ./example5.py
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) w
  /code/example5.py(12)<module>()
-> filename_path = get_file_info(filename)
  /code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb)

```

Don't worry if this looks confusing or if you're not sure what a stack trace or frame is. I'll explain those terms below. It's not as difficult as it might sound.

Since the most recent frame is at the bottom, start there and read from the bottom up. Look at the lines that start with `->`, but skip the 1st instance since that's where `pdb.set_trace()` was used to enter pdb in the function `get_path()`. In this example, the source line that called the function `get_path()` is:

Shell

```

-> file_path = fileutil.get_path(full_fname)

```

The line above each `->` contains the filename, line number (in parentheses), and function name the source line is in. So the caller is:

Shell

```

/code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)

```

That's no surprise in this small example for demonstration purposes, but imagine a large application where you've set a breakpoint with a condition to identify where a bad input value is originating.

Now we know how to find the caller.

But what about this stack trace and frame stuff?

A [stack trace](#) is just a list of all the frames that Python has created to keep track of function calls. A frame is a data structure Python creates when a function is called and deletes when it returns. The stack is simply an ordered list of frames or function calls at any point in time. The (function call) stack grows and shrinks throughout the life of an application as functions are called and then return.

When printed, this ordered list of frames, the stack, is called a [stack trace](#). You can see it at any time by entering the command `w`, as we did above to find the caller.

See this [call stack article on Wikipedia](#) for details.

To understand better and get more out of pdb, let's look more closely at the help for `w`:

Shell

```
(Pdb) h w
w(here)
    Print a stack trace, with the most recent frame at the bottom.
    An arrow indicates the "current frame", which determines the
    context of most commands. 'bt' is an alias for this command.
```

What does pdb mean by “current frame”?

Think of the current frame as the current function where pdb has stopped execution. In other words, the current frame is where your application is currently paused and is used as the “frame” of reference for pdb commands like `p` (`print`).

`p` and other commands will use the current frame for context when needed. In the case of `p`, the current frame will be used for looking up and printing variable references.

When pdb prints a stack trace, an arrow > indicates the current frame.

How is this useful?

You can use the two commands `u` (up) and `d` (down) to change the current frame. Combined with `p`, this allows you to inspect variables and state in your application at any point along the call stack in any frame.

Here's the syntax and description for both commands:

Command	Syntax	Description
<code>u</code>	<code>u(p) [count]</code>	Move the current frame count (default one) levels up in the stack trace (to an older frame).
<code>d</code>	<code>d(own) [count]</code>	Move the current frame count (default one) levels down in the stack trace (to a newer frame).

Let's look at an example using the `u` and `d` commands. In this scenario, we want to inspect the variable `full_fname` that's local to the function `get_file_info()` in `example5.py`. In order to do this, we have to change the current frame up one level using the command `u`:

Shell

```
$ ./example5.py
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) w
  /code/example5.py(12)<module>()
-> filename_path = get_file_info(filename)
  /code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) u
> /code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
(Pdb) p full_fname
'./example5.py'
(Pdb) d
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) p fname
'./example5.py'
(Pdb)
```

The call to `pdb.set_trace()` is in `fileutil.py` in the function `get_path()`, so the current frame is initially set there. You can see it in the 1st line of output above:

Shell

```
> /code/fileutil.py(5)get_path()
```

To access and print the local variable `full_fname` in the function `get_file_info()` in `example5.py`, the command `u` was used to move up one level:

Shell

```
(Pdb) u
> /code/example5.py(7)get_file_info()
-> file_path = fileutil.get_path(full_fname)
```

Note in the output of `u` above that `pdb` printed the arrow `>` at the beginning of the 1st line. This is `pdb` letting you know the frame was changed and this source location is now the current frame. The variable `full_fname` is accessible now. Also, it's important to realize the source line starting with `->` on the 2nd line has been executed. Since the frame was moved up the stack, `fileutil.get_path()` has been called. Using `u`, we moved up the stack (in a sense, back in time) to the function `example5.get_file_info()` where `fileutil.get_path()` was called.

Continuing with the example, after `full_fname` was printed, the current frame was moved to its original location using `d`, and the local variable `fname` in `get_path()` was printed.

If we wanted to, we could have moved multiple frames at once by passing the count argument to `u` or `d`. For example, we could have moved to module level in `example5.py` by entering `u 2`:

Shell

```
$ ./example5.py
> /code/fileutil.py(5)get_path()
-> head, tail = os.path.split(fname)
(Pdb) u 2
> /code/example5.py(12)<module>()
-> filename_path = get_file_info(filename)
(Pdb) p filename
'./example5.py'
(Pdb)
```

It's easy to forget where you are when you're debugging and thinking of many different things. Just remember you can always use the aptly named command `w` (where) to see where execution is paused and what the current frame is.

Essential pdb Commands

Once you've spent a little time with pdb, you'll realize a little knowledge goes a long way. Help is always available with the `h` command.

Just enter `h` or `help <topic>` to get a list of all commands or help for a specific command or topic.

For quick reference, here's a list of essential commands:

Command	Description
<code>p</code>	Print the value of an expression.
<code>pp</code>	Pretty-print the value of an expression.
<code>n</code>	Continue execution until the next line in the current function is reached or it returns.
<code>s</code>	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).
<code>c</code>	Continue execution and only stop when a breakpoint is encountered.
<code>unt</code>	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
<code>l</code>	List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.
<code>ll</code>	List the whole source code for the current function or frame.
<code>b</code>	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file.
<code>w</code>	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.
<code>u</code>	Move the current frame count (default one) levels up in the stack trace (to an older frame).
<code>d</code>	Move the current frame count (default one) levels down in the stack trace (to a newer frame).
<code>h</code>	See a list of available commands.
<code>h <topic></code>	Show help for a command or topic.
<code>h pdb</code>	Show the full pdb documentation.
<code>q</code>	Quit the debugger and exit.

Python Debugging With pdb: Conclusion

In this tutorial, we covered a few basic and common uses of pdb:

- printing expressions

- stepping through code with n (next) and s (step)
- using breakpoints
- continuing execution with unt (until)
- displaying expressions
- finding the caller of a function

I hope it's been helpful to you. If you're curious about learning more, see:

- pdb's full documentation at a pdb prompt near you: (Pdb) h pdb
- [Python's pdb docs](#)

The source code used in the examples can be found on the associated [GitHub repository](#). Be sure to check out our printable pdb Command Reference, which you can use as a cheat sheet while debugging:

Free Bonus: [Click here to get a printable "pdb Command Reference" \(PDF\)](#) that you can keep on your desk and refer to while debugging.

Also, if you'd like to try a GUI-based Python debugger, read our [Python IDEs and Editors Guide](#) to see what options will work best for you. Happy Pythoning!

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Debugging With pdb](#)

About Nathan Jennings

Nathan is a member of the Real Python tutorial team who started his programmer career with C long time ago, but eventually found Python. From web applications and data collection to networking and network security, he enjoys all things Pythonic.

[» More about Nathan](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Dan](#)

[Jim](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#) [tools](#)

Recommended Video Course: [Python Debugging With pdb](#)



Real Python

Getting Started With Testing in Python

by Anthony Shaw 32 Comments best-practices intermediate testing

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Testing Your Code](#)
 - [Automated vs. Manual Testing](#)
 - [Unit Tests vs. Integration Tests](#)
 - [Choosing a Test Runner](#)
- [Writing Your First Test](#)
 - [Where to Write the Test](#)
 - [How to Structure a Simple Test](#)
 - [How to Write Assertions](#)
 - [Side Effects](#)
- [Executing Your First Test](#)
 - [Executing Test Runners](#)
 - [Understanding Test Output](#)
 - [Running Your Tests From PyCharm](#)
 - [Running Your Tests From Visual Studio Code](#)
- [Testing for Web Frameworks Like Django and Flask](#)
 - [Why They're Different From Other Applications](#)
 - [How to Use the Django Test Runner](#)
 - [How to Use unittest and Flask](#)
- [More Advanced Testing Scenarios](#)
 - [Handling Expected Failures](#)
 - [Isolating Behaviors in Your Application](#)
 - [Writing Integration Tests](#)
 - [Testing Data-Driven Applications](#)
- [Testing in Multiple Environments](#)
 - [Installing Tox](#)
 - [Configuring Tox for Your Dependencies](#)
 - [Executing Tox](#)
- [Automating the Execution of Your Tests](#)
 - [Introducing Linters Into Your Application](#)
 - [Keeping Your Test Code Clean](#)
- [What's Next](#)

- [Testing for Performance Degradation Between Changes](#)
- [Testing for Security Flaws in Your Application](#)
- [Conclusion](#)



Your Guided Tour Through the Python 3.9 Interpreter »

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Test-Driven Development With PyTest](#)

This tutorial is for anyone who has written a fantastic application in Python but hasn't yet written any tests.

Testing in Python is a huge topic and can come with a lot of complexity, but it doesn't need to be hard. You can get started creating simple tests for your application in a few easy steps and then build on it from there.

In this tutorial, you'll learn how to create a basic test, execute it, and find the bugs before your users do! You'll learn about the tools available to write and execute tests, check your application's performance, and even look for security issues.

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Testing Your Code

There are many ways to test your code. In this tutorial, you'll learn the techniques from the most basic steps and work towards advanced methods.

Automated vs. Manual Testing

The good news is, you've probably already created a test without realizing it. Remember when you ran your application and used it for the first time? Did you check the features and experiment using them? That's known as **exploratory testing** and is a form of manual testing.

Exploratory testing is a form of testing that is done without a plan. In an exploratory test, you're just exploring the application.

To have a complete set of manual tests, all you need to do is make a list of all the features your application has, the different types of input it can accept, and the expected results. Now, every time you make a change to your code, you need to go through every single item on that list and check it.

That doesn't sound like much fun, does it?

This is where automated testing comes in. Automated testing is the execution of your test plan (the parts of your application you want to test, the order in which you want to test them, and the expected responses) by a script instead of a human. Python already comes with a set of tools and libraries to help you create automated tests for your application. We'll explore those tools and libraries in this tutorial.

Unit Tests vs. Integration Tests

The world of testing has no shortage of terminology, and now that you know the difference between automated and manual testing, it's time to go a level deeper.

Think of how you might test the lights on a car. You would turn on the lights (known as the **test step**) and go outside the car or ask a friend to check that the lights are on (known as the **test assertion**). Testing multiple components is known as **integration testing**.

Think of all the things that need to work correctly in order for a simple task to give the right result. These components are like the parts to your application, all of those classes, functions, and modules you've written.

A major challenge with integration testing is when an integration test doesn't give the right result. It's very hard to diagnose the issue without being able to isolate which part of the system is failing. If the lights didn't turn on, then maybe the bulbs are broken. Is the battery dead? What about the alternator? Is the car's computer failing?

If you have a fancy modern car, it will tell you when your light bulbs have gone. It does this using a form of **unit test**.

A unit test is a smaller test, one that checks that a single component operates in the right way. A unit test helps you to isolate what is broken in your application and fix it faster.

You have just seen two types of tests:

1. An integration test checks that components in your application operate with each other.
2. A unit test checks a small component in your application.

You can write both integration tests and unit tests in Python. To write a unit test for the built-in function `sum()`, you would check the output of `sum()` against a known output.

For example, here's how you check that the `sum()` of the numbers (1, 2, 3) equals 6:

```
Python >>>
>>> assert sum([1, 2, 3]) == 6, "Should be 6"
```

This will not output anything on the REPL because the values are correct.

If the result from `sum()` is incorrect, this will fail with an `AssertionError` and the message "Should be 6". Try an assertion statement again with the wrong values to see an `AssertionError`:

```
Python >>>
>>> assert sum([1, 1, 1]) == 6, "Should be 6"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Should be 6
```

In the REPL, you are seeing the raised `AssertionError` because the result of `sum()` does not match 6.

Instead of testing on the REPL, you'll want to put this into a new Python file called `test_sum.py` and execute it again:

```
Python
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    print("Everything passed")
```

Now you have written a **test case**, an assertion, and an entry point (the command line). You can now execute this at the command line:

```
Shell
$ python test_sum.py
Everything passed
```

You can see the successful result, `Everything passed`.

In Python, `sum()` accepts any iterable as its first argument. You tested with a list. Now test with a tuple as well. Create a new file called `test_sum_2.py` with the following code:

Python

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    test_sum_tuple()
    print("Everything passed")
```

When you execute `test_sum_2.py`, the script will give an error because the `sum()` of `(1, 2, 2)` is 5, not 6. The result of the script gives you the error message, the line of code, and the traceback:

Shell

```
$ python test_sum_2.py
Traceback (most recent call last):
  File "test_sum_2.py", line 9, in <module>
    test_sum_tuple()
  File "test_sum_2.py", line 5, in test_sum_tuple
    assert sum((1, 2, 2)) == 6, "Should be 6"
AssertionError: Should be 6
```

Here you can see how a mistake in your code gives an error on the console with some information on where the error was and what the expected result was.

Writing tests in this way is okay for a simple check, but what if more than one fails? This is where test runners come in. The test runner is a special application designed for running tests, checking the output, and giving you tools for debugging and diagnosing tests and applications.

Choosing a Test Runner

There are many test runners available for Python. The one built into the Python standard library is called `unittest`. In this tutorial, you will be using `unittest` test cases and the `unittest` test runner. The principles of `unittest` are easily portable to other frameworks. The three most popular test runners are:

- `unittest`
- `nose` or `nose2`
- `pytest`

Choosing the best test runner for your requirements and level of experience is important.

unittest

`unittest` has been built into the Python standard library since version 2.1. You'll probably see it in commercial Python applications and open-source projects.

`unittest` contains both a testing framework and a test runner. `unittest` has some important requirements for writing and executing tests.

`unittest` requires that:

- You put your tests into classes as methods
- You use a series of special assertion methods in the `unittest.TestCase` class instead of the built-in `assert` statement

To convert the earlier example to a `unittest` test case, you would have to:

1. Import `unittest` from the standard library
2. Create a class called `TestSum` that inherits from the `TestCase` class
3. Convert the test functions into methods by adding `self` as the first argument
4. Change the assertions to use the `self.assertEqual()` method on the `TestCase` class
5. Change the command-line entry point to call `unittest.main()`

3. Change the command line entry point to call `unittest.main()`

Follow those steps by creating a new file `test_sum_unittest.py` with the following code:

Python

```
import unittest

class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

if __name__ == '__main__':
    unittest.main()
```

If you execute this at the command line, you'll see one success (indicated with .) and one failure (indicated with F):

Shell

```
$ python test_sum_unittest.py
.F
=====
FAIL: test_sum_tuple ( __main__.TestSum )
-----
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

-----
Ran 2 tests in 0.001s

FAILED (failures=1)
```

You have just executed two tests using the `unittest` test runner.

Note: Be careful if you're writing test cases that need to execute in both Python 2 and 3. In Python 2.7 and below, `unittest` is called `unittest2`. If you simply `import` from `unittest`, you will get different versions with different features between Python 2 and 3.

For more information on `unittest`, you can explore the [unittest Documentation](#).

nose

You may find that over time, as you write hundreds or even thousands of tests for your application, it becomes increasingly hard to understand and use the output from `unittest`.

`nose` is compatible with any tests written using the `unittest` framework and can be used as a drop-in replacement for the `unittest` test runner. The development of `nose` as an open-source application fell behind, and a fork called `nose2` was created. If you're starting from scratch, it is recommended that you use `nose2` instead of `nose`.

To get started with `nose2`, install `nose2` from PyPI and execute it on the command line. `nose2` will try to discover all test scripts named `test*.py` and test cases inheriting from `unittest.TestCase` in your current directory:

Shell

```
$ pip install nose2
$ python -m nose2
.F
=====
FAIL: test_sum_tuple ( __main__.TestSum )
-----
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6
```

```
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```

You have just executed the test you created in `test_sum_unittest.py` from the nose2 test runner. nose2 offers many command-line flags for filtering the tests that you execute. For more information, you can explore the [Nose 2 documentation](#).

pytest

[pytest](#) supports execution of `unittest` test cases. The real advantage of pytest comes by writing pytest test cases. pytest test cases are a series of functions in a Python file starting with the name `test_`.

pytest has some other great features:

- Support for the built-in `assert` statement instead of using special `self.assert*`() methods
- Support for filtering for test cases
- Ability to rerun from the last failing test
- An ecosystem of hundreds of plugins to extend the functionality

Writing the `TestSum` test case example for pytest would look like this:

Python

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"
```

You have dropped the `TestCase`, any use of classes, and the command-line entry point.

More information can be found at the [Pytest Documentation Website](#).

Writing Your First Test

Let's bring together what you've learned so far and, instead of testing the built-in `sum()` function, test a simple implementation of the same requirement.

Create a new project folder and, inside that, create a new folder called `my_sum`. Inside `my_sum`, create an empty file called `__init__.py`. Creating the `__init__.py` file means that the `my_sum` folder can be imported as a module from the parent directory.

Your project folder should look like this:

```
project/
  |
  └── my_sum/
      └── __init__.py
```

Open up `my_sum/__init__.py` and create a new function called `sum()`, which takes an iterable (a list, tuple, or set) and adds the values together:

Python

```
def sum(arg):
    total = 0
    for val in arg:
        total += val
    return total
```

This code example creates a variable called `total`, iterates over all the values in `arg`, and adds them to `total`. It then returns the result once the iterable has been exhausted.

Where to Write the Test

To get started writing tests, you can simply create a file called `test.py`, which will contain your first test case. Because the file will need to be able to import your application to be able to test it, you want to place `test.py` above the package folder, so your directory tree will look something like this:

```
project/
|
|--- my_sum/
|     |--- __init__.py
|
|--- test.py
```

You'll find that, as you add more and more tests, your single file will become cluttered and hard to maintain, so you can create a folder called `tests/` and split the tests into multiple files. It is convention to ensure each file starts with `test_` so all test runners will assume that Python file contains tests to be executed. Some very large projects split tests into more subdirectories based on their purpose or usage.

Note: What if your application is a single script?

You can import any attributes of the script, such as classes, functions, and variables by using the built-in `__import__()` function. Instead of `from my_sum import sum`, you can write the following:

Python

```
target = __import__("my_sum.py")
sum = target.sum
```

The benefit of using `__import__()` is that you don't have to turn your project folder into a package, and you can specify the file name. This is also useful if your filename collides with any standard library packages. For example, `math.py` would collide with the `math` module.

How to Structure a Simple Test

Before you dive into writing tests, you'll want to first make a couple of decisions:

1. What do you want to test?
2. Are you writing a unit test or an integration test?

Then the structure of a test should loosely follow this workflow:

1. Create your inputs
2. Execute the code being tested, capturing the output
3. Compare the output with an expected result

For this application, you're testing `sum()`. There are many behaviors in `sum()` you could check, such as:

- Can it sum a list of whole numbers (integers)?
- Can it sum a tuple or set?
- Can it sum a list of floats?
- What happens when you provide it with a bad value, such as a single integer or a string?
- What happens when one of the values is negative?

The most simple test would be a list of integers. Create a file, `test.py` with the following Python code:

Python

```
import unittest

from my_sum import sum

class TestSum(unittest.TestCase):
    def test_list_int(self):
        """
```

```

Test that it can sum a list of integers
"""
data = [1, 2, 3]
result = sum(data)
self.assertEqual(result, 6)

if __name__ == '__main__':
    unittest.main()

```

This code example:

1. Imports `sum()` from the `my_sum` package you created
2. Defines a new test case class called `TestSum`, which inherits from `unittest.TestCase`
3. Defines a test method, `.test_list_int()`, to test a list of integers. The method `.test_list_int()` will:
 - o Declare a variable `data` with a list of numbers `(1, 2, 3)`
 - o Assign the result of `my_sum.sum(data)` to a `result` variable
 - o Assert that the value of `result` equals `6` by using the `.assertEqual()` method on the `unittest.TestCase` class
4. Defines a command-line entry point, which runs the `unittest` test-runner `.main()`

If you're unsure what `self` is or how `.assertEqual()` is defined, you can brush up on your object-oriented programming with [Python 3 Object-Oriented Programming](#).

How to Write Assertions

The last step of writing a test is to validate the output against a known response. This is known as an **assertion**. There are some general best practices around how to write assertions:

- Make sure tests are repeatable and run your test multiple times to make sure it gives the same result every time
- Try and assert results that relate to your input data, such as checking that the result is the actual sum of values in the `sum()` example

`unittest` comes with lots of methods to assert on the values, types, and existence of variables. Here are some of the most commonly used methods:

Method	Equivalent to
<code>.assertEqual(a, b)</code>	<code>a == b</code>
<code>.assertTrue(x)</code>	<code>bool(x) is True</code>
<code>.assertFalse(x)</code>	<code>bool(x) is False</code>
<code>.assertIs(a, b)</code>	<code>a is b</code>
<code>.assertIsNone(x)</code>	<code>x is None</code>
<code>.assertIn(a, b)</code>	<code>a in b</code>
<code>.assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>

`.assertIs()`, `.assertIsNone()`, `.assertIn()`, and `.assertIsInstance()` all have opposite methods, named `.assert IsNot()`, and so forth.

Side Effects

When you're writing tests, it's often not as simple as looking at the return value of a function. Often, executing a piece of code will alter other things in the environment, such as the attribute of a class, a file on the filesystem, or a

value in a database. These are known as **side effects** and are an important part of testing. Decide if the side effect is being tested before including it in your list of assertions.

If you find that the unit of code you want to test has lots of side effects, you might be breaking the [Single Responsibility Principle](#). Breaking the Single Responsibility Principle means the piece of code is doing too many things and would be better off being refactored. Following the Single Responsibility Principle is a great way to design code that it is easy to write repeatable and simple unit tests for, and ultimately, reliable applications.

Executing Your First Test

Now that you've created the first test, you want to execute it. Sure, you know it's going to pass, but before you create more complex tests, you should check that you can execute the tests successfully.

Executing Test Runners

The Python application that executes your test code, checks the assertions, and gives you test results in your console is called the **test runner**.

At the bottom of `test.py`, you added this small snippet of code:

Python

```
if __name__ == '__main__':
    unittest.main()
```

This is a command line entry point. It means that if you execute the script alone by running `python test.py` at the command line, it will call `unittest.main()`. This executes the test runner by discovering all classes in this file that inherit from `unittest.TestCase`.

This is one of many ways to execute the `unittest` test runner. When you have a single test file named `test.py`, calling `python test.py` is a great way to get started.

Another way is using the `unittest` command line. Try this:

Shell

```
$ python -m unittest test
```

This will execute the same test module (called `test`) via the command line.

You can provide additional options to change the output. One of those is `-v` for verbose. Try that next:

Shell

```
$ python -m unittest -v test
test_list_int (test.TestSum) ... ok

-----
Ran 1 tests in 0.000s
```

This executed the one test inside `test.py` and printed the results to the console. Verbose mode listed the names of the tests it executed first, along with the result of each test.

Instead of providing the name of a module containing tests, you can request an auto-discovery using the following:

Shell

```
$ python -m unittest discover
```

This will search the current directory for any files named `test*.py` and attempt to test them.

Once you have multiple test files, as long as you follow the `test*.py` naming pattern, you can provide the name of the directory instead by using the `-s` flag and the name of the directory:

Shell

```
$ python -m unittest discover -s tests
```

```
$ python -m unittest discover -s tests
```

unittest will run all tests in a single test plan and give you the results.

Lastly, if your source code is not in the directory root and contained in a subdirectory, for example in a folder called `src/`, you can tell unittest where to execute the tests so that it can import the modules correctly with the `-t` flag:

Shell

```
$ python -m unittest discover -s tests -t src
```

unittest will change to the `src/` directory, scan for all `test*.py` files inside the the `tests` directory, and execute them.

Understanding Test Output

That was a very simple example where everything passes, so now you're going to try a failing test and interpret the output.

`sum()` should be able to accept other lists of numeric types, like fractions.

At the top of the `test.py` file, add an import statement to import the `Fraction` type from the `fractions` module in the standard library:

Python

```
from fractions import Fraction
```

Now add a test with an assertion expecting the incorrect value, in this case expecting the sum of $\frac{1}{4}$, $\frac{1}{4}$, and $\frac{2}{5}$ to be 1:

Python

```
import unittest

from my_sum import sum

class TestSum(unittest.TestCase):
    def test_list_int(self):
        """
        Test that it can sum a list of integers
        """
        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

    def test_list_fraction(self):
        """
        Test that it can sum a list of fractions
        """
        data = [Fraction(1, 4), Fraction(1, 4), Fraction(2, 5)]
        result = sum(data)
        self.assertEqual(result, 1)

if __name__ == '__main__':
    unittest.main()
```

If you execute the tests again with `python -m unittest test`, you should see the following output:

Shell

```
$ python -m unittest test
F.
=====
FAIL: test_list_fraction (test.TestSum)
-----
Traceback (most recent call last):
  File "test.py", line 21, in test_list_fraction
```

```
    self.assertEqual(result, 1)
AssertionError: Fraction(9, 10) != 1

-----
Ran 2 tests in 0.001s

FAILED (failures=1)
```

In the output, you'll see the following information:

1. The first line shows the execution results of all the tests, one failed (F) and one passed (.).
2. The FAIL entry shows some details about the failed test:
 - The test method name (`test_list_fraction`)
 - The test module (`test`) and the test case (`TestSum`)
 - A traceback to the failing line
 - The details of the assertion with the expected result (1) and the actual result (`Fraction(9, 10)`)

Remember, you can add extra information to the test output by adding the `-v` flag to the `python -m unittest` command.

Running Your Tests From PyCharm

If you're using the PyCharm IDE, you can run `unittest` or `pytest` by following these steps:

1. In the Project tool window, select the `tests` directory.
2. On the context menu, choose the run command for `unittest`. For example, choose *Run ‘Unittests in my Tests…’*.

This will execute `unittest` in a test window and give you the results within PyCharm:

More information is available on the [PyCharm Website](#).

Running Your Tests From Visual Studio Code

If you're using the Microsoft Visual Studio Code IDE, support for `unittest`, `nose`, and `pytest` execution is built into the Python plugin.

If you have the Python plugin installed, you can set up the configuration of your tests by opening the Command Palette with `^Ctrl + ↑Shift + P` and typing “Python test”. You will see a range of options:

Choose `Debug All Unit Tests`, and VSCode will then raise a prompt to configure the test framework. Click on the cog to select the test runner (`unittest`) and the home directory `(.)`.

Once this is set up, you will see the status of your tests at the bottom of the window, and you can quickly access the test logs and run the tests again by clicking on these icons:

This shows the tests are executing, but some of them are failing.

Testing for Web Frameworks Like Django and Flask

If you're writing tests for a web application using one of the popular frameworks like Django or Flask, there are some important differences in the way you write and run the tests.

Why They're Different From Other Applications

Think of all the code you're going to be testing in a web application. The routes, views, and models all require lots of imports and knowledge about the frameworks being used.

This is similar to the car test at the beginning of the tutorial: you have to start up the car's computer before you can run a simple test like checking the lights.

Django and Flask both make this easy for you by providing a test framework based on `unittest`. You can continue writing tests in the way you've been learning but execute them slightly differently.

How to Use the Django Test Runner

The Django `startapp` template will have created a `tests.py` file inside your application directory. If you don't have that already, you can create it with the following contents:

Python

```
from django.test import TestCase

class MyTestCase(TestCase):
    # Your test methods
```

The major difference with the examples so far is that you need to inherit from `django.test.TestCase` instead of `unittest.TestCase`. These classes have the same API, but the Django `TestCase` class sets up all the required state to test.

To execute your test suite, instead of using `unittest` at the command line, you use `manage.py test`:

Shell

```
$ python manage.py test
```

If you want multiple test files, replace `tests.py` with a folder called `tests`, insert an empty file inside called `__init__.py`, and create your `test_*.py` files. Django will discover and execute these.

More information is available at the [Django Documentation Website](#).

How to Use `unittest` and Flask

Flask requires that the app be imported and then set in test mode. You can instantiate a test client and use the test client to make requests to any routes in your application.

All of the test client instantiation is done in the `setUp` method of your test case. In the following example, `my_app` is the name of the application. Don't worry if you don't know what `setUp` does. You'll learn about that in the [More Advanced Testing Scenarios](#) section.

The code within your test file should look like this:

Python

```
import my_app
import unittest

class MyTestCase(unittest.TestCase):

    def setUp(self):
        my_app.app.testing = True
        self.app = my_app.app.test_client()

    def test_home(self):
        result = self.app.get('/')
        # Make your assertions
```

You can then execute the test cases using the `python -m unittest discover` command.

More information is available at the [Flask Documentation Website](#).

More Advanced Testing Scenarios

Before you step into creating tests for your application, remember the three basic steps of every test:

1. Create your inputs
2. Execute the code, capturing the output
3. Compare the output with an expected result

It's not always as easy as creating a static value for the input like a string or a number. Sometimes, your application will require an instance of a class or a context. What do you do then?

The data that you create as an input is known as a **fixture**. It's common practice to create fixtures and reuse them.

If you're running the same test and passing different values each time and expecting the same result, this is known as **parameterization**.

Handling Expected Failures

Earlier, when you made a list of scenarios to test `sum()`, a question came up: What happens when you provide it with a bad value, such as a single integer or a string?

In this case, you would expect `sum()` to throw an error. When it does throw an error, that would cause the test to fail.

There's a special way to handle expected errors. You can use `.assertRaises()` as a context-manager, then inside the `with` block execute the test steps:

Python

```
import unittest

from my_sum import sum

class TestSum(unittest.TestCase):
    def test_list_int(self):
        """
        Test that it can sum a list of integers
        """
        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

    def test_list_fraction(self):
        """
        Test that it can sum a list of fractions
        """
```

```

data = [Fraction(1, 4), Fraction(1, 4), Fraction(2, 5)]
result = sum(data)
self.assertEqual(result, 1)

def test_bad_type(self):
    data = "banana"
    with self.assertRaises(TypeError):
        result = sum(data)

if __name__ == '__main__':
    unittest.main()

```

This test case will now only pass if `sum(data)` raises a `TypeError`. You can replace `TypeError` with any exception type you choose.

Isolating Behaviors in Your Application

Earlier in the tutorial, you learned what a side effect is. Side effects make unit testing harder since, each time a test is run, it might give a different result, or even worse, one test could impact the state of the application and cause another test to fail!

There are some simple techniques you can use to test parts of your application that have many side effects:

- Refactoring code to follow the Single Responsibility Principle
- Mocking out any method or function calls to remove side effects
- Using integration testing instead of unit testing for this piece of the application

If you're not familiar with mocking, see [Python CLI Testing](#) for some great examples.

Writing Integration Tests

So far, you've been learning mainly about unit testing. Unit testing is a great way to build predictable and stable code. But at the end of the day, your application needs to work when it starts!

Integration testing is the testing of multiple components of the application to check that they work together. Integration testing might require acting like a consumer or user of the application by:

- Calling an HTTP REST API
- Calling a Python API
- Calling a web service
- Running a command line

Each of these types of integration tests can be written in the same way as a unit test, following the Input, Execute, and Assert pattern. The most significant difference is that integration tests are checking more components at once and therefore will have more side effects than a unit test. Also, integration tests will require more fixtures to be in place, like a database, a network socket, or a configuration file.

This is why it's good practice to separate your unit tests and your integration tests. The creation of fixtures required for an integration like a test database and the test cases themselves often take a lot longer to execute than unit tests, so you may only want to run integration tests before you push to production instead of once on every commit.

A simple way to separate unit and integration tests is simply to put them in different folders:

```
project/
|
└── my_app/
    └── __init__.py
|
└── tests/
    |
    └── unit/
        ├── __init__.py
        └── test_sum.py
    |
    └── integration/
        ├── __init__.py
        └── test_integration.py
```

There are many ways to execute only a select group of tests. The specify source directory flag, -s, can be added to `unittest discover` with the path containing the tests:

Shell

```
$ python -m unittest discover -s tests/integration
```

`unittest` will have given you the results of all the tests within the `tests/integration` directory.

Testing Data-Driven Applications

Many integration tests will require backend data like a database to exist with certain values. For example, you might want to have a test that checks that the application displays correctly with more than 100 customers in the database, or the order page works even if the product names are displayed in Japanese.

These types of integration tests will depend on different test fixtures to make sure they are repeatable and predictable.

A good technique to use is to store the test data in a folder within your integration testing folder called `fixtures` to indicate that it contains test data. Then, within your tests, you can load the data and run the test.

Here's an example of that structure if the data consisted of JSON files:

```
project/
|
└── my_app/
    └── __init__.py
|
└── tests/
    |
    └── unit/
        ├── __init__.py
        └── test_sum.py
    |
    └── integration/
        |
        └── fixtures/
            ├── test_basic.json
            └── test_complex.json
        |
        ├── __init__.py
        └── test_integration.py
```

Within your test case, you can use the `.setup()` method to load the test data from a fixture file in a known path and execute many tests against that test data. Remember you can have multiple test cases in a single Python file, and the `unittest` discovery will execute both. You can have one test case for each set of test data:

Python

```
import unittest

class TestBasic(unittest.TestCase):
    def setUp(self):
        # Load test data
        self.app = App(database='fixtures/test_basic.json')

    def test_customer_count(self):
        self.assertEqual(len(self.app.customers), 100)

    def test_existence_of_customer(self):
        customer = self.app.get_customer(id=10)
        self.assertEqual(customer.name, "Org XYZ")
        self.assertEqual(customer.address, "10 Red Road, Reading")

class TestComplexData(unittest.TestCase):
    def setUp(self):
        # load test data
        self.app = App(database='fixtures/test_complex.json')

    def test_customer_count(self):
        self.assertEqual(len(self.app.customers), 10000)

    def test_existence_of_customer(self):
        customer = self.app.get_customer(id=9999)
        self.assertEqual(customer.name, u"バナナ")
        self.assertEqual(customer.address, "10 Red Road, Akihabara, Tokyo")

if __name__ == '__main__':
    unittest.main()
```

If your application depends on data from a remote location, like a remote API, you'll want to ensure your tests are repeatable. Having your tests fail because the API is offline or there is a connectivity issue could slow down development. In these types of situations, it is best practice to store remote fixtures locally so they can be recalled and sent to the application.

The `requests` library has a complimentary package called `responses` that gives you ways to create response fixtures and save them in your test folders. Find out more [on their GitHub Page](#).

Testing in Multiple Environments

So far, you've been testing against a single version of Python using a virtual environment with a specific set of dependencies. You might want to check that your application works on multiple versions of Python, or multiple versions of a package. `Tox` is an application that automates testing in multiple environments.

Installing Tox

`Tox` is available on PyPI as a package to install via `pip`:

Shell

```
$ pip install tox
```

Now that you have `Tox` installed, it needs to be configured.

Configuring Tox for Your Dependencies

`Tox` is configured via a configuration file in your project directory. The `Tox` configuration file contains the following:

- The command to run in order to execute tests
- Any additional packages required before executing
- The target Python versions to test against

Instead of having to learn the Tox configuration syntax, you can get a head start by running the quickstart application:

Shell

```
$ tox-quickstart
```

The Tox configuration tool will ask you those questions and create a file similar to the following in `tox.ini`:

Config File

```
[tox]
envlist = py27, py36

[testenv]
deps =

commands =
    python -m unittest discover
```

Before you can run Tox, it requires that you have a `setup.py` file in your application folder containing the steps to install your package. If you don't have one, you can follow [this guide](#) on how to create a `setup.py` before you continue.

Alternatively, if your project is not for distribution on PyPI, you can skip this requirement by adding the following line in the `tox.ini` file under the `[tox]` heading:

Config File

```
[tox]
envlist = py27, py36
skipdist=True
```

If you don't create a `setup.py`, and your application has some dependencies from PyPI, you'll need to specify those on a number of lines under the `[testenv]` section. For example, Django would require the following:

Config File

```
[testenv]
deps = django
```

Once you have completed that stage, you're ready to run the tests.

You can now execute Tox, and it will create two virtual environments: one for Python 2.7 and one for Python 3.6. The Tox directory is called `.tox/`. Within the `.tox/` directory, Tox will execute `python -m unittest discover` against each virtual environment.

You can run this process by calling Tox at the command line:

Shell

```
$ tox
```

Tox will output the results of your tests against each environment. The first time it runs, Tox takes a little bit of time to create the virtual environments, but once it has, the second execution will be a lot faster.

Executing Tox

The output of Tox is quite straightforward. It creates an environment for each version, installs your dependencies,

and then runs the test commands.

There are some additional command line options that are great to remember.

Run only a single environment, such as Python 3.6:

Shell

```
$ tox -e py36
```

Recreate the virtual environments, in case your dependencies have changed or [site-packages](#) is corrupt:

Shell

```
$ tox -r
```

Run Tox with less verbose output:

Shell

```
$ tox -q
```

Running Tox with more verbose output:

Shell

```
$ tox -v
```

More information on Tox can be found at the [Tox Documentation Website](#).

Automating the Execution of Your Tests

So far, you have been executing the tests manually by running a command. There are some tools for executing tests automatically when you make changes and commit them to a source-control repository like Git. Automated testing tools are often known as CI/CD tools, which stands for “Continuous Integration/Continuous Deployment.” They can run your tests, compile and publish any applications, and even deploy them into production.

[Travis CI](#) is one of many available CI (Continuous Integration) services available.

Travis CI works nicely with Python, and now that you’ve created all these tests, you can automate the execution of them in the cloud! Travis CI is free for any open-source projects on GitHub and GitLab and is available for a charge for private projects.

To get started, login to the website and authenticate with your GitHub or GitLab credentials. Then create a file called `.travis.yml` with the following contents:

YAML

```
language: python
python:
  - "2.7"
  - "3.7"
install:
  - pip install -r requirements.txt
script:
  - python -m unittest discover
```

This configuration instructs Travis CI to:

1. Test against Python 2.7 and 3.7 (You can replace those versions with any you choose.)
2. Install all the packages you list in `requirements.txt` (You should remove this section if you don’t have any dependencies.)
3. Run `python -m unittest discover` to run the tests

Once you have committed and pushed this file, Travis CI will run these commands every time you push to your remote Git repository. You can check out the results on their website.

What's Next

Now that you've learned how to create tests, execute them, include them in your project, and even execute them automatically, there are a few advanced techniques you might find handy as your test library grows.

Introducing Linters Into Your Application

Tox and Travis CI have configuration for a test command. The test command you have been using throughout this tutorial is `python -m unittest discover`.

You can provide one or many commands in all of these tools, and this option is there to enable you to add more tools that improve the quality of your application.

One such type of application is called a linter. A linter will look at your code and comment on it. It could give you tips about mistakes you've made, correct trailing spaces, and even predict bugs you may have introduced.

For more information on linters, read the [Python Code Quality tutorial](#).

Passive Linting With flake8

A popular linter that comments on the style of your code in relation to the [PEP 8](#) specification is `flake8`.

You can install `flake8` using pip:

Shell

```
$ pip install flake8
```

You can then run `flake8` over a single file, a folder, or a pattern:

Shell

```
$ flake8 test.py
test.py:6:1: E302 expected 2 blank lines, found 1
test.py:23:1: E305 expected 2 blank lines after class or function definition, found 1
test.py:24:20: W292 no newline at end of file
```

You will see a list of errors and warnings for your code that `flake8` has found.

`flake8` is configurable on the command line or inside a configuration file in your project. If you wanted to ignore certain rules, like `E305` shown above, you can set them in the configuration. `flake8` will inspect a `.flake8` file in the project folder or a `setup.cfg` file. If you decided to use Tox, you can put the `flake8` configuration section inside `tox.ini`.

This example ignores the `.git` and `__pycache__` directories as well as the `E305` rule. Also, it sets the max line length to 90 instead of 80 characters. You will likely find that the default constraint of 79 characters for line-width is very limiting for tests, as they contain long method names, string literals with test values, and other pieces of data that can be longer. It is common to set the line length for tests to up to 120 characters:

Config File

```
[flake8]
ignore = E305
exclude = .git,__pycache__
max-line-length = 90
```

Alternatively, you can provide these options on the command line:

Shell

```
$ flake8 --ignore E305 --exclude .git,__pycache__ --max-line-length=90
```

A full list of configuration options is available on the [Documentation Website](#).

You can now add `flake8` to your CI configuration. For Travis CI, this would look as follows:

YAML

```
matrix:  
  include:  
    - python: "2.7"  
      script: "flake8"
```

Travis will read the configuration in `.flake8` and fail the build if any linting errors occur. Be sure to add the `flake8` dependency to your `requirements.txt` file.

Aggressive Linting With a Code Formatter

`flake8` is a passive linter: it recommends changes, but you have to go and change the code. A more aggressive approach is a code formatter. Code formatters will change your code automatically to meet a collection of style and layout practices.

`black` is a very unforgiving formatter. It doesn't have any configuration options, and it has a very specific style. This makes it great as a drop-in tool to put in your test pipeline.

Note: `black` requires Python 3.6+.

You can install `black` via pip:

Shell

```
$ pip install black
```

Then to run `black` at the command line, provide the file or directory you want to format:

Shell

```
$ black test.py
```

Keeping Your Test Code Clean

When writing tests, you may find that you end up copying and pasting code a lot more than you would in regular applications. Tests can be very repetitive at times, but that is by no means a reason to leave your code sloppy and hard to maintain.

Over time, you will develop a lot of [technical debt](#) in your test code, and if you have significant changes to your application that require changes to your tests, it can be a more cumbersome task than necessary because of the way you structured them.

Try to follow the **DRY** principle when writing tests: **Don't Repeat Yourself**.

Test fixtures and functions are a great way to produce test code that is easier to maintain. Also, readability counts. Consider deploying a linting tool like `flake8` over your test code:

Shell

```
$ flake8 --max-line-length=120 tests/
```

Testing for Performance Degradation Between Changes

There are many ways to benchmark code in Python. The standard library provides the `timeit` module, which can time functions a number of times and give you the distribution. This example will execute `test()` 100 times and `print()` the output:

Python

```
def test():
    # ... your code

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test", number=100))
```

Another option, if you decided to use pytest as a test runner, is the `pytest-benchmark` plugin. This provides a pytest fixture called `benchmark`. You can pass `benchmark()` any callable, and it will log the timing of the callable to the results of pytest.

You can install `pytest-benchmark` from PyPI using pip:

Shell

```
$ pip install pytest-benchmark
```

Then, you can add a test that uses the fixture and passes the callable to be executed:

Python

```
def test_my_function(benchmark):
    result = benchmark(test)
```

Execution of pytest will now give you benchmark results:

More information is available at the [Documentation Website](#).

Testing for Security Flaws in Your Application

Another test you will want to run on your application is checking for common security mistakes or vulnerabilities.

You can install bandit from PyPI using pip:

Shell

```
$ pip install bandit
```

You can then pass the name of your application module with the `-r` flag, and it will give you a summary:

Shell

```
$ bandit -r my_sum
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.5.2
Run started:2018-10-08 00:35:02.669550
```

```
Test results:  
    No issues identified.  
  
Code scanned:  
    Total lines of code: 5  
    Total lines skipped (#nosec): 0  
  
Run metrics:  
    Total issues (by severity):  
        Undefined: 0.0  
        Low: 0.0  
        Medium: 0.0  
        High: 0.0  
    Total issues (by confidence):  
        Undefined: 0.0  
        Low: 0.0  
        Medium: 0.0  
        High: 0.0  
Files skipped (0):
```

As with flake8, the rules that bandit flags are configurable, and if there are any you wish to ignore, you can add the following section to your setup.cfg file with the options:

Config File

```
[bandit]  
exclude: /test  
tests: B101,B102,B301
```

More details are available at the [GitHub Website](#).

Conclusion

Python has made testing accessible by building in the commands and libraries you need to validate that your applications work as designed. Getting started with testing in Python needn't be complicated: you can use unittest and write small, maintainable methods to validate your code.

As you learn more about testing and your application grows, you can consider switching to one of the other test frameworks, like pytest, and start to leverage more advanced features.

Thank you for reading. I hope you have a bug-free future with Python!



This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Test-Driven Development With PyTest](#)

About Anthony Shaw

Anthony is an avid Pythonista and writes for Real Python. Anthony is a Fellow of the Python Software Foundation and member of the Open-Source Apache Foundation.

[» More about Anthony](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[David](#)

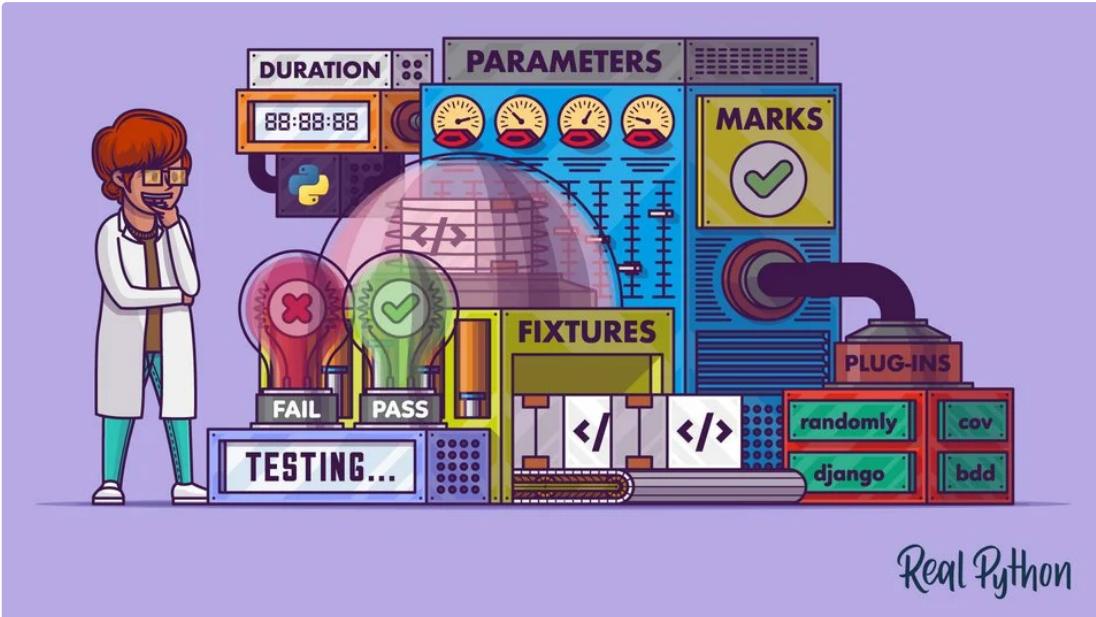
[Geir Arne](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [testing](#)

Recommended Video Course: [Test-Driven Development With PyTest](#)



Real Python

Effective Python Testing With Pytest

by [Dane Hillard](#) ⌚ Apr 20, 2020 💬 7 Comments 🏷️ [intermediate](#) [python](#) [testing](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [How to Install pytest](#)
- [What Makes pytest So Useful?](#)
 - [Less Boilerplate](#)
 - [State and Dependency Management](#)
 - [Test Filtering](#)
 - [Test Parametrization](#)
 - [Plugin-Based Architecture](#)
- [Fixtures: Managing State and Dependencies](#)
 - [When to Create Fixtures](#)
 - [When to Avoid Fixtures](#)
 - [Fixtures at Scale](#)
- [Marks: Categorizing Tests](#)
- [Parametrization: Combining Tests](#)
- [Durations Reports: Fighting Slow Tests](#)
- [Useful pytest Plugins](#)
 - [pytest-randomly](#)
 - [pytest-cov](#)
 - [pytest-django](#)
 - [pytest-bdd](#)
- [Conclusion](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

[Testing your code](#) brings a wide variety of benefits. It increases your confidence that the code behaves as you expect and ensures that changes to your code won't cause regressions. Writing and maintaining tests is hard work, so you should leverage all the tools at your disposal to make it as painless as possible. [pytest](#) is one of the best tools you

can use to boost your testing productivity.

In this tutorial, you'll learn:

IN THIS TUTORIAL, YOU WILL LEARN...

- What **benefits** pytest offers
- How to ensure your tests are **stateless**
- How to make repetitious tests more **comprehensible**
- How to run **subsets** of tests by name or custom groups
- How to create and maintain **reusable** testing utilities

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

How to Install pytest

To follow along with some of the examples in this tutorial, you'll need to install pytest. As with most [Python packages](#), you can install pytest in a [virtual environment](#) from [PyPI](#) using [pip](#):

Shell

```
$ python -m pip install pytest
```

The pytest command will now be available in your installation environment.

What Makes pytest So Useful?

If you've written unit tests for your Python code before, then you may have used Python's built-in `unittest` module. `unittest` provides a solid base on which to build your test suite, but it has a few shortcomings.

A number of third-party testing frameworks attempt to address some of the issues with `unittest`, and [pytest has proven to be one of the most popular](#). pytest is a feature-rich, plugin-based ecosystem for testing your Python code.

If you haven't had the pleasure of using pytest yet, then you're in for a treat! Its philosophy and features will make your testing experience more productive and enjoyable. With pytest, common tasks require less code and advanced tasks can be achieved through a variety of time-saving commands and plugins. It will even run your existing tests out of the box, including those written with `unittest`.

As with most frameworks, some development patterns that make sense when you first start using pytest can start causing pains as your test suite grows. This tutorial will help you understand some of the tools pytest provides to keep your testing efficient and effective even as it scales.

Less Boilerplate

Most functional tests follow the Arrange-Act-Assert model:

1. **Arrange**, or set up, the conditions for the test
2. **Act** by calling some function or method
3. **Assert** that some end condition is true

Testing frameworks typically hook into your test's [assertions](#) so that they can provide information when an assertion fails. `unittest`, for example, provides a number of helpful assertion utilities out of the box. However, even a small set of tests requires a fair amount of [boilerplate code](#).

Imagine you'd like to write a test suite just to make sure `unittest` is working properly in your project. You might want to write one test that always passes and one that always fails:

Python

```
# test_with_unittest.py

from unittest import TestCase

class TryTesting(TestCase):
    def test_always_passes(self):
        pass
```

```
    def test_always_passes(self):
        self.assertTrue(True)

    def test_always_fails(self):
        self.assertTrue(False)
```

You can then run those tests from the command line using the `discover` option of `unittest`:

Shell

```
$ python -m unittest discover
F.
=====
FAIL: test_always_fails (test_with_unittest.TryTesting)
-----
Traceback (most recent call last):
  File "/.../test_with_unittest.py", line 9, in test_always_fails
    self.assertTrue(False)
AssertionError: False is not True

-----
Ran 2 tests in 0.001s

FAILED (failures=1)
```

As expected, one test passed and one failed. You've proven that `unittest` is working, but look at what you had to do:

1. Import the `TestCase` class from `unittest`
2. Create `TryTesting`, a [subclass](#) of `TestCase`
3. Write a method in `TryTesting` for each test
4. Use one of the `self.assert*` methods from `unittest.TestCase` to make assertions

That's a significant amount of code to write, and because it's the minimum you need for *any* test, you'd end up writing the same code over and over. `pytest` simplifies this workflow by allowing you to use Python's `assert` keyword directly:

Python

```
# test_with_pytest.py

def test_always_passes():
    assert True

def test_always_fails():
    assert False
```

That's it. You don't have to deal with any imports or classes. Because you can use the `assert` keyword, you don't need to learn or remember all the different `self.assert*` methods in `unittest`, either. If you can write an expression that you expect to evaluate to `True`, then `pytest` will test it for you. You can run it using the `pytest` command:

Shell

```
$ pytest
===== test session starts =====
platform darwin -- Python 3.7.3, pytest-5.3.0, py-1.8.0, pluggy-0.13.0
rootdir: /.../effective-python-testing-with-pytest
collected 2 items

test_with_pytest.py .F                                [100%]

===== FAILURES =====
____ test_always_fails ____
```

```
def test_always_fails():
>     assert False
E     assert False

test_with_pytest.py:5: AssertionError
===== 1 failed, 1 passed in 0.07s =====
```

pytest presents the test results differently than unittest. The report shows:

1. The system state, including which versions of Python, pytest, and any plugins you have installed
2. The `rootdir`, or the directory to search under for configuration and tests
3. The number of tests the runner discovered

The output then indicates the status of each test using a syntax similar to unittest:

- **A dot (.)** means that the test passed.
- **An F** means that the test has failed.
- **An E** means that the test raised an unexpected exception.

For tests that fail, the report gives a detailed breakdown of the failure. In the example above, the test failed because `assert False` always fails. Finally, the report gives an overall status report of the test suite.

Here are a few more quick assertion examples:

Python

```
def test_uppercase():
    assert "loud noises".upper() == "LOUD NOISES"

def test_reversed():
    assert list(reversed([1, 2, 3, 4])) == [4, 3, 2, 1]

def test_some_primes():
    assert 37 in {
        num
        for num in range(1, 50)
        if num != 1 and not any([num % div == 0 for div in range(2, num)])
    }
```

The learning curve for pytest is shallower than it is for unittest because you don't need to learn new constructs for most tests. Also, the use of `assert`, which you may have used before in your implementation code, makes your tests more understandable.

State and Dependency Management

Your tests will often depend on pieces of data or [test doubles](#) for some of the objects in your code. In unittest, you might extract these dependencies into `setUp()` and `tearDown()` methods so each test in the class can make use of them. But in doing so, you may inadvertently make the test's dependence on a particular piece of data or object entirely **implicit**.

Over time, implicit dependencies can lead to a complex tangle of code that you have to unwind to make sense of your tests. Tests should help you make your code more understandable. If the tests themselves are difficult to understand, then you may be in trouble!

pytest takes a different approach. It leads you toward **explicit** dependency declarations that are still reusable thanks to the availability of [fixtures](#). pytest fixtures are functions that create data or test doubles or initialize some system state for the test suite. Any test that wants to use a fixture must explicitly accept it as an argument, so dependencies are always stated up front.

Fixtures can also make use of other fixtures, again by declaring them explicitly as dependencies. That means that, over time, your fixtures can become bulky and modular. Although the ability to insert fixtures into other fixtures provides enormous flexibility, it can also make managing dependencies more challenging as your test suite grows. Later in this tutorial, you'll learn [more about fixtures](#) and try a few techniques for handling these challenges.

Test Filtering

As your test suite grows, you may find that you want to run just a few tests on a feature and save the full suite for later. pytest provides a few ways of doing this:

- **Name-based filtering:** You can limit pytest to running only those tests whose fully qualified names match a particular expression. You can do this with the `-k` parameter.
- **Directory scoping:** By default, pytest will run only those tests that are in or under the current directory.
- **Test categorization:** pytest can include or exclude tests from particular categories that you define. You can do this with the `-m` parameter.

Test categorization in particular is a subtly powerful tool. pytest enables you to create **marks**, or custom labels, for any test you like. A test may have multiple labels, and you can use them for granular control over which tests to run. Later in this tutorial, you'll see an example of [how pytest marks work](#) and learn how to make use of them in a large test suite.

Test Parametrization

When you're testing functions that process data or perform generic transformations, you'll find yourself writing many similar tests. They may differ only in the input or output of the code being tested. This requires duplicating test code, and doing so can sometimes obscure the behavior you're trying to test.

unittest offers a way of collecting several tests into one, but they don't show up as individual tests in result reports. If one test fails and the rest pass, then the entire group will still return a single failing result. pytest offers its own solution in which each test can pass or fail independently. You'll see [how to parametrize tests](#) with pytest later in this tutorial.

Plugin-Based Architecture

One of the most beautiful features of pytest is its openness to customization and new features. Almost every piece of the program can be cracked open and changed. As a result, pytest users have developed a rich ecosystem of helpful plugins.

Although some pytest plugins focus on specific frameworks like [Django](#), others are applicable to most test suites. You'll see [details on some specific plugins](#) later in this tutorial.

Fixtures: Managing State and Dependencies

pytest fixtures are a way of providing data, test doubles, or state setup to your tests. Fixtures are functions that can return a wide range of values. Each test that depends on a fixture must explicitly accept that fixture as an argument.

When to Create Fixtures

Imagine you're writing a function, `format_data_for_display()`, to process the data returned by an API endpoint. The data represents a list of people, each with a given name, family name, and job title. The function should output a list of strings that include each person's full name (their `given_name` followed by their `family_name`), a colon, and their `title`. To test this, you might write the following code:

```
Python
def format_data_for_display(people):
    ... # Implement this!

def test_format_data_for_display():
    people = [
        {
            "given_name": "Alfonsa",
            "family_name": "Ruiz",
            "title": "Senior Software Engineer",
        },
        {
            "given_name": "Sayid",
            "family_name": "Khan"
        }
    ]
    assert format_data_for_display(people) == [
        "Alfonsa Ruiz: Senior Software Engineer",
        "Sayid Khan"
    ]
```

```

        "given_name": "Sayid",
        "title": "Project Manager",
    ],
]

assert format_data_for_display(people) == [
    "Alfonsa Ruiz: Senior Software Engineer",
    "Sayid Khan: Project Manager",
]

```

Now suppose you need to write another function to transform the data into comma-separated values for use in Excel. The test would look awfully similar:

Python

```

def format_data_for_excel(people):
    ... # Implement this!

def test_format_data_for_excel():
    people = [
        {
            "given_name": "Alfonsa",
            "family_name": "Ruiz",
            "title": "Senior Software Engineer",
        },
        {
            "given_name": "Sayid",
            "family_name": "Khan",
            "title": "Project Manager",
        },
    ]

    assert format_data_for_excel(people) == """given,family,title
Alfonsa,Ruiz,Senior Software Engineer
Sayid,Khan,Project Manager
"""

```

If you find yourself writing several tests that all make use of the same underlying test data, then a fixture may be in your future. You can pull the repeated data into a single function decorated with `@pytest.fixture` to indicate that the function is a pytest fixture:

Python

```

import pytest

@pytest.fixture
def example_people_data():
    return [
        {
            "given_name": "Alfonsa",
            "family_name": "Ruiz",
            "title": "Senior Software Engineer",
        },
        {
            "given_name": "Sayid",
            "family_name": "Khan",
            "title": "Project Manager",
        },
    ]

```

You can use the fixture by adding it as an argument to your tests. Its value will be the return value of the fixture function:

Python

```

def test_format_data_for_display(example_people_data):
    assert format_data_for_display(example_people_data) == [
        "Alfonsa Ruiz: Senior Software Engineer",
        "Sayid Khan: Project Manager",
    ]

def test_format_data_for_excel(example_people_data):
    assert format_data_for_excel(example_people_data) == """given,family,title
Alfonsa,Ruiz,Senior Software Engineer
Sayid,Khan,Project Manager
"""

```

Each test is now notably shorter but still has a clear path back to the data it depends on. Be sure to name your fixture something specific. That way, you can quickly determine if you want to use it when writing new tests in the future!

When to Avoid Fixtures

Fixtures are great for extracting data or objects that you use across multiple tests. They aren't always as good for tests that require slight variations in the data. Littering your test suite with fixtures is no better than littering it with plain data or objects. It might even be worse because of the added layer of indirection.

As with most abstractions, it takes some practice and thought to find the right level of fixture use.

Fixtures at Scale

As you extract more fixtures from your tests, you might see that some fixtures could benefit from further extraction. Fixtures are **modular**, so they can depend on other fixtures. You may find that fixtures in two separate test modules share a common dependency. What can you do in this case?

You can move fixtures from test [modules](#) into more general fixture-related modules. That way, you can import them back into any test modules that need them. This is a good approach when you find yourself using a fixture repeatedly throughout your project.

pytest looks for `conftest.py` modules throughout the directory structure. Each `conftest.py` provides configuration for the file tree pytest finds it in. You can use any fixtures that are defined in a particular `conftest.py` throughout the file's parent directory and in any subdirectories. This is a great place to put your most widely used fixtures.

Another interesting use case for fixtures is in guarding access to resources. Imagine that you've written a test suite for code that deals with [API calls](#). You want to ensure that the test suite doesn't make any real network calls, even if a test accidentally executes the real network call code. pytest provides a [monkeypatch](#) fixture to replace values and behaviors, which you can use to great effect:

Python

```

# conftest.py

import pytest
import requests

@pytest.fixture(autouse=True)
def disable_network_calls(monkeypatch):
    def stunted_get():
        raise RuntimeError("Network access not allowed during testing!")
    monkeypatch.setattr(requests, "get", lambda *args, **kwargs: stunted_get())

```

By placing `disable_network_calls()` in `conftest.py` and adding the `autouse=True` option, you ensure that network calls will be disabled in every test across the suite. Any test that executes code calling `requests.get()` will raise a `RuntimeError` indicating that an unexpected network call would have occurred.

Marks: Categorizing Tests

In any large test suite, some of the tests will inevitably be slow. They might test timeout behavior, for example, or

they might exercise a broad area of the code. Whatever the reason, it would be nice to avoid running *all* the slow tests when you're trying to iterate quickly on a new feature.

pytest enables you to define categories for your tests and provides options for including or excluding categories when you run your suite. You can mark a test with any number of categories.

Marking tests is useful for categorizing tests by subsystem or dependencies. If some of your tests require access to a database, for example, then you could create a `@pytest.mark.database_access` mark for them.

Pro tip: Because you can give your marks any name you want, it can be easy to mistype or misremember the name of a mark. pytest will warn you about marks that it doesn't recognize.

The `--strict-markers` flag to the pytest command ensures that all marks in your tests are registered in your pytest configuration. It will prevent you from running your tests until you register any unknown marks.

For more information on registering marks, check out the [pytest documentation](#).

When the time comes to run your tests, you can still run them all by default with the `pytest` command. If you'd like to run only those tests that require database access, then you can use `pytest -m database_access`. To run all tests *except* those that require database access, you can use `pytest -m "not database_access"`. You can even use an `autouse` fixture to limit database access to those tests marked with `database_access`.

Some plugins expand on the functionality of marks by guarding access to resources. The [pytest-django](#) plugin provides a `django_db` mark. Any tests without this mark that try to access the database will fail. The first test that tries to access the database will trigger the creation of Django's test database.

The requirement that you add the `django_db` mark nudges you toward stating your dependencies explicitly. That's the pytest philosophy, after all! It also means that you can run tests that don't rely on the database much more quickly, because `pytest -m "not django_db"` will prevent the test from triggering database creation. The time savings really add up, especially if you're diligent about running your tests frequently.

pytest provides a few marks out of the box:

- `skip` skips a test unconditionally.
- `skipif` skips a test if the expression passed to it evaluates to True.
- `xfail` indicates that a test is expected to fail, so if the test *does* fail, the overall suite can still result in a passing status.
- `parametrize` (note the spelling) creates multiple variants of a test with different values as arguments. You'll learn more about this mark shortly.

You can see a list of all the marks pytest knows about by running `pytest --markers`.

Parametrization: Combining Tests

You saw earlier in this tutorial how pytest fixtures can be used to reduce code duplication by extracting common dependencies. Fixtures aren't quite as useful when you have several tests with slightly different inputs and expected outputs. In these cases, you can `parametrize` a single test definition, and pytest will create variants of the test for you with the parameters you specify.

Imagine you've written a function to tell if a string is a [palindrome](#). An initial set of tests could look like this:

Python

```
def test_is_palindrome_empty_string():
    assert is_palindrome("")

def test_is_palindrome_single_character():
    assert is_palindrome("a")

def test_is_palindrome_mixed_casing():
    assert is_palindrome("Bob")

def test_is_palindrome_with_spaces():
    assert is_palindrome("Never odd or even")
```

```

def test_is_palindrome_with_punctuation():
    assert is_palindrome("Do geese see God?")

def test_is_palindrome_not_palindrome():
    assert not is_palindrome("abc")

def test_is_palindrome_not Quite():
    assert not is_palindrome("abab")

```

All of these tests except the last two have the same shape:

Python

```

def test_is_palindrome_<in some situation>():
    assert is_palindrome("<some string>")

```

You can use `@pytest.mark.parametrize()` to fill in this shape with different values, reducing your test code significantly:

Python

```

@pytest.mark.parametrize("palindrome", [
    "",
    "a",
    "Bob",
    "Never odd or even",
    "Do geese see God?",
])
def test_is_palindrome(palindrome):
    assert is_palindrome(palindrome)

@pytest.mark.parametrize("non_palindrome", [
    "abc",
    "abab",
])
def test_is_palindrome_not_palindrome(non_palindrome):
    assert not is_palindrome(non_palindrome)

```

The first argument to `parametrize()` is a comma-delimited string of parameter names. The second argument is a [list](#) of either [tuples](#) or single values that represent the parameter value(s). You could take your parametrization a step further to combine all your tests into one:

Python

```

@pytest.mark.parametrize("maybe_palindrome, expected_result", [
    ("", True),
    ("a", True),
    ("Bob", True),
    ("Never odd or even", True),
    ("Do geese see God?", True),
    ("abc", False),
    ("abab", False),
])
def test_is_palindrome(maybe_palindrome, expected_result):
    assert is_palindrome(maybe_palindrome) == expected_result

```

Even though this shortened your code, it's important to note that in this case, it didn't do much to clarify your test code. Use parametrization to separate the test data from the test behavior so that it's clear what the test is testing!

Durations Reports: Fighting Slow Tests

Each time you switch contexts from implementation code to test code, you incur some [overhead](#). If your tests are slow to begin with, then overhead can cause friction and frustration.

You read earlier about using marks to filter out slow tests when you run your suite. If you want to improve the speed of your tests, then it's useful to know which tests might offer the biggest improvements. pytest can automatically record test durations for you and report the top offenders.

Use the `--durations` option to the pytest command to include a duration report in your test results. `--durations` expects an integer value `n` and will report the slowest `n` number of tests. The output will follow your test results:

Shell

```
$ pytest --durations=3
3.03s call    test_code.py::test_request_read_timeout
1.07s call    test_code.py::test_request_connection_timeout
0.57s call    test_code.py::test_database_read
=====
===== 7 passed in 10.06s =====
```

Each test that shows up in the durations report is a good candidate to speed up because it takes an above-average amount of the total testing time.

Be aware that some tests may have an invisible setup overhead. You read earlier about how the first test marked with `django_db` will trigger the creation of the Django test database. The durations report reflects the time it takes to set up the database in the test that triggered the database creation, which can be misleading.

Useful pytest Plugins

You learned about a few valuable pytest plugins earlier in this tutorial. You can explore those and a few others in more depth below.

pytest-randomly

[pytest-randomly](#) does something seemingly simple but with valuable effect: It forces your tests to run in a random order. pytest always collects all the tests it can find before running them, so `pytest-randomly` shuffles that list of tests just before execution.

This is a great way to uncover tests that depend on running in a specific order, which means they have a **stateful dependency** on some other test. If you built your test suite from scratch in pytest, then this isn't very likely. It's more likely to happen in test suites that you migrate to pytest.

The plugin will print a seed value in the configuration description. You can use that value to run the tests in the same order as you try to fix the issue.

pytest-cov

If you measure how well your tests cover your implementation code, you likely use the [coverage package](#). [pytest-cov](#) integrates coverage, so you can run `pytest --cov` to see the test coverage report.

pytest-django

[pytest-django](#) provides a handful of useful fixtures and marks for dealing with Django tests. You saw the `django_db` mark earlier in this tutorial, and the `rf` fixture provides direct access to an instance of Django's [RequestFactory](#). The `settings` fixture provides a quick way to set or override Django settings. This is a great boost to your Django testing productivity!

If you're interested in learning more about using pytest with Django, then check out [How to Provide Test Fixtures for Django Models in Pytest](#).

pytest -bdd

pytest can be used to run tests that fall outside the traditional scope of unit testing. [Behavior-driven development](#) (BDD) encourages writing plain-language descriptions of likely user actions and expectations, which you can then use to determine whether to implement a given feature. [pytest-bdd](#) helps you use [Gherkin](#) to write feature tests for your code.

You can see which other plugins are available for pytest with this extensive [list of third-party plugins](#).

Conclusion

pytest offers a core set of productivity features to filter and optimize your tests along with a flexible plugin system that extends its value even further. Whether you have a huge legacy unittest suite or you're starting a new project from scratch, pytest has something to offer you.

In this tutorial, you learned how to use:

- **Fixtures** for handling test dependencies, state, and reusable functionality
- **Marks** for categorizing tests and limiting access to external resources
- **Parametrization** for reducing duplicated code between tests
- **Durations** to identify your slowest tests
- **Plugins** for integrating with other frameworks and testing tools

Install pytest and give it a try. You'll be glad you did. Happy testing!

About Dane Hillard

Dane is a Lead Web Application Developer at ITHAKA and author of Practices of the Python Pro.

[» More about Dane](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Geir Arne](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#) [testing](#)



Python Code Quality: Tools & Best Practices

by Alexander VanTol 8 Comments best-practices python tools

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What is Code Quality?](#)
- [Why Does Code Quality Matter?](#)
 - [It does not do what it is supposed to do](#)
 - [It does contain defects and problems](#)
 - [It is difficult to read, maintain, or extend](#)
- [How to Improve Python Code Quality.](#)
 - [Style Guides](#)
 - [Linters](#)
- [When Can I Check My Code Quality?](#)
 - [As You Write](#)
 - [Before You Check In Code](#)
 - [When Running Tests](#)
- [Conclusion](#)



In this article, we'll identify high-quality Python code and show you how to improve the quality of your own code.

We'll analyze and compare tools you can use to take your code to the next level. Whether you've been using Python for a while, or just beginning, you can benefit from the practices and tools talked about here.

What is Code Quality?

Of course you want quality code, who wouldn't? But to improve code quality, we have to define what it is.

A quick Google search yields many results defining code quality. As it turns out, the term can mean many different things to people.

One way of trying to define code quality is to look at one end of the spectrum: high-quality code. Hopefully, you can agree on the following high-quality code identifiers:

- It does what it is supposed to do.
- It does not contain defects or problems.
- It is easy to read, maintain, and extend.

These three identifiers, while simplistic, seem to be generally agreed upon. In an effort to expand these ideas further, let's delve into why each one matters in the realm of software.

Why Does Code Quality Matter?

To determine why high-quality code is important, let's revisit those identifiers. We'll see what happens when code doesn't meet them.

It does **not** do what it is supposed to do

Meeting requirements is the basis of any product, software or otherwise. We make software to do something. If in the end, it doesn't do it... well it's definitely not high quality. If it doesn't meet basic requirements, it's hard to even call it low quality.

It **does** contain defects and problems

If something you're using has issues or causes you problems, you probably wouldn't call it high-quality. In fact, if it's bad enough, you may stop using it altogether.

For the sake of not using software as an example, let's say your vacuum works great on regular carpet. It cleans up all the dust and cat hair. One fateful night the cat knocks over a plant, spilling dirt everywhere. When you try to use the vacuum to clean the pile of dirt, it breaks, spewing the dirt everywhere.

While the vacuum worked under some circumstances, it didn't efficiently handle the occasional extra load. Thus, you wouldn't call it a high-quality vacuum cleaner.

That is a problem we want to avoid in our code. If things break on edge cases and defects cause unwanted behavior, we don't have a high-quality product.

It is **difficult** to read, maintain, or extend

Imagine this: a customer requests a new feature. The person who wrote the original code is gone. The person who has replaced them now has to make sense of the code that's already there. That person is you.

If the code is easy to comprehend, you'll be able to analyze the problem and come up with a solution much quicker. If the code is complex and convoluted, you'll probably take longer and possibly make some wrong assumptions.

It's also nice if it's easy to add the new feature without disrupting previous features. If the code is *not* easy to extend, your new feature could break other things.

No one *wants* to be in the position where they have to read, maintain, or extend low-quality code. It means more headaches and more work for everyone.

It's bad enough that you have to deal with low-quality code, but don't put someone else in the same situation. You can improve the quality of code that you write.

If you work with a team of developers, you can start putting into place methods to ensure better overall code quality. Assuming that you have their support, of course. You may have to win some people over (feel free to send them this article 😊).

How to Improve Python Code Quality

There are a few things to consider on our journey for high-quality code. First, this journey is not one of pure objectivity. There are some strong feelings of what high-quality code looks like.

While everyone can hopefully agree on the identifiers mentioned above, the way they get achieved is a subjective road. The most opinionated topics usually come up when you talk about achieving readability, maintenance, and extensibility.

So keep in mind that while this article will try to stay objective throughout, there is a very-opinionated world out there when it comes to code.

So, let's start with the most opinionated topic: code style.

Style Guides

Ah, yes. The age-old question: [spaces or tabs?](#)

Regardless of your personal view on how to represent whitespace, it's safe to assume that you at least want consistency in code.

A style guide serves the purpose of defining a consistent way to write your code. Typically this is all cosmetic, meaning it doesn't change the logical outcome of the code. Although, some stylistic choices do avoid common logical mistakes.

Style guides serve to help facilitate the goal of making code easy to read, maintain, and extend.

As far as Python goes, there is a well-accepted standard. It was written, in part, by the author of the Python programming language itself.

[PEP 8](#) provides coding conventions for Python code. It is fairly common for Python code to follow this style guide. It's a great place to start since it's already well-defined.

A sister Python Enhancement Proposal, [PEP 257](#) describes conventions for Python's docstrings, which are strings intended to document modules, classes, functions, and methods. As an added bonus, if docstrings are consistent, there are tools capable of generating documentation directly from the code.

All these guides do is *define* a way to style code. But how do you enforce it? And what about defects and problems in the code, how can you detect those? That's where linters come in.

Linters

What is a Linter?

First, let's talk about lint. Those tiny, annoying little defects that somehow get all over your clothes. Clothes look and feel much better without all that lint. Your code is no different. Little mistakes, stylistic inconsistencies, and dangerous logic don't make your code feel great.

But we all make mistakes. You can't expect yourself to always catch them in time. Mistyped variable names, forgetting a closing bracket, incorrect tabbing in Python, calling a function with the wrong number of arguments, the list goes on and on. Linters help to identify those problem areas.

Additionally, [most editors and IDEs](#) have the ability to run linters in the background as you type. This results in an environment capable of highlighting, underlining, or otherwise identifying problem areas in the code before you run it. It is like an advanced spell-check for code. It underlines issues in squiggly red lines much like your favorite word processor does.

Linters analyze code to detect various categories of lint. Those categories can be broadly defined as the following:

1. Logical Lint
 - Code errors
 - Code with potentially unintended results
 - Dangerous code patterns
2. Stylistic Lint
 - Code not conforming to defined conventions

There are also code analysis tools that provide other insights into your code. While maybe not linters by definition, these tools are usually used side-by-side with linters. They too hope to improve the quality of the code.

Finally, there are tools that automatically format code to some specification. These automated tools ensure that our inferior human minds don't mess up conventions.

What Are My Linter Options For Python?

Before delving into your options, it's important to recognize that some "linters" are just multiple linters packaged nicely together. Some popular examples of those combo-linters are the following:

Flake8: Capable of detecting both logical and stylistic lint. It adds the style and complexity checks of pycodestyle to the logical lint detection of PyFlakes. It combines the following linters:

- PyFlakes
- pycodestyle (formerly pep8)
- McCabe

Pylama: A code audit tool composed of a large number of linters and other tools for analyzing code. It combines the following:

- pycodestyle (formerly pep8)
- pydocstyle (formerly pep257)
- PyFlakes
- McCabe
- Pylint
- Radon
- gjslint

Here are some stand-alone linters categorized with brief descriptions:

Linter	Category	Description
Pylint	Logical & Stylistic	Checks for errors, tries to enforce a coding standard, looks for code smells
PyFlakes	Logical	Analyzes programs and detects various errors
pycodestyle	Stylistic	Checks against some of the style conventions in PEP 8
pydocstyle	Stylistic	Checks compliance with Python docstring conventions
Bandit	Logical	Analyzes code to find common security issues
MyPy	Logical	Checks for optionally-enforced static types

And here are some code analysis and formatting tools:

Tool	Category	Description
McCabe	Analytical	Checks McCabe complexity
Radon	Analytical	Analyzes code for various metrics (lines of code, complexity, and so on)
Black	Formatter	Formats Python code without compromise
Isort	Formatter	Formats imports by sorting alphabetically and separating into sections

Comparing Python Linters

Let's get a better idea of what different linters are capable of catching and what the output looks like. To do this, I ran the same code through a handful of different linters with the default settings.

The code I ran through the linters is below. It contains various logical and stylistic issues:

The comparison below shows the linters I used and their runtime for analyzing the above file. I should point out that these aren't all entirely comparable as they serve different purposes. PyFlakes, for example, does not identify stylistic errors like Pylint does.

Linter	Command	Time
Pylint	pylint code_with_lint.py	1.16s
PyFlakes	pyflakes code_with_lint.py	0.15s
pycodestyle	pycodestyle code_with_lint.py	0.14s
pydocstyle	pydocstyle code_with_lint.py	0.21s

For the outputs of each, see the sections below.

Pylint

Pylint is one of the oldest linters (circa 2006) and is still well-maintained. Some might call this software battle-hardened. It's been around long enough that contributors have fixed most major bugs and the core features are well-developed.

The common complaints against Pylint are that it is slow, too verbose by default, and takes a lot of configuration to get it working the way you want. Slowness aside, the other complaints are somewhat of a double-edged sword. Verbosity can be because of thoroughness. Lots of configuration can mean lots of adaptability to your preferences.

Without further ado, the output after running Pylint against the lint-filled code from above:

Text

```
No config file found, using default configuration
*****
Module code_with_lint
W: 23, 0: Unnecessary semicolon (unnecessary-semicolon)
C: 27, 0: Unnecessary parens after 'return' keyword (superfluous-parens)
C: 27, 0: No space allowed after bracket
    return( 'an unlucky number!')
           ^
           (bad-whitespace)
C: 29, 0: Unnecessary parens after 'return' keyword (superfluous-parens)
C: 33, 0: Exactly one space required after comma
    def __init__(self, some_arg,  some_other_arg, verbose = False):
           ^
           (bad-whitespace)
C: 33, 0: No space allowed around keyword argument assignment
    def __init__(self, some_arg,  some_other_arg, verbose = False):
           ^
           (bad-whitespace)
C: 34, 0: Exactly one space required around assignment
    self.some_other_arg  =  some_other_arg
           ^
           (bad-whitespace)
C: 35, 0: Exactly one space required around assignment
    self.some_arg  =  some_arg
           ^
           (bad-whitespace)
```

```

    sc11.some_var      = some_var
                           ^ (bad-whitespace)
C: 40, 0: Final newline missing (missing-final-newline)
W:  6, 0: Redefining built-in 'pow' (redefined-builtin)
W:  6, 0: Wildcard import math ( wildcard-import)
C: 11, 0: Constant name "some_global_var" doesn't conform to UPPER_CASE naming style (invalid-name)
C: 13, 0: Argument name "x" doesn't conform to snake_case naming style (invalid-name)
C: 13, 0: Argument name "y" doesn't conform to snake_case naming style (invalid-name)
C: 13, 0: Missing function docstring (missing-docstring)
W: 14, 4: Redefining name 'some_global_var' from outer scope (line 11) (redefined-outer-name)
W: 17, 4: Unreachable code (unreachable)
W: 14, 4: Unused variable 'some_global_var' (unused-variable)
...
R: 24,12: Unnecessary "else" after "return" (no-else-return)
R: 20, 0: Either all return statements in a function should return an expression, or none of them sh
C: 31, 0: Missing class docstring (missing-docstring)
W: 37, 8: Redefining name 'time' from outer scope (line 9) (redefined-outer-name)
E: 37,15: Using variable 'time' before assignment (used-before-assignment)
W: 33,50: Unused argument 'verbose' (unused-argument)
W: 36, 8: Unused variable 'list_comprehension' (unused-variable)
W: 39, 8: Unused variable 'date_and_time' (unused-variable)
R: 31, 0: Too few public methods (0/2) (too-few-public-methods)
W:  5, 0: Unused import io (unused-import)
W:  6, 0: Unused import acos from wildcard import (unused-wildcard-import)
...
W:  9, 0: Unused time imported from time (unused-import)

```

Note that I've condensed this with ellipses for similar lines. It's quite a bit to take in, but there *is* a lot of lint in this code.

Note that Pylint prefixes each of the problem areas with a R, C, W, E, or F, meaning:

- [R]efactor for a “good practice” metric violation
- [C]onvention for coding standard violation
- [W]arning for stylistic problems, or minor programming issues
- [E]rror for important programming issues (i.e. most probably bug)
- [F]atal for errors which prevented further processing

The above list is directly from Pylint's [user guide](#).

PyFlakes

Pyflakes “makes a simple promise: it will never complain about style, and it will try very, very hard to never emit false positives”. This means that Pyflakes won’t tell you about missing docstrings or argument names not conforming to a naming style. It focuses on logical code issues and potential errors.

The benefit here is speed. PyFlakes runs in a fraction of the time Pylint takes.

Output after running against lint-filled code from above:

```

Text
code_with_lint.py:5: 'io' imported but unused
code_with_lint.py:6: 'from math import *' used; unable to detect undefined names
code_with_lint.py:14: local variable 'some_global_var' is assigned to but never used
code_with_lint.py:36: 'pi' may be undefined, or defined from star imports: math
code_with_lint.py:36: local variable 'list_comprehension' is assigned to but never used
code_with_lint.py:37: local variable 'time' (defined in enclosing scope on line 9) referenced before
code_with_lint.py:37: local variable 'time' is assigned to but never used
code_with_lint.py:39: local variable 'date_and_time' is assigned to but never used

```

The downside here is that parsing this output may be a bit more difficult. The various issues and errors are not labeled or organized by type. Depending on how you use this, that may not be a problem at all.

pycodestyle (formerly pep8)

Used to check *some* style conventions from [PEP8](#). Naming conventions are not checked and neither are docstrings. The errors and warnings it does catch are categorized in [this table](#).

Output after running against lint-filled code from above:

Text

```
code_with_lint.py:13:1: E302 expected 2 blank lines, found 1
code_with_lint.py:15:15: E225 missing whitespace around operator
code_with_lint.py:20:1: E302 expected 2 blank lines, found 1
code_with_lint.py:21:10: E711 comparison to None should be 'if cond is not None:'
code_with_lint.py:23:25: E703 statement ends with a semicolon
code_with_lint.py:27:24: E201 whitespace after '('
code_with_lint.py:31:1: E302 expected 2 blank lines, found 1
code_with_lint.py:33:58: E251 unexpected spaces around keyword / parameter equals
code_with_lint.py:33:60: E251 unexpected spaces around keyword / parameter equals
code_with_lint.py:34:28: E221 multiple spaces before operator
code_with_lint.py:34:31: E222 multiple spaces after operator
code_with_lint.py:35:22: E221 multiple spaces before operator
code_with_lint.py:35:31: E222 multiple spaces after operator
code_with_lint.py:36:80: E501 line too long (83 > 79 characters)
code_with_lint.py:40:15: W292 no newline at end of file
```

The nice thing about this output is that the lint is labeled by category. You can choose to ignore certain errors if you don't care to adhere to a specific convention as well.

pydocstyle (formerly pep257)

Very similar to pycodestyle, except instead of checking against PEP8 code style conventions, it checks docstrings against conventions from [PEP257](#).

Output after running against lint-filled code from above:

Text

```
code_with_lint.py:1 at module level:
    D200: One-line docstring should fit on one line with quotes (found 3)
code_with_lint.py:1 at module level:
    D400: First line should end with a period (not '!')
code_with_lint.py:13 in public function `multiply`:
    D103: Missing docstring in public function
code_with_lint.py:20 in public function `is_sum_lucky`:
    D103: Missing docstring in public function
code_with_lint.py:31 in public class `SomeClass`:
    D101: Missing docstring in public class
code_with_lint.py:33 in public method `__init__`:
    D107: Missing docstring in __init__
```

Again, like pycodestyle, pydocstyle labels and categorizes the various errors it finds. And the list doesn't conflict with anything from pycodestyle since all the errors are prefixed with a D for docstring. A list of those errors can be found [here](#).

Code Without Lint

You can adjust the previously lint-filled code based on the linter's output and you'll end up with something like the following:

Python Code Without Lint

Show/Hide

That code is lint-free according to the linters above. While the logic itself is mostly nonsensical, you can see that at a minimum, consistency is enforced.

In the above case, we ran linters after writing all the code. However, that's not the only way to go about checking code quality.

When Can I Check My Code Quality?

You can check your code's quality:

- As you write it
- When it's checked in

- When you're running your tests

It's useful to have linters run against your code frequently. If automation and consistency aren't there, it's easy for a large team or project to lose sight of the goal and start creating lower quality code. It happens slowly, of course. Some poorly written logic or maybe some code with formatting that doesn't match the neighboring code. Over time, all that lint piles up. Eventually, you can get stuck with something that's buggy, hard to read, hard to fix, and a pain to maintain.

To avoid that, check code quality often!

As You Write

You can use linters as you write code, but configuring your environment to do so may take some extra work. It's generally a matter of finding the plugin for your IDE or editor of choice. In fact, most IDEs will already have linters built in.

Here's some general info on Python linting for various editors:

- [Sublime Text](#)
- [VS Code](#)
- [Atom](#)
- [Vim](#)
- [Emacs](#)

Before You Check In Code

If you're using Git, Git hooks can be set up to run your linters before committing. Other version control systems have similar methods to run scripts before or after some action in the system. You can use these methods to block any new code that doesn't meet quality standards.

While this may seem drastic, forcing every bit of code through a screening for lint is an important step towards ensuring continued quality. Automating that screening at the front gate to your code may be the best way to avoid lint-filled code.

When Running Tests

You can also place linters directly into whatever system you may use for [continuous integration](#). The linters can be set up to fail the build if the code doesn't meet quality standards.

Again, this may seem like a drastic step, especially if there are already lots of linter errors in the existing code. To combat this, some continuous integration systems will allow you the option of only failing the build if the new code increases the number of linter errors that were already present. That way you can start improving quality without doing a whole rewrite of your existing code base.

Conclusion

High-quality code does what it's supposed to do without breaking. It is easy to read, maintain, and extend. It functions without problems or defects and is written so that it's easy for the next person to work with.

Hopefully it goes without saying that you should strive to have such high-quality code. Luckily, there are methods and tools to help improve code quality.

Style guides will bring consistency to your code. [PEP8](#) is a great starting point for Python. Linters will help you identify problem areas and inconsistencies. You can use linters throughout the development process, even automating them to flag lint-filled code before it gets too far.

Having linters complain about style also avoids the need for style discussions during code reviews. Some people may find it easier to receive candid feedback from these tools instead of a team member. Additionally, some team members may not want to "nitpick" style during code reviews. Linters avoid the politics, save time, and complain about any inconsistency.

In addition, all the linters mentioned in this article have various command line options and configurations that let

you tailor the tool to your liking. You can be as strict or as loose as you want, which is an important thing to realize.

Improving code quality is a process. You can take steps towards improving it without completely disallowing all nonconformant code. Awareness is a great first step. It just takes a person, like you, to first realize how important high-quality code is.

About Alexander VanTol

Alexander is an avid Pythonista who spends his time on various creative projects involving programming, music, and creative writing.

[» More about Alexander](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Adriana](#)

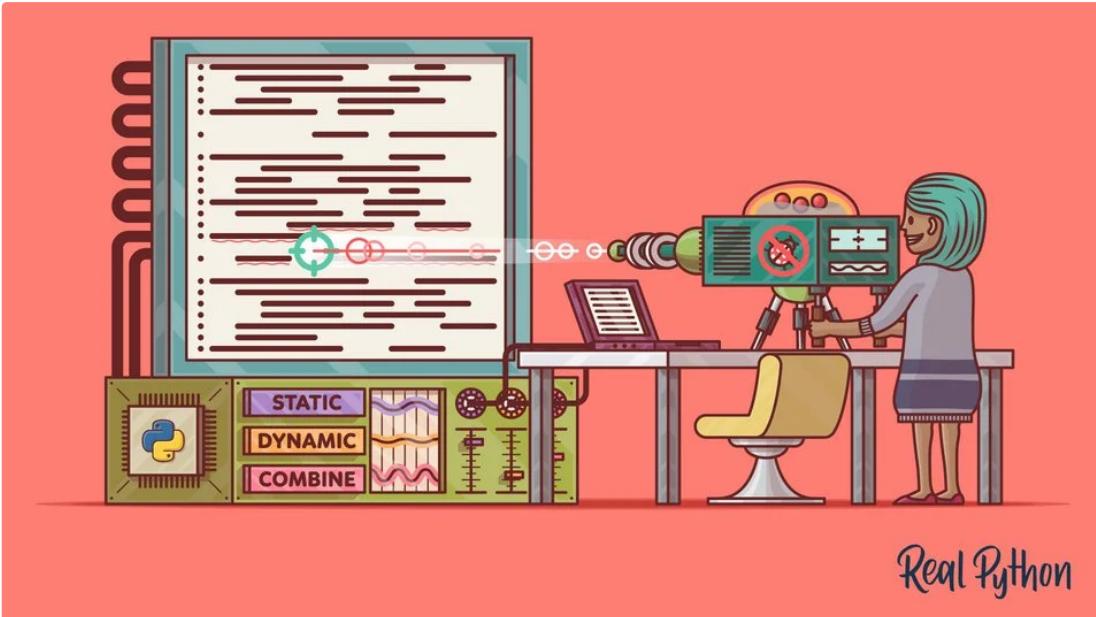
[Dan](#)

[Jim](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [python](#) [tools](#)



Python Type Checking (Guide)

by [Geir Arne Hjelle](#) 29 Comments [best-practices](#) [intermediate](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Type Systems](#)
 - [Dynamic Typing](#)
 - [Static Typing](#)
 - [Duck Typing](#)
- [Hello Types](#)
- [Pros and Cons](#)
- [Annotations](#)
 - [Function Annotations](#)
 - [Variable Annotations](#)
 - [Type Comments](#)
 - [So, Type Annotations or Type Comments?](#)
- [Playing With Python Types, Part 1](#)
 - [Example: A Deck of Cards](#)
 - [Sequences and Mappings](#)
 - [Type Aliases](#)
 - [Functions Without Return Values](#)
 - [Example: Play Some Cards](#)
 - [The Any Type](#)
- [Type Theory](#)
 - [Subtypes](#)
 - [Covariant, Contravariant, and Invariant](#)
 - [Gradual Typing and Consistent Types](#)
- [Playing With Python Types, Part 2](#)
 - [Type Variables](#)
 - [Duck Types and Protocols](#)
 - [The Optional Type](#)
 - [Example: The Object\(ive\) of the Game](#)
 - [Type Hints for Methods](#)
 - [Classes as Types](#)
 - [Returning self or cls](#)
 - [Annotating *args and **kwargs](#)

- [Callables](#)
- [Example: Hearts](#)
- [Static Type Checking](#)
 - [The Mypy Project](#)
 - [Running Mypy](#)
 - [Adding Stubs](#)
 - [Typeshed](#)
 - [Other Static Type Checkers](#)
 - [Using Types at Runtime](#)
- [Conclusion](#)



[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Type Checking](#)

In this guide, you will get a look into Python type checking. Traditionally, types have been handled by the Python interpreter in a flexible but implicit way. Recent versions of Python allow you to specify explicit type hints that can be used by different tools to help you develop your code more efficiently.

In this tutorial, you'll learn about the following:

- Type annotations and type hints
- Adding static types to code, both your code and the code of others
- Running a static type checker
- Enforcing types at runtime

This is a comprehensive guide that will cover a lot of ground. If you want to just get a quick glimpse of how type hints work in Python, and see whether type checking is something you would include in your code, you don't need to read all of it. The two sections [Hello Types](#) and [Pros and Cons](#) will give you a taste of how type checking works and recommendations about when it'll be useful.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Type Systems

All programming languages include some kind of [type system](#) that formalizes which categories of objects it can work with and how those categories are treated. For instance, a type system can define a numerical type, with 42 as one example of an object of numerical type.

Dynamic Typing

Python is a dynamically typed language. This means that the Python interpreter does type checking only as code runs, and that the type of a variable is allowed to change over its lifetime. The following dummy examples demonstrate that Python has dynamic typing:

```
Python >>> if False:
...     1 + "two" # This line never runs, so no TypeError is raised
... else:
...     1 + 2
...
```

```
>>> 1 + "two" # Now this is type checked, and a TypeError is raised
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the first example, the branch `1 + "two"` never runs so it's never type checked. The second example shows that when `1 + "two"` is evaluated it raises a `TypeError` since you can't add an integer and a string in Python.

Next, let's see if variables can change type:

```
Python >>>
>>> thing = "Hello"
>>> type(thing)
<class 'str'>

>>> thing = 28.1
>>> type(thing)
<class 'float'>
```

`type()` returns the type of an object. These examples confirm that the type of `thing` is allowed to change, and Python correctly infers the type as it changes.

Static Typing

The opposite of dynamic typing is static typing. Static type checks are performed without running the program. In most statically typed languages, for instance C and Java, this is done as your program is compiled.

With static typing, variables generally are not allowed to change types, although mechanisms for casting a variable to a different type may exist.

Let's look at a quick example from a statically typed language. Consider the following Java snippet:

```
Java
String thing;
thing = "Hello";
```

The first line declares that the variable name `thing` is bound to the `String` type at compile time. The name can never be rebound to another type. In the second line, `thing` is assigned a value. It can never be assigned a value that is not a `String` object. For instance, if you were to later say `thing = 28.1f` the compiler would raise an error because of incompatible types.

Python will always [remain a dynamically typed language](#). However, [PEP 484](#) introduced type hints, which make it possible to also do static type checking of Python code.

Unlike how types work in most other statically typed languages, type hints by themselves don't cause Python to enforce types. As the name says, type hints just suggest types. There are other tools, which [you'll see later](#), that perform static type checking using type hints.

Duck Typing

Another term that is often used when talking about Python is [duck typing](#). This moniker comes from the phrase "if it walks like a duck and it quacks like a duck, then it must be a duck" (or [any of its variations](#)).

Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines. Using duck typing you do not check types at all. Instead you check for the presence of a given method or attribute.

As an example, you can call `len()` on any Python object that defines a `__len__()` method:

```
Python >>>
>>> class TheHobbit:
...     def __len__(self):
...         return 95022
```

```
...
>>> the_hobbit = TheHobbit()
>>> len(the_hobbit)
95022
```

Note that the call to `len()` gives the return value of the `__len__()` method. In fact, the implementation of `len()` is essentially equivalent to the following:

Python

```
def len(obj):
    return obj.__len__()
```

In order to call `len(obj)`, the only real constraint on `obj` is that it must define a `__len__()` method. Otherwise, the object can be of types as different as `str`, `list`, `dict`, or `TheHobbit`.

Duck typing is somewhat supported when doing static type checking of Python code, using [structural subtyping](#). You'll learn [more about duck typing](#) later.

Hello Types

In this section you'll see how to add type hints to a function. The following function turns a text string into a headline by adding proper capitalization and a decorative line:

Python

```
def headline(text, align=True):
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "o")
```

By default the function returns the headline left aligned with an underline. By setting the `align` flag to `False` you can alternatively have the headline be centered with a surrounding line of o:

Python

>>>

```
>>> print(headline("python type checking"))
Python Type Checking
-----
>>> print(headline("python type checking", align=False))
ooooooooooooooo Python Type Checking oooooooooooooooo
```

It's time for our first type hints! To add information about types to the function, you simply annotate its arguments and return value as follows:

Python

```
def headline(text: str, align: bool = True) -> str:
    ...
```

The `text: str` syntax says that the `text` argument should be of type `str`. Similarly, the optional `align` argument should have type `bool` with the default value `True`. Finally, the `-> str` notation specifies that `headline()` will return a `string`.

In [terms of style](#), [PEP 8](#) recommends the following:

- Use normal rules for colons, that is, no space before and one space after a colon: `text: str`.
- Use spaces around the `=` sign when combining an argument annotation with a default value: `align: bool = True`.
- Use spaces around the `->` arrow: `def headline(...) -> str`.

Adding type hints like this has no runtime effect: they are only hints and are not enforced on their own. For instance, if we use a wrong type for the (admittedly badly named) `align` argument, the code still runs without any problems

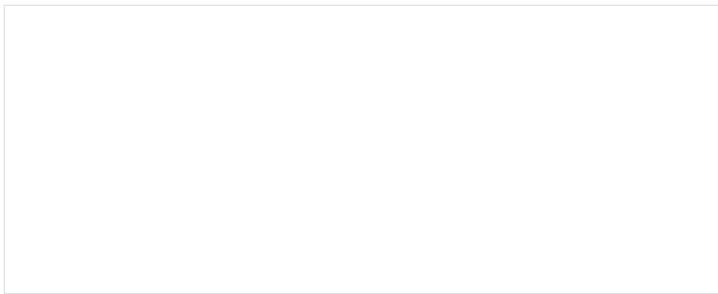
or warnings:

```
Python >>> print(headline("python type checking", align="left"))
Python Type Checking
-----
```

Note: The reason this seemingly works is that the string "left" [compares as truthy](#). Using align="center" would not have the desired effect as "center" is also truthy.

To catch this kind of error you can use a static type checker. That is, a tool that checks the types of your code without actually running it in the traditional sense.

You might already have such a type checker built into your editor. For instance [PyCharm](#) immediately gives you a warning:



The most common tool for doing type checking is [Mypy](#) though. You'll get a short introduction to Mypy in a moment, while you can learn much more about how it works [later](#).

If you don't already have Mypy on your system, you can install it using pip:

Shell

```
$ pip install mypy
```

Put the following code in a file called `headlines.py`:

Python

```
1 # headlines.py
2
3 def headline(text: str, align: bool = True) -> str:
4     if align:
5         return f"{text.title()}\n{'-' * len(text)}"
6     else:
7         return f" {text.title()} ".center(50, "o")
8
9 print(headline("python type checking"))
10 print(headline("use mypy", align="center"))
```

This is essentially the same code you saw earlier: the definition of `headline()` and two examples that are using it.

Now run Mypy on this code:

Shell

```
€ mypy headlines.py
```

```
↳ mypy headlines.py
headlines.py:10: error: Argument "align" to "headline" has incompatible
      type "str"; expected "bool"
```

Based on the type hints, Mypy is able to tell us that we are using the wrong type on line 10.

To fix the issue in the code you should change the value of the align argument you are passing in. You might also rename the align flag to something less confusing:

Python

```
1 # headlines.py
2
3 def headline(text: str, centered: bool = False) -> str:
4     if not centered:
5         return f"{text.title()}\n{'-' * len(text)}"
6     else:
7         return f" {text.title()} ".center(50, "o")
8
9 print(headline("python type checking"))
10 print(headline("use mypy", centered=True))
```

Here you've changed align to centered, and correctly used a boolean value for centered when calling headline(). The code now passes Mypy:

Shell

```
$ mypy headlines.py
$
```

No output from Mypy means that no type errors were detected. Furthermore, when you run the code you see the expected output:

Shell

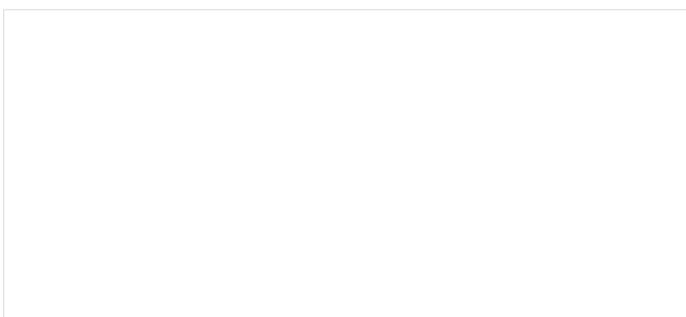
```
$ python headlines.py
Python Type Checking
-----
oooooooooooooooooooo Use Mypy ooooooooooooooooooooo
```

The first headline is aligned to the left, while the second one is centered.

Pros and Cons

The previous section gave you a little taste of what type checking in Python looks like. You also saw an example of one of the advantages of adding types to your code: type hints help **catch certain errors**. Other advantages include:

- Type hints help **document your code**. Traditionally, you would use [docstrings](#) if you wanted to document the expected types of a function's arguments. This works, but as there is no standard for docstrings (despite [PEP 257](#)) they can't be easily used for automatic checks.
- Type hints **improve IDEs and linters**. They make it much easier to statically reason about your code. This in turn allows IDEs to offer better code completion and similar features. With the type annotation, PyCharm knows that text is a string, and can give specific suggestions based on this:

- 
- Type hints help you **build and maintain a cleaner architecture**. The act of writing type hints forces you to think about the types in your program. While the dynamic nature of Python is one of its great assets, being conscious

about relying on duck typing, overloaded methods, or multiple return types is a good thing.

Of course, static type checking is not all peaches and cream. There are also some downsides you should consider:

- Type hints **take developer time and effort to add**. Even though it probably pays off in spending less time debugging, you will spend more time entering code.
- Type hints **work best in modern Pythons**. Annotations were introduced in Python 3.0, and it's possible to use [type comments](#) in Python 2.7. Still, improvements like [variable annotations](#) and [postponed evaluation of type hints](#) mean that you'll have a better experience doing type checks using Python 3.6 or even [Python 3.7](#).
- Type hints **introduce a slight penalty in start-up time**. If you need to use the [typing module](#) the [import](#) time may be significant, especially in short scripts.

So, should you use static type checking in your own code? Well, it's not an all-or-nothing question. Luckily, Python supports the concept of [gradual typing](#). This means that you can gradually introduce types into your code. Code without type hints will be ignored by the static type checker. Therefore, you can start adding types to critical components, and continue as long as it adds value to you.

Looking at the lists above of pros and cons you'll notice that adding types will have no effect on your running program or the users of your program. Type checking is meant to make your life as a developer better and more convenient.

A few rules of thumb on whether to add types to your project are:

- If you are just beginning to learn Python, you can safely wait with type hints until you have more experience.
- Type hints add little value in [short throw-away scripts](#).
- In libraries that will be used by others, especially ones [published on PyPI](#), type hints add a lot of value. Other code using your libraries need these type hints to be properly type checked itself. For examples of projects using type hints see [cursive_re](#), [black](#), our own [Real Python Reader](#), and [Mypy](#) itself.
- In bigger projects, type hints help you understand how types flow through your code, and are highly recommended. Even more so in projects where you cooperate with others.

In his excellent article [The State of Type Hints in Python](#) Bernát Gábor recommends that “**type hints should be used whenever unit tests are worth writing**.” Indeed, type hints play a similar role as [tests](#) in your code: they help you as a developer write better code.

Hopefully you now have an idea about how type checking works in Python and whether it's something you would like to employ in your own projects.

In the rest of this guide, we'll go into more detail about the Python type system, including how you run static type checkers (with particular focus on Mypy), how you type check code that uses libraries without type hints, and how you use annotations at runtime.

Annotations

Annotations were [introduced in Python 3.0](#), originally without any specific purpose. They were simply a way to associate arbitrary expressions to function arguments and return values.

Years later, [PEP 484](#) defined how to add type hints to your Python code, based off work that Jukka Lehtosalo had done on his Ph.D. project—Mypy. The main way to add type hints is using annotations. As type checking is becoming more and more common, this also means that annotations should mainly be reserved for type hints.

The next sections explain how annotations work in the context of type hints.

Function Annotations

For functions, you can annotate arguments and the return value. This is done as follows:

Python

```
def func(arg: arg_type, optarg: arg_type = default) -> return_type:  
    ...
```

For arguments the syntax is argument: annotation, while the return type is annotated using -> annotation. Note that the annotation must be a valid Python expression.

The following simple example adds annotations to a function that calculates the circumference of a circle:

Python

```
import math  
  
def circumference(radius: float) -> float:  
    return 2 * math.pi * radius
```

When running the code, you can also inspect the annotations. They are stored in a special `__annotations__` attribute on the function:

Python

>>>

```
>>> circumference(1.23)  
7.728317927830891  
  
>>> circumference.__annotations__  
{'radius': <class 'float'>, 'return': <class 'float'>}
```

Sometimes you might be confused by how Mypy is interpreting your type hints. For those cases there are special Mypy expressions: `reveal_type()` and `reveal_locals()`. You can add these to your code before running Mypy, and Mypy will dutifully report which types it has inferred. As an example, save the following code to `reveal.py`:

Python

```
1 # reveal.py  
2  
3 import math  
4 reveal_type(math.pi)  
5  
6 radius = 1  
7 circumference = 2 * math.pi * radius  
8 reveal_locals()
```

Next, run this code through Mypy:

Shell

```
$ mypy reveal.py  
reveal.py:4: error: Revealed type is 'builtins.float'  
  
reveal.py:8: error: Revealed local types are:  
  reveal.py:8: error: circumference: builtins.float  
  reveal.py:8: error: radius: builtins.int
```

Even without any annotations Mypy has correctly inferred the types of the built-in `math.pi`, as well as our local variables `radius` and `circumference`.

Note: The reveal expressions are only meant as a tool helping you add types and debug your type hints. If you try to run the `reveal.py` file as a Python script it will crash with a `NameError` since `reveal_type()` is not a function known to the Python interpreter.

If mypy says that ``name `reveal_locals` is not defined`` you might need to update your mypy installation. The `reveal_locals()` expression is available in [Mypy version 0.610](#) and later.

Variable Annotations

In the definition of `circumference()` in the previous section, you only annotated the arguments and the return value. You did not add any annotations inside the function body. More often than not, this is enough.

However, sometimes the type checker needs help in figuring out the types of variables as well. Variable annotations were defined in [PEP 526](#) and introduced in Python 3.6. The syntax is the same as for function argument annotations:

Python

```
pi: float = 3.142

def circumference(radius: float) -> float:
    return 2 * pi * radius
```

The variable `pi` has been annotated with the `float` type hint.

Note: Static type checkers are more than able to figure out that `3.142` is a `float`, so in this example the annotation of `pi` is not necessary. As you learn more about the Python type system, you'll see more relevant examples of variable annotations.

Annotations of variables are stored in the module level `__annotations__` dictionary:

Python

```
>>> circumference(1)
6.284

>>> __annotations__
{'pi': <class 'float'>}
```

>>>

You're allowed to annotate a variable without giving it a value. This adds the annotation to the `__annotations__` dictionary, while the variable remains undefined:

Python

```
>>> nothing: str
>>> nothing
NameError: name 'nothing' is not defined

>>> __annotations__
{'nothing': <class 'str'>}
```

>>>

Since no value was assigned to `nothing`, the name `nothing` is not yet defined.

Type Comments

As mentioned, annotations were introduced in Python 3, and they've not been backported to Python 2. This means that if you're writing code that needs to support [legacy Python](#), you can't use annotations.

Instead, you can use type comments. These are specially formatted comments that can be used to add type hints compatible with older code. To add type comments to a function you do something like this:

Python

```
import math
```

```
def circumference(radius):
    # type: (float) -> float
    return 2 * math.pi * radius
```

The type comments are just comments, so they can be used in any version of Python.

Type comments are handled directly by the type checker, so these types are not available in the `__annotations__` dictionary:

```
Python >>>
>>> circumference.__annotations__
{}
```

A type comment must start with the `type:` literal, and be on the same or the following line as the function definition. If you want to annotate a function with several arguments, you write each type separated by comma:

```
Python
def headline(text, width=80, fill_char="-"):
    # type: (str, int, str) -> str
    return f" {text.title()} ".center(width, fill_char)

print(headline("type comments work", width=40))
```

You are also allowed to write each argument on a separate line with its own annotation:

```
Python
1 # headlines.py
2
3 def headline(
4     text,          # type: str
5     width=80,      # type: int
6     fill_char="-", # type: str
7 ):              # type: (...) -> str
8     return f" {text.title()} ".center(width, fill_char)
9
10 print(headline("type comments work", width=40))
```

Run the example through Python and Mypy:

```
Shell
$ python headlines.py
----- Type Comments Work -----
$ mypy headline.py
$
```

If you have errors, for instance if you happened to call `headline()` with `width="full"` on line 10, Mypy will tell you:

```
Shell
$ mypy headline.py
headline.py:10: error: Argument "width" to "headline" has incompatible
type "str"; expected "int"
```

You can also add type comments to variables. This is done similarly to how you add type comments to arguments:

```
Python
pi = 3.142 # type: float
```

In this example, `pi` will be type checked as a float variable.

So, Type Annotations or Type Comments?

Should you use annotations or type comments when adding type hints to your own code? In short: **Use annotations if you can, use type comments if you must.**

Annotations provide a cleaner syntax keeping type information closer to your code. They are also the [officially recommended way](#) of writing type hints, and will be further developed and properly maintained in the future.

Type comments are more verbose and might conflict with other kinds of comments in your code like [linter directives](#). However, they can be used in code bases that don't support annotations.

There is also hidden option number three: [stub files](#). You will learn about these later, when we discuss [adding types to third party libraries](#).

Stub files will work in any version of Python, at the expense of having to maintain a second set of files. In general, you only want to use stub files if you can't change the original source code.

Playing With Python Types, Part 1

Up until now you've only used basic types like `str`, `float`, and `bool` in your type hints. The Python type system is quite powerful, and supports many kinds of more complex types. This is necessary as it needs to be able to reasonably model Python's dynamic duck typing nature.

In this section you will learn more about this type system, while implementing a simple card game. You will see how to specify:

- The type of [sequences and mappings](#) like tuples, lists and dictionaries
- [Type aliases](#) that make code easier to read
- That functions and methods [do not return anything](#)
- Objects that may be of [any type](#)

After a short detour into some [type theory](#) you will then see [even more ways to specify types in Python](#). You can find the code examples from this section [here](#).

Example: A Deck of Cards

The following example shows an implementation of a [regular \(French\) deck of cards](#):

Python

```
1 # game.py
2
3 import random
4
5 SUITS = "\u2660 \u2661 \u2662 \u2663".split()
6 RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()
7
8 def create_deck(shuffle=False):
9     """Create a new deck of 52 cards"""
10    deck = [(s, r) for r in RANKS for s in SUITS]
11    if shuffle:
12        random.shuffle(deck)
13    return deck
14
15 def deal_hands(deck):
16    """Deal the cards in the deck into four hands"""
17    return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])
18
19 def play():
20    """Play a 4-player card game"""
21    deck = create_deck(shuffle=True)
22    names = "P1 P2 P3 P4".split()
23    hands = {n: h for n, h in zip(names, deal_hands(deck))}
24
25    for name, cards in hands.items():
26        card_str = " ".join(f"{s}{r}" for (s, r) in cards)
27        print(f'{name}: {card_str}'
```

```

27     print(f"\n{name}: {card_str}")
28
29 if __name__ == "__main__":
30     play()

```

Each card is represented as a tuple of strings denoting the suit and rank. The deck is represented as a list of cards. `create_deck()` creates a regular deck of 52 playing cards, and optionally shuffles the cards. `deal_hands()` deals the deck of cards to four players.

Finally, `play()` plays the game. As of now, it only prepares for a card game by constructing a shuffled deck and dealing cards to each player. The following is a typical output:

Shell

```
$ python game.py
P4: ♣9 ♦9 ♥2 ♦7 ♥7 ♣A ♦6 ♥K ♥5 ♦6 ♦3 ♦3 ♦Q
P1: ♥A ♦2 ♣10 ♦J ♣10 ♦4 ♦5 ♥Q ♥5 ♦A ♦5 ♦4
P2: ♦2 ♦7 ♥8 ♦K ♦3 ♥3 ♦K ♦J ♦A ♦7 ♥6 ♥10 ♦K
P3: ♦2 ♦8 ♦8 ♦J ♦Q ♥9 ♥J ♦4 ♦8 ♦10 ♦9 ♥4 ♦Q
```

You will see how to extend this example into a more interesting game as we move along.

Sequences and Mappings

Let's add type hints to our card game. In other words, let's annotate the functions `create_deck()`, `deal_hands()`, and `play()`. The first challenge is that you need to annotate composite types like the list used to represent the deck of cards and the tuples used to represent the cards themselves.

With simple types like `str`, `float`, and `bool`, adding type hints is as easy as using the type itself:

```
Python >>>
>>> name: str = "Guido"
>>> pi: float = 3.142
>>> centered: bool = False
```

With composite types, you are allowed to do the same:

```
Python >>>
>>> names: list = ["Guido", "Jukka", "Ivan"]
>>> version: tuple = (3, 7, 1)
>>> options: dict = {"centered": False, "capitalize": True}
```

However, this does not really tell the full story. What will be the types of `names[2]`, `version[0]`, and `options["centered"]`? In this concrete case you can see that they are `str`, `int`, and `bool`, respectively. However, the type hints themselves give no information about this.

Instead, you should use the special types defined in the [typing module](#). These types add syntax for specifying the types of elements of composite types. You can write the following:

```
Python >>>
>>> from typing import Dict, List, Tuple
>>> names: List[str] = ["Guido", "Jukka", "Ivan"]
>>> version: Tuple[int, int, int] = (3, 7, 1)
>>> options: Dict[str, bool] = {"centered": False, "capitalize": True}
```

Note that each of these types start with a capital letter and that they all use square brackets to define item types:

- `names` is a list of strings
- `version` is a 3-tuple consisting of three integers
- `options` is a dictionary mapping strings to boolean values

The `typing` module contains many more composite types, including `Counter`, `Deque`, `FrozenSet`, `NamedTuple`, and `Set`. In addition, the module includes other kinds of types that you'll see in later sections.

Let's return to the card game. A card is represented by a tuple of two strings. You can write this as `Tuple[str, str]`, so the type of the deck of cards becomes `List[Tuple[str, str]]`. Therefore you can annotate `create_deck()` as follows:

Python

```
8 | def create_deck(shuffle: bool = False) -> List[Tuple[str, str]]:
9 |     """Create a new deck of 52 cards"""
10|     deck = [(s, r) for r in RANKS for s in SUITS]
11|     if shuffle:
12|         random.shuffle(deck)
13|     return deck
```

In addition to the return value, you've also added the `bool` type to the optional `shuffle` argument.

Note: Tuples and lists are annotated differently.

A tuple is an immutable sequence, and typically consists of a fixed number of possibly differently typed elements. For example, we represent a card as a tuple of suit and rank. In general, you write `Tuple[t_1, t_2, ..., t_n]` for an `n`-tuple.

A list is a mutable sequence and usually consists of an unknown number of elements of the same type, for instance a list of cards. No matter how many elements are in the list there is only one type in the annotation: `List[t]`.

In many cases your functions will expect some kind of [sequence](#), and not really care whether it is a list or a tuple. In these cases you should use `typing.Sequence` when annotating the function argument:

Python

```
from typing import List, Sequence

def square(nums: Sequence[float]) -> List[float]:
    return [x**2 for x in nums]
```

Using `Sequence` is an example of using duck typing. A `Sequence` is anything that supports `len()` and `__getitem__()`, independent of its actual type.

Type Aliases

The type hints might become quite oblique when working with nested types like the deck of cards. You may need to stare at `List[Tuple[str, str]]` a bit before figuring out that it matches our representation of a deck of cards.

Now consider how you would annotate `deal_hands()`:

Python

```
15 | def deal_hands(
16 |     deck: List[Tuple[str, str]]
17 | ) -> Tuple[
18 |     List[Tuple[str, str]],
19 |     List[Tuple[str, str]],
20 |     List[Tuple[str, str]],
21 |     List[Tuple[str, str]],
22 | ]:
23 |     """Deal the cards in the deck into four hands"""
24 |     return (deck[0:4], deck[1:4], deck[2:4], deck[3:4])
```

That's just terrible!

Recall that type annotations are regular Python expressions. That means that you can define your own type aliases by assigning them to new variables. You can for instance create `Card` and `Deck` type aliases:

Python

```
from typing import List, Tuple
```

```
Card = Tuple[str, str]
Deck = List[Card]
```

Card can now be used in type hints or in the definition of new type aliases, like Deck in the example above.

Using these aliases, the annotations of deal_hands() become much more readable:

Python

```
15 def deal_hands(deck: Deck) -> Tuple[Deck, Deck, Deck, Deck]:
16     """Deal the cards in the deck into four hands"""
17     return (deck[0:4], deck[1:4], deck[2:4], deck[3:4])
```

Type aliases are great for making your code and its intent clearer. At the same time, these aliases can be inspected to see what they represent:

Python

>>>

```
>>> from typing import List, Tuple
>>> Card = Tuple[str, str]
>>> Deck = List[Card]

>>> Deck
typing.List[typing.Tuple[str, str]]
```

Note that when printing Deck, it shows that it's an alias for a list of 2-tuples of strings.

Functions Without Return Values

You may know that functions without an explicit return still return [None](#):

Python

>>>

```
>>> def play(player_name):
...     print(f"{player_name} plays")
...
>>> ret_val = play("Jacob")
Jacob plays
>>> print(ret_val)
None
```

While such functions technically return something, that return value is not useful. You should add type hints saying as much by using None also as the return type:

Python

```
1 # play.py
2
3 def play(player_name: str) -> None:
4     print(f"{player_name} plays")
5
6 ret_val = play("Filip")
```

The annotations help catch the kinds of subtle bugs where you are trying to use a meaningless return value. Mypy will give you a helpful warning:

Shell

```
$ mypy play.py
play.py:6: error: "play" does not return a value
```

Note that being explicit about a function not returning anything is different from not adding a type hint about the return value:

Python

```
# play.py

def play(player_name: str):
    print(f"{player_name} plays")

ret_val = play("Henrik")
```

In this latter case Mypy has no information about the return value so it will not generate any warning:

Shell

```
$ mypy play.py
$
```

As a more exotic case, note that you can also annotate functions that are never expected to return normally. This is done using [NoReturn](#):

Python

```
from typing import NoReturn

def black_hole() -> NoReturn:
    raise Exception("There is no going back ...")
```

Since `black_hole()` always raises an exception, it will never return properly.

Example: Play Some Cards

Let's return to our [card game example](#). In this second version of the game, we deal a hand of cards to each player as before. Then a start player is chosen and the players take turns playing their cards. There are not really any rules in the game though, so the players will just play random cards:

Python

```
1 # game.py
2
3 import random
4 from typing import List, Tuple
5
6 SUITS = "♠ ♦ ♣ ♤".split()
7 RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()
8
9 Card = Tuple[str, str]
10 Deck = List[Card]
11
12 def create_deck(shuffle: bool = False) -> Deck:
13     """Create a new deck of 52 cards"""
14     deck = [(s, r) for r in RANKS for s in SUITS]
15     if shuffle:
16         random.shuffle(deck)
17     return deck
18
19 def deal_hands(deck: Deck) -> Tuple[Deck, Deck, Deck, Deck]:
20     """Deal the cards in the deck into four hands"""
21     return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])
22
23 def choose(items):
24     """Choose and return a random item"""
25     return random.choice(items)
26
27 def player_order(names, start=None):
28     """Rotate player order so that start goes first"""
29     if start is None:
30         start = choose(names)
31     start_idx = names.index(start)
32     return names[start_idx:] + names[:start_idx]
33
34 def play() -> None:
35     """Play a 4-player card game"""
36     deck = create_deck(shuffle=True)
37     names = "P1 P2 P3 P4".split()
38     hands = {n: h for n, h in zip(names, deal_hands(deck))}
39     start_player = choose(names)
40     turn_order = player_order(names, start=start_player)
41
42     # Randomly play cards from each player's hand until empty
43     while hands[start_player]:
44         for name in turn_order:
45             card = choose(hands[name])
46             hands[name].remove(card)
47             print(f"{name}: {card[0]} {card[1]}{card[1] < 3}, end=""")
48             print()
49
50 if __name__ == "__main__":
51     play()
```

Note that in addition to changing `play()`, we have added two new functions that need type hints: `choose()` and `player_order()`. Before discussing how we'll add type hints to them, here is an example output from running the game:

Shell

```
$ python game.py
P3: ♦10 P4: ♣4 P1: ♥8 P2: ♥Q
P3: ♣8 P4: ♣6 P1: ♣5 P2: ♥K
P3: ♦9 P4: ♥J P1: ♣A P2: ♥A
P3: ♣Q P4: ♣3 P1: ♣7 P2: ♣A
P3: ♥4 P4: ♥6 P1: ♣2 P2: ♣K
P3: ♣K P4: ♣7 P1: ♥7 P2: ♣2
P3: ♣10 P4: ♣4 P1: ♦5 P2: ♥3
P3: ♣Q P4: ♦K P1: ♣J P2: ♥9
P3: ♦2 P4: ♦4 P1: ♣9 P2: ♣10
P3: ♦A P4: ♥5 P1: ♣J P2: ♦Q
P3: ♣8 P4: ♦7 P1: ♦3 P2: ♦J
P3: ♣3 P4: ♥10 P1: ♣9 P2: ♥2
P3: ♦6 P4: ♣6 P1: ♣5 P2: ♦8
```

In this example, player P3 was randomly chosen as the starting player. In turn, each player plays a card: first P3, then P4, then P1, and finally P2. The players keep playing cards as long as they have any left in their hand.

The Any Type

`choose()` works for both lists of names and lists of cards (and any other sequence for that matter). One way to add type hints for this would be the following:

Python

```
import random
from typing import Any, Sequence

def choose(items: Sequence[Any]) -> Any:
    return random.choice(items)
```

This means more or less what it says: `items` is a sequence that can contain items of any type and `choose()` will return one such item of any type. Unfortunately, this is not that useful. Consider the following example:

Python

```
1 # choose.py
2
3 import random
4 from typing import Any, Sequence
5
6 def choose(items: Sequence[Any]) -> Any:
7     return random.choice(items)
8
9 names = ["Guido", "Jukka", "Ivan"]
10 reveal_type(names)
11
12 name = choose(names)
13 reveal_type(name)
```

While Mypy will correctly infer that `names` is a list of strings, that information is lost after the call to `choose()` because of the use of the Any type:

Shell

```
$ mypy choose.py
choose.py:10: error: Revealed type is 'builtins.list[builtins.str*]'
choose.py:13: error: Revealed type is 'Any'
```

You'll see a better way shortly. First though, let's have a more theoretical look at the Python type system, and the special role Any plays.

Type Theory

This tutorial is mainly a practical guide and we will only scratch the surface of the theory underpinning Python type hints. For more details [PEP 483](#) is a good starting point. If you want to get back to the practical examples, feel free to [skip to the next section](#).

Subtypes

One important concept is that of **subtypes**. Formally, we say that a type T is a subtype of U if the following two conditions hold:

- Every value from T is also in the set of values of U type.
- Every function from U type is also in the set of functions of T type.

These two conditions guarantees that even if type T is different from U , variables of type T can always pretend to be U .

For a concrete example, consider $T = \text{bool}$ and $U = \text{int}$. The `bool` type takes only two values. Usually these are denoted `True` and `False`, but these names are just aliases for the integer values `1` and `0`, respectively:

```
Python >>>
>>> int(False)
0

>>> int(True)
1

>>> True + True
2

>>> issubclass(bool, int)
True
```

Since `0` and `1` are both integers, the first condition holds. Above you can see that booleans can be added together, but they can also do anything else integers can. This is the second condition above. In other words, `bool` is a subtype of `int`.

The importance of subtypes is that a subtype can always pretend to be its supertype. For instance, the following code type checks as correct:

```
Python
def double(number: int) -> int:
    return number * 2

print(double(True)) # Passing in bool instead of int
```

Subtypes are somewhat related to subclasses. In fact all subclasses corresponds to subtypes, and `bool` is a subtype of `int` because `bool` is a subclass of `int`. However, there are also subtypes that do not correspond to subclasses. For instance `int` is a subtype of `float`, but `int` is not a subclass of `float`.

Covariant, Contravariant, and Invariant

What happens when you use subtypes inside composite types? For instance, is `Tuple[bool]` a subtype of `Tuple[int]`? The answer depends on the composite type, and whether that type is [covariant, contravariant, or invariant](#). This gets technical fast, so let's just give a few examples:

- `Tuple` is covariant. This means that it preserves the type hierarchy of its item types: `Tuple[bool]` is a subtype of `Tuple[int]` because `bool` is a subtype of `int`.
- `List` is invariant. Invariant types give no guarantee about subtypes. While all values of `List[bool]` are values of `List[int]`, you can append an `int` to `List[int]` and not to `List[bool]`. In other words, the second condition for subtypes does not hold, and `List[bool]` is not a subtype of `List[int]`.

- `Callable` is contravariant in its arguments. This means that it reverses the type hierarchy. You will see now `Callable` works [later](#), but for now think of `Callable[[T], ...]` as a function with its only argument being of type `T`. An example of a `Callable[[int], ...]` is the `double()` function defined above. Being contravariant means that if a function operating on a `bool` is expected, then a function operating on an `int` would be acceptable.

In general, you don't need to keep these expression straight. However, you should be aware that subtypes and composite types may not be simple and intuitive.

Gradual Typing and Consistent Types

Earlier we mentioned that Python supports [gradual typing](#), where you can gradually add type hints to your Python code. Gradual typing is essentially made possible by the `Any` type.

Somehow `Any` sits both at the top and at the bottom of the type hierarchy of subtypes. `Any` type behaves as if it is a subtype of `Any`, and `Any` behaves as if it is a subtype of any other type. Looking at the definition of subtypes above this is not really possible. Instead we talk about **consistent types**.

The type `T` is consistent with the type `U` if `T` is a subtype of `U` or either `T` or `U` is `Any`.

The type checker only complains about inconsistent types. The takeaway is therefore that you will never see type errors arising from the `Any` type.

This means that you can use `Any` to explicitly fall back to dynamic typing, describe types that are too complex to describe in the Python type system, or describe items in composite types. For instance, a dictionary with string keys that can take any type as its values can be annotated `Dict[str, Any]`.

Do remember, though, if you use `Any` the static type checker will effectively not do any type any checking.

Playing With Python Types, Part 2

Let's return to our practical examples. Recall that you were trying to annotate the general `choose()` function:

Python

```
import random
from typing import Any, Sequence

def choose(items: Sequence[Any]) -> Any:
    return random.choice(items)
```

The problem with using `Any` is that you are needlessly losing type information. You know that if you pass a list of strings to `choose()`, it will return a string. Below you'll see how to express this using type variables, as well as how to work with:

- [Duck types and protocols](#)
- Arguments with [None as default value](#)
- [Class methods](#)
- [The type of your own classes](#)
- [Variable number of arguments](#)

Type Variables

A type variable is a special variable that can take on any type, depending on the situation.

Let's create a type variable that will effectively encapsulate the behavior of `choose()`:

Python

```
1 # choose.py
2
3 import random
4 from typing import Sequence, TypeVar
5
6 Choosable = TypeVar("Choosable")
7
8 def choose(items: Sequence[Choosable]) -> Choosable:
```

```

9     return random.choice(items)
10
11 names = ["Guido", "Jukka", "Ivan"]
12 reveal_type(names)
13
14 name = choose(names)
15 reveal_type(name)

```

A type variable must be defined using `TypeVar` from the `typing` module. When used, a type variable ranges over all possible types and takes the most specific type possible. In the example, `name` is now a `str`:

Shell

```

$ mypy choose.py
choose.py:12: error: Revealed type is 'builtins.list[builtins.str*]'
choose.py:15: error: Revealed type is 'builtins.str*'

```

Consider a few other examples:

Python

```

1 # choose_examples.py
2
3 from choose import choose
4
5 reveal_type(choose(["Guido", "Jukka", "Ivan"]))
6 reveal_type(choose([1, 2, 3]))
7 reveal_type(choose([True, 42, 3.14]))
8 reveal_type(choose(["Python", 3, 7]))

```

The first two examples should have type `str` and `int`, but what about the last two? The individual list items have different types, and in that case the choosable type variable does its best to accommodate:

Shell

```

$ mypy choose_examples.py
choose_examples.py:5: error: Revealed type is 'builtins.str*'
choose_examples.py:6: error: Revealed type is 'builtins.int*'
choose_examples.py:7: error: Revealed type is 'builtins.float*'
choose_examples.py:8: error: Revealed type is 'builtins.object*'

```

As you've already seen `bool` is a subtype of `int`, which again is a subtype of `float`. So in the third example the return value of `choose()` is guaranteed to be something that can be thought of as a `float`. In the last example, there is no subtype relationship between `str` and `int`, so the best that can be said about the return value is that it is an `object`.

Note that none of these examples raised a type error. Is there a way to tell the type checker that `choose()` should accept both strings and numbers, but not both at the same time?

You can constrain type variables by listing the acceptable types:

Python

```

1 # choose.py
2
3 import random
4 from typing import Sequence, TypeVar
5
6 Choosable = TypeVar("Choosable", str, float)
7
8 def choose(items: Sequence[Choosable]) -> Choosable:
9     return random.choice(items)
10
11 reveal_type(choose(["Guido", "Jukka", "Ivan"]))
12 reveal_type(choose([1, 2, 3]))

```

```
13 | reveal_type(choose([True, 42, 3.14]))
14 | reveal_type(choose(["Python", 3, 7]))
```

Now Choosable can only be either `str` or `float`, and Mypy will note that the last example is an error:

Shell

```
$ mypy choose.py
choose.py:11: error: Revealed type is 'builtins.str'
choose.py:12: error: Revealed type is 'builtins.float'
choose.py:13: error: Revealed type is 'builtins.float'
choose.py:14: error: Revealed type is 'builtins.object'
choose.py:14: error: Value of type variable "Choosable" of "choose"
                  cannot be "object"
```

Also note that in the second example the type is considered `float` even though the input list only contains `int` objects. This is because Choosable was restricted to strings and floats and `int` is a subtype of `float`.

In our card game we want to restrict `choose()` to be used for `str` and `Card`:

Python

```
Choosable = TypeVar("Choosable", str, Card)

def choose(items: Sequence[Choosable]) -> Choosable:
    ...
```

We briefly mentioned that `Sequence` represents both lists and tuples. As we noted, a `Sequence` can be thought of as a duck type, since it can be any object with `.__len__()` and `.__getitem__()` implemented.

Duck Types and Protocols

Recall the following example from [the introduction](#):

Python

```
def len(obj):
    return obj.__len__()
```

`len()` can return the length of any object that has implemented the `.__len__()` method. How can we add type hints to `len()`, and in particular the `obj` argument?

The answer hides behind the academic sounding term [structural subtyping](#). One way to categorize type systems is by whether they are **nominal** or **structural**:

- In a **nominal** system, comparisons between types are based on names and declarations. The Python type system is mostly nominal, where an `int` can be used in place of a `float` because of their subtype relationship.
- In a **structural** system, comparisons between types are based on structure. You could define a structural type `Sized` that includes all instances that define `.__len__()`, irrespective of their nominal type.

There is ongoing work to bring a full-fledged structural type system to Python through [PEP 544](#) which aims at adding a concept called protocols. Most of PEP 544 is already [implemented in Mypy](#) though.

A protocol specifies one or more methods that must be implemented. For example, all classes defining `.__len__()` fulfill the `typing.Sized` protocol. We can therefore annotate `len()` as follows:

Python

```
from typing import Sized

def len(obj: Sized) -> int:
    return obj.__len__()
```

Other [examples of protocols](#) defined in the `typing` module include `Container`, `Iterable`, `Awaitable`, and `ContextManager`.

You can also define your own protocols. This is done by inheriting from `Protocol` and defining the function signatures (with empty function bodies) that the protocol expects. The following example shows how `len()` and `sized` could have been implemented:

Python

```
from typing_extensions import Protocol

class Sized(Protocol):
    def __len__(self) -> int: ...

def len(obj: Sized) -> int:
    return obj.__len__()
```

At the time of writing the support for self-defined protocols is still experimental and only available through the `typing_extensions` module. This module must be explicitly installed from [PyPI](#) by doing `pip install typing-extensions`.

The Optional Type

A common pattern in Python is to use `None` as a default value for an argument. This is usually done either to avoid problems with [mutable default values](#) or to have a sentinel value flagging special behavior.

In the card example, the `player_order()` function uses `None` as a sentinel value for `start` saying that if no start player is given it should be chosen randomly:

Python

```
27 | def player_order(names, start=None):
28 |     """Rotate player order so that start goes first"""
29 |     if start is None:
30 |         start = choose(names)
31 |     start_idx = names.index(start)
32 |     return names[start_idx:] + names[:start_idx]
```

The challenge this creates for type hinting is that in general `start` should be a string. However, it may also take the special non-string value `None`.

In order to annotate such arguments you can use the `Optional` type:

Python

```
from typing import Sequence, Optional

def player_order(
    names: Sequence[str], start: Optional[str] = None
) -> Sequence[str]:
    ...
    ...
```

The `Optional` type simply says that a variable either has the type specified or is `None`. An equivalent way of specifying the same would be using the `Union` type: `Union[None, str]`

Note that when using either `Optional` or `Union` you must take care that the variable has the correct type when you operate on it. This is done in the example by testing whether `start is None`. Not doing so would cause both static type errors as well as possible runtime errors:

Python

```
1 | # player_order.py
2 |
3 | from typing import Sequence, Optional
```

```

4
5     def player_order(
6         names: Sequence[str], start: Optional[str] = None
7     ) -> Sequence[str]:
8         start_idx = names.index(start)
9         return names[start_idx:] + names[:start_idx]

```

Mypy tells you that you have not taken care of the case where `start` is `None`:

Shell

```
$ mypy player_order.py
player_order.py:8: error: Argument 1 to "index" of "list" has incompatible
type "Optional[str]"; expected "str"
```

Note: The use of `None` for optional arguments is so common that MyPy handles it automatically. MyPy assumes that a default argument of `None` indicates an optional argument even if the type hint does not explicitly say so. You could have used the following:

Python

```
def player_order(names: Sequence[str], start: str = None) -> Sequence[str]:
    ...
```

If you don't want MyPy to make this assumption you can turn it off with the `--no-implicit-optional` command line option.

Example: The Object(ive) of the Game

Let's rewrite the card game to be more [object-oriented](#). This will allow us to discuss how to properly annotate classes and methods.

A more or less direct translation of our card game into code that uses classes for `Card`, `Deck`, `Player`, and `Game` looks something like the following:

Python

```

1 # game.py
2
3 import random
4 import sys
5
6 class Card:
7     SUITS = "\u2660 \u2661 \u2662 \u2663".split()
8     RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()
9
10    def __init__(self, suit, rank):
11        self.suit = suit
12        self.rank = rank
13
14    def __repr__(self):
15        return f"{self.suit}{self.rank}"
16
17 class Deck:
18    def __init__(self, cards):
19        self.cards = cards
20
21    @classmethod
22    def create(cls, shuffle=False):
23        """Create a new deck of 52 cards"""
24        cards = [Card(s, r) for r in Card.RANKS for s in Card.SUITS]
25        if shuffle:
26            random.shuffle(cards)
27        return cls(cards)
28
29    def deal(self, num_hands):
30        """Deal the cards in the deck into a number of hands"""
31        cls = self.__class__
32        return tuple(cls(self.cards[i::num_hands]) for i in range(num_hands))
33

```

```

34 class Player:
35     def __init__(self, name, hand):
36         self.name = name
37         self.hand = hand
38
39     def play_card(self):
40         """play a card from the player's hand"""
41         card = random.choice(self.hand.cards)
42         self.hand.cards.remove(card)
43         print(f"{self.name}: {card!r:<3} ", end="")
44         return card
45
46 class Game:
47     def __init__(self, *names):
48         """Set up the deck and deal cards to 4 players"""
49         deck = Deck.create(shuffle=True)
50         self.names = (list(names) + "P1 P2 P3 P4".split())[:4]
51         self.hands = {
52             n: Player(n, h) for n, h in zip(self.names, deck.deal(4))
53         }
54
55     def play(self):
56         """Play a card game"""
57         start_player = random.choice(self.names)
58         turn_order = self.player_order(start=start_player)
59
60         # Play cards from each player's hand until empty
61         while self.hands[start_player].hand.cards:
62             for name in turn_order:
63                 self.hands[name].play_card()
64             print()
65
66     def player_order(self, start=None):
67         """Rotate player order so that start goes first"""
68         if start is None:
69             start = random.choice(self.names)
70             start_idx = self.names.index(start)
71             return self.names[start_idx:] + self.names[:start_idx]
72
73 if __name__ == "__main__":
74     # Read player names from command line
75     player_names = sys.argv[1:]
76     game = Game(*player_names)
77     game.play()

```

Now let's add types to this code.

Type Hints for Methods

First of all type hints for methods work much the same as type hints for functions. The only difference is that the `self` argument need not be annotated, as it always will be a class instance. The types of the `Card` class are easy to add:

Python

```

6  class Card:
7      SUITS = "♠ ♥ ♦ ♣".split()
8      RANKS = "2 3 4 5 6 7 8 9 10 J Q K A".split()
9
10     def __init__(self, suit: str, rank: str) -> None:
11         self.suit = suit
12         self.rank = rank
13
14     def __repr__(self) -> str:
15         return f"{self.suit}{self.rank}"

```

Note that the `__init__()` method always should have `None` as its return type.

Classes as Types

There is a correspondence between classes and types. For example, all instances of the `Card` class together form the `Card` type. To use classes as types you simply use the name of the class.

For example, a Deck essentially consists of a list of Card objects. You can annotate this as follows:

Python

```
17 | class Deck:
18 |     def __init__(self, cards: List[Card]) -> None:
19 |         self.cards = cards
```

Mypy is able to connect your use of Card in the annotation with the definition of the Card class.

This doesn't work as cleanly though when you need to refer to the class currently being defined. For example, the Deck.create() class method returns an object with type Deck. However, you can't simply add -> Deck as the Deck class is not yet fully defined.

Instead, you are allowed to use string literals in annotations. These strings will only be evaluated by the type checker later, and can therefore contain self and forward references. The .create() method should use such string literals for its types:

Python

```
20 | class Deck:
21 |     @classmethod
22 |     def create(cls, shuffle: bool = False) -> "Deck":
23 |         """Create a new deck of 52 cards"""
24 |         cards = [Card(s, r) for r in Card.RANKS for s in Card.SUITS]
25 |         if shuffle:
26 |             random.shuffle(cards)
27 |         return cls(cards)
```

Note that the Player class also will reference the Deck class. This is however no problem, since Deck is defined before Player:

Python

```
34 | class Player:
35 |     def __init__(self, name: str, hand: Deck) -> None:
36 |         self.name = name
37 |         self.hand = hand
```

Usually annotations are not used at runtime. This has given wings to the idea of [postponing the evaluation of annotations](#). Instead of evaluating annotations as Python expressions and storing their value, the proposal is to store the string representation of the annotation and only evaluate it when needed.

Such functionality is planned to become standard in the still mythical [Python 4.0](#). However, in [Python 3.7](#) and later, forward references are available through a __future__ import:

Python

```
from __future__ import annotations

class Deck:
    @classmethod
    def create(cls, shuffle: bool = False) -> Deck:
        ...
```

With the __future__ import you can use Deck instead of "Deck" even before Deck is defined.

Returning self or cls

As noted, you should typically not annotate the self or cls arguments. Partly, this is not necessary as self points to an instance of the class, so it will have the type of the class. In the card example, self has the implicit type Card. Also, adding this type explicitly would be cumbersome since the class is not defined yet. You would have to use the string literal syntax, self: "Card".

There is one case where you might want to annotate `self` or `cls`, though. Consider what happens if you have a superclass that other classes inherit from, and which has methods that return `self` or `cls`:

Python

```
1 # dogs.py
2
3 from datetime import date
4
5 class Animal:
6     def __init__(self, name: str, birthday: date) -> None:
7         self.name = name
8         self.birthday = birthday
9
10    @classmethod
11    def newborn(cls, name: str) -> "Animal":
12        return cls(name, date.today())
13
14    def twin(self, name: str) -> "Animal":
15        cls = self.__class__
16        return cls(name, self.birthday)
17
18 class Dog(Animal):
19     def bark(self) -> None:
20         print(f"{self.name} says woof!")
21
22 fido = Dog.newborn("Fido")
23 pluto = fido.twin("Pluto")
24 fido.bark()
25 pluto.bark()
```

While the code runs without problems, Mypy will flag a problem:

Shell

```
$ mypy dogs.py
dogs.py:24: error: "Animal" has no attribute "bark"
dogs.py:25: error: "Animal" has no attribute "bark"
```

The issue is that even though the inherited `Dog.newborn()` and `Dog.twin()` methods will return a `Dog` the annotation says that they return an `Animal`.

In cases like this you want to be more careful to make sure the annotation is correct. The return type should match the type of `self` or the instance type of `cls`. This can be done using type variables that keep track of what is actually passed to `self` and `cls`:

Python

```
# dogs.py
from datetime import date
from typing import Type, TypeVar

TAnimal = TypeVar("TAnimal", bound="Animal")

class Animal:
    def __init__(self, name: str, birthday: date) -> None:
        self.name = name
        self.birthday = birthday

    @classmethod
    def newborn(cls: Type[TAnimal], name: str) -> TAnimal:
        return cls(name, date.today())

    def twin(self: TAnimal, name: str) -> TAnimal:
        cls = self.__class__
        return cls(name, self.birthday)

class Dog(Animal):
    def bark(self) -> None:
        print(f"{self.name} says woof!")
```

```

fido = Dog.newborn("Fido")
pluto = fido.twin("Pluto")
fido.bark()
pluto.bark()

```

There are a few things to note in this example:

- The type variable `TAnimal` is used to denote that return values might be instances of subclasses of `Animal`.
- We specify that `Animal` is an upper bound for `TAnimal`. Specifying bound means that `TAnimal` will only be `Animal` or one of its subclasses. This is needed to properly restrict the types that are allowed.
- The `typing.Type[]` construct is the typing equivalent of `type()`. You need it to note that the class method expects a class and returns an instance of that class.

Annotating *args and **kwargs

In the [object oriented version](#) of the game, we added the option to name the players on the command line. This is done by listing player names after the name of the program:

Shell

```

$ python game.py GeirArne Dan Joanna
Dan: ♦A Joanna: ♥9 P1: ♣A GeirArne: ♣2
Dan: ♥A Joanna: ♥6 P1: ♣4 GeirArne: ♦8
Dan: ♦K Joanna: ♦Q P1: ♣K GeirArne: ♣5
Dan: ♥2 Joanna: ♥J P1: ♣7 GeirArne: ♥K
Dan: ♦10 Joanna: ♣3 P1: ♦4 GeirArne: ♣8
Dan: ♣6 Joanna: ♥Q P1: ♣Q GeirArne: ♦J
Dan: ♦2 Joanna: ♥4 P1: ♣8 GeirArne: ♥7
Dan: ♥10 Joanna: ♦3 P1: ♥3 GeirArne: ♣2
Dan: ♠K Joanna: ♣5 P1: ♣7 GeirArne: ♣J
Dan: ♦6 Joanna: ♦9 P1: ♣J GeirArne: ♣10
Dan: ♣3 Joanna: ♥5 P1: ♣9 GeirArne: ♣Q
Dan: ♣A Joanna: ♥9 P1: ♣10 GeirArne: ♥8
Dan: ♦6 Joanna: ♦5 P1: ♦7 GeirArne: ♣4

```

This is implemented by unpacking and passing in `sys.argv` to `Game()` when it's instantiated. The `__init__()` method uses `*names` to pack the given names into a tuple.

Regarding type annotations: even though `names` will be a tuple of strings, you should only annotate the type of each name. In other words, you should use `str` and not `Tuple[str]`:

Python

```

46 class Game:
47     def __init__(self, *names: str) -> None:
48         """Set up the deck and deal cards to 4 players"""
49         deck = Deck.create(shuffle=True)
50         self.names = (list(names) + "P1 P2 P3 P4".split())[:4]
51         self.hands = {
52             n: Player(n, h) for n, h in zip(self.names, deck.deal(4))
53         }

```

Similarly, if you have a function or method accepting `**kwargs` you should only annotate the type of each possible keyword argument.

Callables

Functions are [first-class objects](#) in Python. This means that you can use functions as arguments to other functions. That also means that you need to be able to add type hints representing functions

That also means that you need to be able to add type hints representing functions.

Functions, as well as lambdas, methods and classes, are represented by [typing.Callable](#). The types of the arguments and the return value are usually also represented. For instance, `Callable[[A1, A2, A3], Rt]` represents a function with three arguments with types A1, A2, and A3, respectively. The return type of the function is Rt.

In the following example, the function `do_twice()` calls a given function twice and prints the return values:

Python

```
1 # do_twice.py
2
3 from typing import Callable
4
5 def do_twice(func: Callable[[str], str], argument: str) -> None:
6     print(func(argument))
7     print(func(argument))
8
9 def create_greeting(name: str) -> str:
10     return f"Hello {name}"
11
12 do_twice(create_greeting, "Jekyll")
```

Note the annotation of the `func` argument to `do_twice()` on line 5. It says that `func` should be a callable with one string argument, that also returns a string. One example of such a callable is `create_greeting()` defined on line 9.

Most callable types can be annotated in a similar manner. However, if you need more flexibility, check out [callback protocols](#) and [extended callable types](#).

Example: Hearts

Let's end with a full example of the game of [Hearts](#). You might already know this game from other computer simulations. Here is a quick recap of the rules:

- Four players play with a hand of 13 cards each.
- The player holding the ♠2 starts the first round, and must play ♠2.
- Players take turns playing cards, following the leading suit if possible.
- The player playing the highest card in the leading suit wins the trick, and becomes start player in the next turn.
- A player can not lead with a ♥ until a ♥ has already been played in an earlier trick.
- After all cards are played, players get points if they take certain cards:
 - 13 points for the ♠Q
 - 1 point for each ♥
- A game lasts several rounds, until one player has 100 points or more. The player with the least points wins.

More details can be found [found online](#).

There are not many new typing concepts in this example that you have not already seen. We'll therefore not go through this code in detail, but leave it as an example of annotated code.

Source Code for the Hearts Card Game

Show/Hide

Here are a few points to note in the code:

- For type relationships that are hard to express using Union or type variables, you can use the `@overload` decorator. See `Deck.__getitem__()` for an example and [the documentation](#) for more information.
- Subclasses correspond to subtypes, so that a `HumanPlayer` can be used wherever a `Player` is expected.

- When a subclass reimplements a method from a superclass, the type annotations must match. See `HumanPlayer.play_card()` for an example.

When starting the game, you control the first player. Enter numbers to choose which cards to play. The following is an example of game play, with the highlighted lines showing where the player made a choice:

Shell

```
$ python hearts.py GeirArne Aldren Joanna Brad

Starting new round:
Brad -> ♣2
  0: ♣5  1: ♣Q  2: ♣K  (Rest: ♦6 ♥10 ♥6 ♣J ♥3 ♥9 ♦10 ♣7 ♣K ♣4)
  GeirArne, choose card: 2
GeirArne => ♣K
Aldren -> ♣10
Joanna -> ♣9
GeirArne wins the trick

  0: ♣4  1: ♣5  2: ♦6  3: ♣7  4: ♦10  5: ♣J  6: ♣Q  7: ♣K  (Rest: ♥10 ♥6 ♥3 ♥9)
  GeirArne, choose card: 0
GeirArne => ♣4
Aldren -> ♣5
Joanna -> ♣3
Brad -> ♣2
Aldren wins the trick

...
Joanna -> ♥J
Brad -> ♥2
  0: ♥6  1: ♥9  (Rest: )
  GeirArne, choose card: 1
GeirArne => ♥9
Aldren -> ♥A
Aldren wins the trick

Aldren -> ♠A
Joanna -> ♥Q
Brad -> ♣J
  0: ♥6  (Rest: )
  GeirArne, choose card: 0
GeirArne => ♥6
Aldren wins the trick

Scores:
Brad      14 14
Aldren    10 10
GeirArne   1  1
Joanna    1  1
```

Static Type Checking

So far you have seen how to add type hints to your code. In this section you'll learn more about how to actually perform static type checking of Python code.

The Mypy Project

Mypy was started by Jukka Lehtosalo during his Ph.D. studies at Cambridge around 2012. Mypy was originally envisioned as a Python variant with seamless dynamic and static typing. See [Jukka's slides from PyCon Finland 2012](#) for examples of the original vision of Mypy.

Most of those original ideas still play a big part in the Mypy project. In fact, the slogan “Seamless dynamic and static typing” is still [prominently visible on Mypy’s home page](#) and describes the motivation for using type hints in Python well.

The biggest change since 2012 is that Mypy is no longer a *variant* of Python. In its first versions Mypy was a stand-alone language that was compatible with Python except for its type declarations. Following a [suggestion by Guido van Rossum](#), Mypy was rewritten to use annotations instead. Today Mypy is a static type checker for *regular* Python code.

Running Mypy

Before running Mypy for the first time, you must install the program. This is most easily done using pip:

Shell

```
$ pip install mypy
```

With Mypy installed, you can run it as a regular command line program:

Shell

```
$ mypy my_program.py
```

Running Mypy on your `my_program.py` Python file will check it for type errors without actually executing the code.

There are many available options when type checking your code. As Mypy is still under very active development, command line options are liable to change between versions. You should refer to Mypy’s help to see which settings are default on your version:

Shell

```
$ mypy --help
usage: mypy [-h] [-v] [-V] [more options; see below]
             [-m MODULE] [-p PACKAGE] [-c PROGRAM_TEXT] [files ...]

Mypy is a program that will type check your Python code.

[... The rest of the help hidden for brevity ...]
```

Additionally, the [Mypy command line documentation online](#) has a lot of information.

Let’s look at some of the most common options. First of all, if you are using third-party packages without type hints, you may want to silence Mypy’s warnings about these. This can be done with the `--ignore-missing-imports` option.

The following example uses [NumPy](#) to calculate and print the cosine of several numbers:

Python

```
1 # cosine.py
2
3 import numpy as np
4
5 def print_cosine(x: np.ndarray) -> None:
6     with np.printoptions(precision=3, suppress=True):
7         print(np.cos(x))
8
9 x = np.linspace(0, 2 * np.pi, 9)
10 print_cosine(x)
```

Note that `np.printoptions()` is only available in version 1.15 and later of NumPy. Running this example prints

some numbers to the console:

Shell

```
$ python cosine.py
[ 1.      0.707  0.     -0.707 -1.     -0.707 -0.     0.707  1.     ]
```

The actual output of this example is not important. However, you should note that the argument `x` is annotated with `np.ndarray` on line 5, as we want to print the cosine of a full array of numbers.

You can run Mypy on this file as usual:

Shell

```
$ mypy cosine.py
cosine.py:3: error: No library stub file for module 'numpy'
cosine.py:3: note: (Stub files are from https://github.com/python/typeshed)
```

These warnings may not immediately make much sense to you, but you'll learn about [stubs](#) and [typeshed](#) soon. You can essentially read the warnings as Mypy saying that the Numpy package does not contain type hints.

In most cases, missing type hints in third-party packages is not something you want to be bothered with so you can silence these messages:

Shell

```
$ mypy --ignore-missing-imports cosine.py
$
```

If you use the `--ignore-missing-imports` command line option, Mypy will [not try to follow or warn about any missing imports](#). This might be a bit heavy-handed though, as it also ignores actual mistakes, like misspelling the name of a package.

Two less intrusive ways of handling third-party packages are using type comments or configuration files.

In a simple example as the one above, you can silence the numpy warning by adding a type comment to the line containing the import:

Python

```
3 | import numpy as np # type: ignore
```

The literal `# type: ignore` tells Mypy to ignore the import of Numpy.

If you have several files, it might be easier to keep track of which imports to ignore in a configuration file. Mypy reads a file called `mypy.ini` in the current directory if it is present. This configuration file must contain a section called `[mypy]` and may contain module specific sections of the form `[mypy-module]`.

The following configuration file will ignore that Numpy is missing type hints:

Config File

```
# mypy.ini

[mypy]

[mypy-numpy]
ignore_missing_imports = True
```

There are many options that can be specified in the configuration file. It is also possible to specify a global configuration file. See the [documentation](#) for more information.

Adding Stubs

Type hints are available for all the packages in the Python standard library. However, if you are using third-party packages you've already seen that the situation can be different.

The following example uses the [Parse package](#) to do simple text parsing. To follow along you should first install Parse:

Shell

```
$ pip install parse
```

Parse can be used to recognize simple patterns. Here is a small program that tries its best to figure out your name:

Python

```
1 # parse_name.py
2
3 import parse
4
5 def parse_name(text: str) -> str:
6     patterns = (
7         "my name is {name}",
8         "i'm {name}",
9         "i am {name}",
10        "call me {name}",
11        "{name}",
12    )
13    for pattern in patterns:
14        result = parse.parse(pattern, text)
15        if result:
16            return result["name"]
17    return ""
18
19 answer = input("What is your name? ")
20 name = parse_name(answer)
21 print(f"Hi {name}, nice to meet you!")
```

The main flow is defined in the last three lines: ask for your name, parse the answer, and print a greeting. The parse package is called on line 14 in order to try to find a name based on one of the patterns listed on lines 7-11.

The program can be used as follows:

Shell

```
$ python parse_name.py
What is your name? I am Geir Arne
Hi Geir Arne, nice to meet you!
```

Note that even though I answer I am Geir Arne, the program figures out that I am is not part of my name.

Let's add a small bug to the program, and see if Mypy is able to help us detect it. Change line 16 from `return result["name"]` to `return result`. This will return a `parse.Result` object instead of the string containing the name.

Next run Mypy on the program:

Shell

```
$ mypy parse_name.py
parse_name.py:3: error: Cannot find module named 'parse'
parse_name.py:3: note: (Perhaps setting MYPYPATH or using the
      "--ignore-missing-imports" flag would help)
```

Mypy prints a similar error to the one you saw in the previous section: It doesn't know about the `parse` package. You could try to ignore the import:

Shell

```
$ mypy parse_name.py --ignore-missing-imports
$
```

Unfortunately, ignoring the import means that Mypy has no way of discovering the bug in our program. A better

solution would be to add type hints to the Parse package itself. As [Parse is open source](#) you can actually add types to the source code and send a pull request.

Alternatively, you can add the types in a [stub file](#). A stub file is a text file that contains the signatures of methods and functions, but not their implementations. Their main function is to add type hints to code that you for some reason can't change. To show how this works, we will add some stubs for the Parse package.

First of all, you should put all your stub files inside one common directory, and set the `MYPYPATH` environment variable to point to this directory. On Mac and Linux you can set `MYPYPATH` as follows:

Shell

```
$ export MYPYPATH=/home/gahjelle/python/stubs
```

You can set the variable permanently by adding the line to your `.bashrc` file. On Windows you can click the start menu and search for *environment variables* to set `MYPYPATH`.

Next, create a file inside your `stubs` directory that you call `parse.pyi`. It must be named for the package that you are adding type hints for, with a `.pyi` suffix. Leave this file empty for now. Then run Mypy again:

Shell

```
$ mypy parse_name.py
parse_name.py:14: error: Module has no attribute "parse"
```

If you have set everything up correctly, you should see this new error message. Mypy uses the new `parse.pyi` file to figure out which functions are available in the `parse` package. Since the stub file is empty, Mypy assumes that `parse.parse()` does not exist, and then gives the error you see above.

The following example does not add types for the whole `parse` package. Instead it shows the type hints you need to add in order for Mypy to type check your use of `parse.parse()`:

Python

```
# parse.pyi

from typing import Any, Mapping, Optional, Sequence, Tuple, Union

class Result:
    def __init__(
        self,
        fixed: Sequence[str],
        named: Mapping[str, str],
        spans: Mapping[int, Tuple[int, int]],
    ) -> None: ...
    def __getitem__(self, item: Union[int, str]) -> str: ...
    def __repr__(self) -> str: ...

def parse(
    format: str,
    string: str,
    evaluate_result: bool = ...,
    case_sensitive: bool = ...,
) -> Optional[Result]: ...
```

The ellipsis `...` are part of the file, and should be written exactly as above. The stub file should only contain type hints for variables, attributes, functions, and methods, so the implementations should be left out and replaced by `...` markers.

Finally Mypy is able to spot the bug we introduced:

Shell

```
$ mypy parse_name.py
parse_name.py:16: error: Incompatible return value type (got
    "Result", expected "str")
```

This points straight to line 16 and the fact that we return a `Result` object and not the name string. Change `return result` back to `return result["name"]`, and run Mypy again to see that it's happy.

Typeshed

You've seen how to use stubs to add type hints without changing the source code itself. In the previous section we added some type hints to the third-party `Parse` package. Now, it wouldn't be very effective if everybody needs to create their own stubs files for all third-party packages they are using.

[Typeshed](#) is a Github repository that contains type hints for the Python standard library, as well as many third-party packages. Typeshed comes included with Mypy so if you are using a package that already has type hints defined in Typeshed, the type checking will just work.

You can also [contribute type hints to Typeshed](#). Make sure to get the permission of the owner of the package first though, especially because they might be working on adding type hints into the source code itself—which is the [preferred approach](#).

Other Static Type Checkers

In this tutorial, we have mainly focused on type checking using Mypy. However, there are other static type checkers in the Python ecosystem.

The [PyCharm](#) editor comes with its own type checker included. If you are using PyCharm to write your Python code, it will be automatically type checked.

Facebook has developed [Pyre](#). One of its stated goals is to be fast and performant. While there are some differences, Pyre functions mostly similar to Mypy. See the [documentation](#) if you're interested in trying out Pyre.

Furthermore, Google has created [Pytype](#). This type checker also works mostly the same as Mypy. In addition to checking annotated code, Pytype has some support for running type checks on unannotated code and even adding annotations to code automatically. See the [quickstart](#) document for more information.

Using Types at Runtime

As a final note, it's possible to use type hints also at runtime during execution of your Python program. Runtime type checking will probably never be natively supported in Python.

However, the type hints are available at runtime in the `__annotations__` dictionary, and you can use those to do type checks if you desire. Before you run off and write your own package for enforcing types, you should know that there are already several packages doing this for you. Have a look at [Enforce](#), [Pydantic](#), or [Pytypes](#) for some examples.

Another use of type hints is for translating your Python code to C and compiling it for optimization. The popular [Cython project](#) uses a hybrid C/Python language to write statically typed Python code. However, since version 0.27 Cython has also supported type annotations. Recently, the [Mypyc project](#) has become available. While not yet ready for general use, it can compile some type annotated Python code to C extensions.

Conclusion

Type hinting in Python is a very useful feature that you can happily live without. Type hints don't make you capable of writing any code you can't write without using type hints. Instead, using type hints makes it easier for you to reason about code, find subtle bugs, and maintain a clean architecture.

In this tutorial you have learned how type hinting works in Python, and how gradual typing makes type checks in Python more flexible than in many other languages. You've seen some of the pros and cons of using type hints, and how they can be added to code using annotations or type comments. Finally you saw many of the different types that Python supports, as well as how to perform static type checking.

There are many resources to learn more about static type checking in Python. [PEP 483](#) and [PEP 484](#) give a lot of background about how type checking is implemented in Python. The [Mypy documentation](#) has a great [reference section](#) detailing all the different types available.

[SECTION](#) detailing all the different types available.

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Python Type Checking](#)

About Geir Arne Hjelle

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

[» More about Geir Arne](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

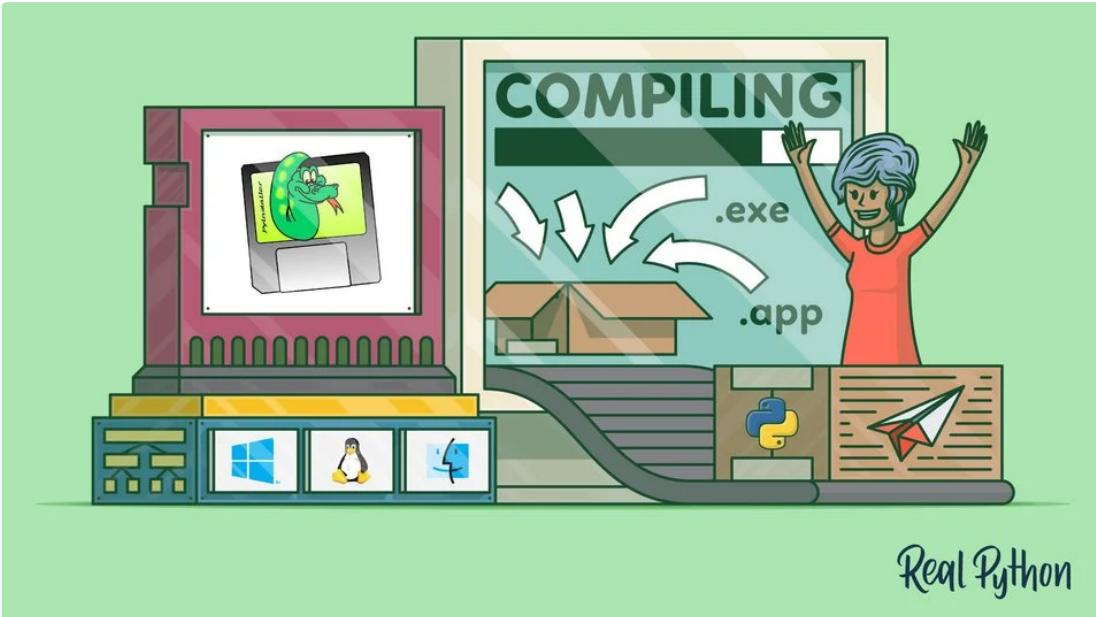
[Brad](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#)

Recommended Video Course: [Python Type Checking](#)



Real Python

Using PyInstaller to Easily Distribute Python Applications

by Luke Lee 20 Comments best-practices intermediate

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Distribution Problems](#)
- [PyInstaller](#)
- [Preparing Your Project](#)
- [Using PyInstaller](#)
- [Digging Into PyInstaller Artifacts](#)
 - [Spec File](#)
 - [Build Folder](#)
 - [Dist Folder](#)
- [Customizing Your Builds](#)
- [Testing Your New Executable](#)
- [Debugging PyInstaller Executables](#)
 - [Use the Terminal](#)
 - [Debug Files](#)
 - [Single Directory Builds](#)
 - [Additional CLI Options](#)
 - [Additional PyInstaller Docs](#)
 - [Assisting in Dependency Detection](#)
- [Limitations](#)
- [Conclusion](#)



Are you jealous of [Go](#) developers building an executable and easily shipping it to users? Wouldn't it be great if your users could **run your application without installing anything?** That is the dream, and [PyInstaller](#) is one way to get there in the Python ecosystem.

There are countless tutorials on how to [set up virtual environments](#), [manage dependencies](#), and [publish to PyPi](#), which is useful when you're creating Python libraries. There is much less information for **developers building**

Python applications. This tutorial is for developers who want to distribute applications to users who may or may not be Python developers.

In this tutorial, you'll learn the following:

- How PyInstaller can simplify application distribution
- How to use PyInstaller on your own projects
- How to debug PyInstaller errors
- What PyInstaller can't do

[PyInstaller](#) gives you the ability to create a folder or executable that users can immediately run without any extra installation. To fully appreciate PyInstaller's power, it's useful to revisit some of the distribution problems PyInstaller helps you avoid.

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

[Remove ads](#)

Distribution Problems

Setting up a Python project can be frustrating, especially for non-developers. Often, the setup starts with opening a Terminal, which is a non-starter for a huge group of potential users. This roadblock stops users even before the installation guide delves into the complicated details of virtual environments, Python versions, and the myriad of potential dependencies.

Think about what you typically go through when setting up a new machine for Python development. It probably goes something like this:

- Download and install a specific version of Python
- Set up pip
- Set up a virtual environment
- Get a copy of your code
- Install dependencies

Stop for a moment and consider if any of the above steps make any sense if you're not a developer, let alone a Python developer. Probably not.

These problems explode if your user is lucky enough to get to the dependencies portion of the installation. This has gotten much better in the last few years with the prevalence of wheels, but some dependencies still require C/C++ or even FORTRAN compilers!

This barrier to entry is way too high if your goal is to make an application available to as many users as possible. As [Raymond Hettinger](#) often says in his [excellent talks](#), "There has to be a better way."

PyInstaller

[PyInstaller](#) abstracts these details from the user by finding all your dependencies and bundling them together. Your users won't even know they're running a [Python project](#) because the Python Interpreter itself is bundled into your application. Goodbye complicated installation instructions!

PyInstaller performs this amazing feat by [introspecting](#) your Python code, detecting your dependencies, and then packaging them into a suitable format depending on your Operating System.

There are lots of interesting details about PyInstaller, but for now you'll learn the basics of how it works and how to use it. You can always refer to the [excellent PyInstaller docs](#) if you want more details.

In addition, PyInstaller can create executables for Windows, Linux, or macOS. This means Windows users will get a .exe, Linux users get a regular executable, and macOS users get a .app bundle. There are some caveats to this. See the [limitations](#) section for more information.

Preparing Your Project

PyInstaller requires your application to conform to some minimal structure, namely that you have a CLI script to start your application. Often, this means creating a small script *outside* of your Python package that simply imports your package and runs `main()`.

The entry-point script is a Python script. You can technically do anything you want in the entry-point script, but you should avoid using [explicit relative imports](#). You can still use relative imports throughout the rest of your application if that's your preferred style.

Note: An entry-point is the code that starts your project or application.

You can give this a try with your own project or follow along with the [Real Python feed reader project](#). For more detailed information on the [reader project](#), check out the the tutorial on [Publishing a Package on PyPI](#).

The first step to building an executable version of this project is to add the entry-point script. Luckily, the feed reader project is well structured, so all you need is a short script *outside* the package to run it. For example, you can create a file called `cli.py` alongside the `reader` package with the following code:

Python

```
from reader.__main__ import main

if __name__ == '__main__':
    main()
```

This `cli.py` script calls `main()` to start up the feed reader.

Creating this entry-point script is straightforward when you're working on your own project because you're familiar with the code. However, it's not as easy to find the entry-point of another person's code. In this case, you can start by looking at the `setup.py` file in the third-party project.

Look for a reference to the `entry_points` argument in the project's `setup.py`. For example, here's the `reader` project's `setup.py`:

Python

```
setup(
    name="realpython-reader",
    version="1.0.0",
    description="Read the latest Real Python tutorials",
    long_description=README,
    long_description_content_type="text/markdown",
    url="https://github.com/realpython/reader",
    author="Real Python",
    author_email="office@realpython.com",
    license="MIT",
```

```

classifiers=[
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python",
    "Programming Language :: Python :: 2",
    "Programming Language :: Python :: 3",
],
packages=["reader"],
include_package_data=True,
install_requires=[
    "feedparser", "html2text", "importlib_resources", "typing"
],
entry_points={"console_scripts": ["realpython=reader.__main__:main"]},
)

```

As you can see, the entry-point `cli.py` script calls the same function mentioned in the `entry_points` argument.

After this change, the reader project directory should look like this, assuming you checked it out into a folder called `reader`:

```

reader/
|
└── reader/
    ├── __init__.py
    ├── __main__.py
    ├── config.cfg
    ├── feed.py
    └── viewer.py
|
├── cli.py
├── LICENSE
├── MANIFEST.in
├── README.md
└── setup.py
└── tests

```

Notice there is no change to the reader code itself, just a new file called `cli.py`. This entry-point script is usually all that's necessary to use your project with PyInstaller.

However, you'll also want to look out for uses of `__import__()` or imports inside of functions. These are referred to as [hidden imports](#) in PyInstaller terminology.

You can manually specify the hidden imports to force PyInstaller to include those dependencies if changing the imports in your application is too difficult. You'll see how to do this later in this tutorial.

Once you can launch your application with a Python script *outside* of your package, you're ready to give PyInstaller a try at creating an executable.

[Remove ads](#)

Using PyInstaller

The first step is to install PyInstaller from [PyPI](#). You can do this using pip like other [Python packages](#):

Shell

```
$ pip install pyinstaller
```

pip will install PyInstaller's dependencies along with a new command: `pyinstaller`. PyInstaller can be imported in your Python code and used as a library, but you'll likely only use it as a CLI tool.

You'll use the library interface if you [create your own hook files](#).

You'll increase the likelihood of PyInstaller's defaults creating an executable if you only have pure Python dependencies. However, don't stress too much if you have more complicated dependencies with C/C++ extensions.

PyInstaller supports lots of popular packages like [NumPy](#), [PyQt](#), and [Matplotlib](#) without any additional work from you. You can see more about the list of packages that PyInstaller officially supports by referring to the [PyInstaller documentation](#).

Don't worry if some of your dependencies aren't listed in the official docs. Many Python packages work fine. In fact, PyInstaller is popular enough that many projects have explanations on how to get things working with PyInstaller.

In short, the chances of your project working out of the box are high.

To try creating an executable with all the defaults, simply give PyInstaller the name of your main entry-point script.

First, cd in the folder with your entry-point and pass it as an argument to the `pyinstaller` command that was added to your PATH when PyInstaller was installed.

For example, type the following after you cd into the top-level reader directory if you're following along with the feed reader project:

Shell

```
$ pyinstaller cli.py
```

Don't be alarmed if you see a lot of output while building your executable. PyInstaller is verbose by default, and the verbosity can be cranked way up for debugging, which you'll see later.

Digging Into PyInstaller Artifacts

PyInstaller is complicated under the hood and will create a lot of output. So, it's important to know what to focus on first. Namely, the executable you can distribute to your users and potential debugging information. By default, the `pyinstaller` command will create a few things of interest:

- A `*.spec` file
- A `build/` folder
- A `dist/` folder

Spec File

The spec file will be named after your CLI script by default. Sticking with our previous example, you'll see a file called `cli.spec`. Here's what the default spec file looks like after running PyInstaller on the `cli.py` file:

Python

```
# -*- mode: python -*-

block_cipher = None

a = Analysis(['cli.py'],
             pathex=['/Users/realpython/pyinstaller/reader'],
             binaries=[],
             datas=[],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)
exe = EXE(pyz,
          icon='reader/icon/icon.ico',
          name='reader',
          version='0.1',
          compress=True,
          strip=False,
          upx=True,
          upx_exclude=[],
          name_mangler=None,
          compress_level=9,
          upx_copyright='Copyright © 2017 Real Python. All rights reserved.')  
# Excluded from UPX
```

```
    a.scripts,
    [],
    exclude_binaries=True,
    name='cli',
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=True,
    console=True )
coll = COLLECT(exe,
                a.binaries,
                a.zipfiles,
                a.datas,
                strip=False,
                upx=True,
                name='cli')
```

This file will be automatically created by the `pyinstaller` command. Your version will have different paths, but the majority should be the same.

Don't worry, you don't need to understand the above code to effectively use PyInstaller!

This file can be modified and re-used to create executables later. You can make future builds a bit faster by providing this spec file instead of the entry-point script to the `pyinstaller` command.

There are a few [specific use-cases for PyInstaller spec files](#). However, for simple projects, you won't need to worry about those details unless you want to heavily customize how your project is built.

[Remove ads](#)

Build Folder

The `build/` folder is where PyInstaller puts most of the metadata and internal bookkeeping for building your executable. The default contents will look something like this:

```
build/
|
└── cli/
    ├── Analysis-00.toc
    ├── base_library.zip
    ├── COLLECT-00.toc
    ├── EXE-00.toc
    ├── PKG-00.pkg
    ├── PKG-00.toc
    ├── PYZ-00.pyz
    ├── PYZ-00.toc
    ├── warn-cli.txt
    └── xref-cli.html
```

The build folder can be useful for debugging, but unless you have problems, this folder can largely be ignored. You'll learn more about debugging later in this tutorial.

Dist Folder

After building, you'll end up with a `dist/` folder similar to the following:

```
dist/
|__ cli/
    __ cli
```

The `dist/` folder contains the final artifact you'll want to ship to your users. Inside the `dist/` folder, there is a folder named after your entry-point. So in this example, you'll have a `dist/cli` folder that contains all the dependencies and executable for our application. The executable to run is `dist/cli/cli` or `dist/cli/cli.exe` if you're on Windows.

You'll also find lots of files with the extension `.so`, `.pyd`, and `.dll` depending on your Operating System. These are the shared libraries that represent the dependencies of your project that PyInstaller created and collected.

Note: You can add `*.spec`, `build/`, and `dist/` to your `.gitignore` file to keep `git status` clean if you're using `git` for version control. The [default GitHub gitignore file for Python projects](#) already does this for you.

You'll want to distribute the entire `dist/cli` folder, but you can rename `cli` to anything that suits you.

At this point you can try running the `dist/cli/cli` executable if you're following along with the feed reader example.

You'll notice that running the executable results in errors mentioning the `version.txt` file. This is because the feed reader and its dependencies require some extra data files that PyInstaller doesn't know about. To fix that, you'll have to tell PyInstaller that `version.txt` is required, which you'll learn about when [testing your new executable](#).

Customizing Your Builds

PyInstaller comes with lots of options that can be provided as spec files or normal CLI options. Below, you'll find some of the most common and useful options.

`--name`

Change the name of your executable.

This is a way to avoid your executable, spec file, and build artifact folders being named after your entry-point script. `--name` is useful if you have a habit of naming your entry-point script something like `cli.py`, as I do.

You can build an executable called `realpython` from the `cli.py` script with a command like this:

Shell

```
$ pyinstaller cli.py --name realpython
```

`--onefile`

Package your entire application into a single executable file.

The default options create a folder of dependencies *and* an executable, whereas `--onefile` keeps distribution easier by creating *only* an executable.

This option takes no arguments. To bundle your project into a single file, you can build with a command like this:

Shell

```
$ pyinstaller cli.py --onefile
```

With the above command, your `dist/` folder will only contain a single executable instead of a folder with all the dependencies in separate files.

--hidden-import

List multiple top-level imports that PyInstaller was unable to detect automatically.

This is one way to work around your code using `import` inside functions and `__import__()`. You can also use `--hidden-import` multiple times in the same command.

This option requires the name of the package that you want to include in your executable. For example, if your project imported the [requests](#) library inside of a function, then PyInstaller would not automatically include requests in your executable. You could use the following command to force requests to be included:

Shell

```
$ pyinstaller cli.py --hiddenimport=requests
```

You can specify this multiple times in your build command, once for each hidden import.

--add-data and --add-binary

Instruct PyInstaller to insert additional data or binary files into your build.

This is useful when you want to bundle in configuration files, examples, or other non-code data. You'll see an example of this later if you're following along with the feed reader project.

--exclude-module

Exclude some modules from being included with your executable

This is useful to exclude developer-only requirements like testing frameworks. This is a great way to keep the artifact you give users as small as possible. For example, if you use [pytest](#), you may want to exclude this from your executable:

Shell

```
$ pyinstaller cli.py --exclude-module=pytest
```

-w

Avoid automatically opening a console window for stdout logging.

This is only useful if you're building a GUI-enabled application. This helps your hide the details of your implementation by allowing users to never see a terminal.

Similar to the `--onefile` option, `-w` takes no arguments:

Shell

```
$ pyinstaller cli.py -w
```

.spec file

As mentioned earlier, you can reuse the automatically generated `.spec` file to further customize your executable. The `.spec` file is a regular Python script that implicitly uses the PyInstaller library API.

Since it's a regular Python script, you can do almost anything inside of it. You can refer to the [official PyInstaller Spec file documentation](#) for more information on that API.

[Remove ads](#)

Testing Your New Executable

The best way to test your new executable is on a new machine. The new machine should have the same OS as your build machine. Ideally, this machine should be as similar as possible to what your users use. That may not always be possible, so the next best thing is testing on your own machine.

The key is to run the resulting executable *without* your development environment activated. This means run without virtualenv, conda, or any other **environment** that can access your Python installation. Remember, one of the main goals of a PyInstaller-created executable is for users to not need anything installed on their machine.

Picking up with the feed reader example, you'll notice that running the default cli executable in the dist/cli folder fails. Luckily the error points you to the problem:

Shell

```
FileNotFoundException: 'version.txt' resource not found in 'importlib_resources'  
[15110] Failed to execute script cli
```

The importlib_resources package requires a version.txt file. You can add this file to the build using the --add-data option. Here's an example of how to include the required version.txt file:

Shell

```
$ pyinstaller cli.py \
    --add-data venv/reader/lib/python3.6/site-packages/importlib_resources/version.txt:importlib_res
```

This command tells PyInstaller to include the version.txt file in the importlib_resources folder in a new folder in your build called importlib_resources.

Note: The pyinstaller commands use the \ character to make the command easier to read. You can omit the \ when running commands on your own or copy and paste the commands as-is below provided you're using the same paths.

You'll want to adjust the path in the above command to match where you installed the feed reader dependencies.

Now running the new executable will result in a new error about a config.cfg file.

This file is required by the feed reader project, so you'll need to make sure to include it in your build:

Shell

```
$ pyinstaller cli.py \
    --add-data venv/reader/lib/python3.6/site-packages/importlib_resources/version.txt:importlib_res
    --add-data reader/config.cfg:reader
```

Again, you'll need to adjust the path to the file based on where you have the feed reader project.

At this point, you should have a working executable that can be given directly to users!

DEBBUGGING + YOUR FIRST EXECUTABLES

As you saw above, you might encounter problems when running your executable. Depending on the complexity of your project, the fixes could be as simple as including data files like the feed reader example. However, sometimes you need more debugging techniques.

Below are a few common strategies that are in no particular order. Often times one of these strategies or a combination will lead to a break-through in tough debugging sessions.

Use the Terminal

First, try running the executable from a terminal so you can see all the output.

Remember to remove the `-w` build flag to see all the stdout in a console window. Often, you'll see `ImportError` exceptions if a dependency is missing.

[Remove ads](#)

Debug Files

Inspect the `build/cli/warn-cli.txt` file for any problems. PyInstaller creates *lots* of output to help you understand exactly what it's creating. Digging around in the `build/` folder is a great place to start.

Single Directory Builds

Use the `--onedir` distribution mode of creating distribution folder instead of a single executable. Again, this is the default mode. Building with `--onedir` gives you the opportunity to inspect all the dependencies included instead of everything being hidden in a single executable.

`--onedir` is useful for debugging, but `--onefile` is typically easier for users to comprehend. After debugging you may want to switch to `--onefile` mode to simplify distribution.

Additional CLI Options

PyInstaller also has options to control the amount of information printed during the build process. Rebuild the executable with the `--log-level=DEBUG` option to PyInstaller and review the output.

PyInstaller will create *a lot* of output when increasing the verbosity with `--log-level=DEBUG`. It's useful to save this output to a file you can refer to later instead of scrolling in your Terminal. To do this, you can use your shell's [redirection functionality](#). Here's an example:

Shell

```
$ pyinstaller --log-level=DEBUG cli.py 2> build.txt
```

By using the above command, you'll have a file called `build.txt` containing lots of additional `DEBUG` messages.

Note: The standard redirection with `>` is not sufficient. PyInstaller prints to the `stderr` stream, *not* `stdout`. This means you need to redirect the `stderr` stream to a file, which can be done using a `2` as in the previous command.

Here's a sample of what your `build.txt` file might look like:

Text

```
67 INFO: PyInstaller: 3.4
67 INFO: Python: 3.6.6
73 INFO: Platform: Darwin-18.2.0-x86_64-i386-64bit
74 INFO: wrote /Users/realpython/pyinstaller/reader/cli.spec
74 DEBUG: Testing for UPX ...
77 INFO: UPX is not available.
--
```

```
78 DEBUG: script: /users/realpython/pyinstaller/reader/c11.py
78 INFO: Extending PYTHONPATH with paths
['/Users/realpython/pyinstaller/reader',
 '/Users/realpython/pyinstaller/reader']
```

This file will have a lot of detailed information about what was included in your build, why something was not included, and how the executable was packaged.

You can also rebuild your executable using the `--debug` option in addition to using the `--log-level` option for even more information.

Note: The `-y` and `--clean` options are useful when rebuilding, especially when initially configuring your builds or building with [Continuous Integration](#). These options remove old builds and omit the need for user input during the build process.

Additional PyInstaller Docs

The [PyInstaller GitHub Wiki](#) has lots of useful links and debugging tips. Most notably are the sections on [making sure everything is packaged correctly](#) and what to do [if things go wrong](#).

Assisting in Dependency Detection

The most common problem you'll see is `ImportError` exceptions if PyInstaller couldn't properly detect all your dependencies. As mentioned before, this can happen if you're using `__import__()`, imports inside functions, or other types of [hidden imports](#).

Many of these types of problems can be resolved by using the `--hidden-import` PyInstaller CLI option. This tells PyInstaller to include a module or package even if it doesn't automatically detect it. This is the easiest way to work around lots of [dynamic import magic](#) in your application.

Another way to work around problems is [hook files](#). These files contain additional information to help PyInstaller package up a dependency. You can write your own hooks and tell PyInstaller to use them with the `--additional-hooks-dir` CLI option.

Hook files are how PyInstaller itself works internally so you can find lots of example hook files in the [PyInstaller source code](#).

[Remove ads](#)

Limitations

PyInstaller is incredibly powerful, but it does have some limitations. Some of the limitations were discussed previously: hidden imports and relative imports in entry-point scripts.

PyInstaller supports making executables for Windows, Linux, and macOS, but it cannot [cross compile](#). Therefore, you cannot make an executable targeting one Operating System from another Operating System. So, to distribute executables for multiple types of OS, you'll need a build machine for each supported OS.

Related to the cross compile limitation, it's useful to know that PyInstaller does not technically bundle absolutely everything your application needs to run. Your executable is still dependent on the users' [glibc](#). Typically, you can work around the glibc limitation by building on the oldest version of each OS you intend to target.

For example, if you want to target a wide array of Linux machines, then you can build on an older version of [CentOS](#). This will give you compatibility with most versions newer than the one you build on. This is the same strategy described in [PEP 0513](#) and is what the [PyPA](#) recommends for building compatible wheels.

In fact, you might want to investigate using the [PyPA's manylinux docker image](#) for your Linux build environment. You could start with the base image then install PyInstaller along with all your dependencies and have a build image that supports most variants of Linux.

Conclusion

PyInstaller can help make complicated installation documents unnecessary. Instead, your users can simply run your executable to get started as quickly as possible. The PyInstaller workflow can be summed up by doing the following:

1. Create an entry-point script that calls your main function.
2. Install PyInstaller.
3. Run PyInstaller on your entry-point.
4. Test your new executable.
5. Ship your resulting dist/ folder to users.

Your users don't have to know what version of Python you used or that your application uses Python at all!

About Luke Lee

Luke has professionally written software for applications ranging from Python desktop and web applications to embedded C drivers for Solid State Disks. He has also spoken at PyCon, PyTexas PyArkansas, PyconDE, and meetup groups.

[» More about Luke](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

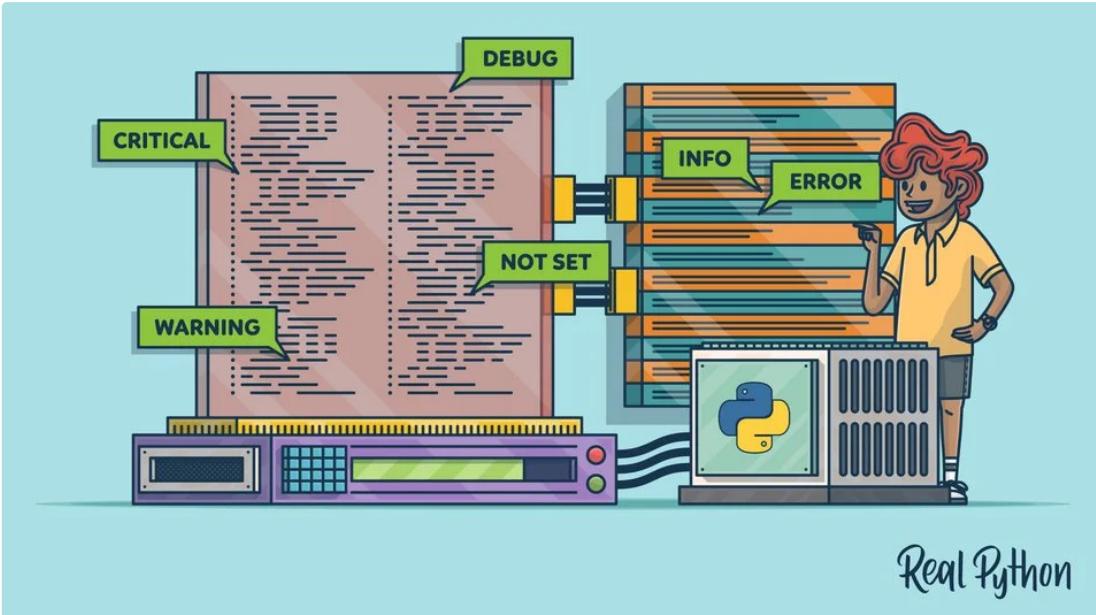
[Aldren](#)

[David](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#)



Python Logging: A Stroll Through the Source Code

by Brad Solomon 5 Comments best-practices intermediate

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [How to Follow Along](#)
- [Preliminaries](#)
 - [Preliminary #1: A Level Is Just an int!](#)
 - [Preliminary #2: Logging Is Thread-Safe, but Not Process-Safe](#)
- [Package Architecture: Logging's MRO](#)
 - [The LogRecord Class](#)
 - [The Logger and Handler Classes](#)
 - [The Filter and Filterer Classes](#)
 - [The Manager Class](#)
- [The All-Important Root Logger](#)
 - [The Logger Hierarchy](#)
 - [A Multi-Handler Design](#)
- [The “Why Didn’t My Log Message Go Anywhere?” Dilemma](#)
- [Taking Advantage of Lazy Formatting](#)
- [Functions vs Methods](#)
- [What Does getLogger\(\) Really Do?](#)
- [Library vs Application Logging: What Is NullHandler?](#)
- [What Logging Does With Exceptions](#)
- [Logging Python Tracebacks](#)
- [Conclusion](#)



The Python logging package is a lightweight but extensible package for keeping better track of what your own code does. Using it gives you much more flexibility than just littering your code with superfluous `print()` calls.

However, Python’s logging package can be complicated in certain spots. Handlers, loggers, levels, namespaces, filters: it’s not easy to keep track of all of these pieces and how they interact.

One way to tie up the loose ends in your understanding of logging is to peek under the hood to its CPython source code. The Python code behind logging is concise and modular, and reading through it can help you get that *aha* moment.

This article is meant to complement the logging [HOWTO](#) document as well as [Logging in Python](#), which is a walkthrough on how to use the package.

By the end of this article, you'll be familiar with the following:

- logging levels and how they work
- Thread-safety versus process-safety in logging
- The design of logging from an OOP perspective
- Logging in libraries vs applications
- Best practices and design patterns for using logging

For the most part, we'll go line-by-line down the core module in Python's logging package in order to build a picture of how it's laid out.

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

How to Follow Along

Because the logging source code is central to this article, you can assume that any code block or link is based on a specific commit in the Python 3.7 CPython repository, namely [commit d730719](#). You can find the logging package itself in the [Lib/](#) directory within the CPython source.

Within the logging package, most of the heavy lifting occurs within [logging/_init_.py](#), which is the file you'll spend the most time on here:

```
cpython/
|
└── Lib/
    ├── logging/
    │   ├── __init__.py
    │   ├── config.py
    │   └── handlers.py
    │   ...
    └── Modules/
    └── Include/
    ...
... [truncated]
```

With that, let's jump in.

Preliminaries

Before we get to the heavyweight classes, the top hundred lines or so of [__init__.py](#) introduce a few subtle but important concepts.

Preliminary #1: A Level Is Just an int!

Objects like `logging.INFO` or `logging.DEBUG` can seem a bit opaque. What are these variables internally, and how are they defined?

In fact, the uppercase constants from Python's logging [are just integers](#), forming an enum-like collection of numerical levels:

Python

```
CRITICAL = 50
FATAL = CRITICAL
```

```
ERROR = 40
WARNING = 30
WARN = WARNING
INFO = 20
DEBUG = 10
NOTSET = 0
```

Why not just use the strings "INFO" or "DEBUG"? Levels are `int` constants to allow for the simple, unambiguous comparison of one level with another. They are given names as well to lend them semantic meaning. Saying that a message has a severity of 50 may not be immediately clear, but saying that it has a level of `CRITICAL` lets you know that you've got a flashing red light somewhere in your program.

Now, technically, you can pass just the `str` form of a level in some places, such as `logger.setLevel("DEBUG")`. Internally, this will call `_checkLevel()`, which ultimately does a dict lookup for the corresponding `int`:

Python

```
_nameToLevel = {
    'CRITICAL': CRITICAL,
    'FATAL': FATAL,
    'ERROR': ERROR,
    'WARN': WARNING,
    'WARNING': WARNING,
    'INFO': INFO,
    'DEBUG': DEBUG,
    'NOTSET': NOTSET,
}

def _checkLevel(level):
    if isinstance(level, int):
        rv = level
    elif str(level) == level:
        if level not in _nameToLevel:
            raise ValueError("Unknown level: %r" % level)
        rv = _nameToLevel[level]
    else:
        raise TypeError("Level not an integer or a valid string: %r" % level)
    return rv
```

Which should you prefer? I'm not too opinionated on this, but it's notable that the logging docs consistently use the form `logging.DEBUG` rather than "DEBUG" or 10. Also, passing the `str` form isn't an option in Python 2, and some logging methods such as `logger.isEnabledFor()` will accept only an `int`, not its `str` cousin.

Preliminary #2: Logging Is Thread-Safe, but Not Process-Safe

A few lines down, you'll find the following short [code block](#), which is sneakily critical to the whole package:

Python

```
import threading

_lock = threading.RLock()

def _acquireLock():
    if _lock:
        _lock.acquire()

def _releaseLock():
    if _lock:
```

```
_lock.release()
```

The `_lock` object is a [reentrant lock](#) that sits in the global namespace of the `logging/__init__.py` module. It makes pretty much every object and operation in the entire logging package thread-safe, enabling threads to do read and write operations without the threat of a race condition. You can see in the module source code that `_acquireLock()` and `_releaseLock()` are ubiquitous to the module and its classes.

There's something not accounted for here, though: what about process safety? The short answer is that the `logging` module is *not* process safe. This isn't inherently a fault of `logging`—generally, two processes can't write to same file without a lot of proactive effort on behalf of the programmer first.

This means that you'll want to be careful before using classes such as `logging.FileHandler` with `multiprocessing` involved. If two processes want to read from and write to the same underlying file concurrently, then you can run into a nasty bug halfway through a long-running routine.

If you want to get around this limitation, there's a thorough [recipe](#) in the official Logging Cookbook. Because this entails a decent amount of setup, one alternative is to have each process log to a separate file based on its process ID, which you can grab with `os.getpid()`.

Package Architecture: Logging's MRO

Now that we've covered some preliminary setup code, let's take a high-level look at how `logging` is laid out. The `logging` package uses a healthy dose of [OOP](#) and [inheritance](#). Here's a partial look at the method resolution order (MRO) for some of the most important classes in the package:

```
object
|
+-- LogRecord
+-- Filterer
|   +-- Logger
|   |   +-- RootLogger
|   +-- Handler
|       +-- StreamHandler
|       +-- NullHandler
+-- Filter
+-- Manager
```

The tree diagram above doesn't cover all of the classes in the module, just those that are most worth highlighting.

Note: You can use the dunder attribute `logging.StreamHandler.__mro__` to see the chain of inheritance. A definitive guide to the MRO can be found in the Python 2 docs, though it is applicable to Python 3 as well.

This litany of classes is typically one source of confusion because there's a lot going on, and it's all jargon-heavy. `Filter` versus `Filterer`? `Logger` versus `Handler`? It can be challenging to keep track of everything, much less visualize how it fits together. A picture is worth a thousand words, so here's a diagram of a scenario where one logger with two handlers attached to it writes a log message with level `logging.INFO`:

In Python code, everything above would look like this:

Python

```
import logging
import sys

logger = logging.getLogger("pylog")
logger.setLevel(logging.DEBUG)
h1 = logging.FileHandler(filename="/tmp/records.log")
h1.setLevel(logging.INFO)
h2 = logging.StreamHandler(sys.stderr)
h2.setLevel(logging.ERROR)
logger.addHandler(h1)
logger.addHandler(h2)
logger.info("testing %d.. %d.. %d..", 1, 2, 3)
```

There's a more detailed map of this flow in the [Logging HOWTO](#). What's shown above is a simplified scenario.

Your code defines just one Logger instance, `logger`, along with two Handler instances, `h1` and `h2`.

When you call `logger.info("testing %d.. %d.. %d..", 1, 2, 3)`, the `logger` object serves as a filter because it also has a level associated with it. Only if the message level is severe enough will the logger do anything with the message. Because the logger has level `DEBUG`, and the message carries a higher `INFO` level, it gets the go-ahead to move on.

Internally, `logger` calls [`logger.makeRecord\(\)`](#) to put the message string "testing %d.. %d.. %d.." and its arguments `(1, 2, 3)` into a bona fide class instance of a `LogRecord`, which is just a container for the message and its metadata.

The `logger` object looks around for its handlers (instances of `Handler`), which may be tied directly to `logger` itself or to its parents (a concept that we'll touch on later). In this example, it finds two handlers:

1. One with level `INFO` that dumps log data to a file at `/tmp/records.log`
2. One that writes to `sys.stderr` but only if the incoming message is at level `ERROR` or higher

At this point, there's another round of tests that kicks in. Because the `LogRecord` and its message only carry level `INFO`, the record gets written to Handler 1 (green arrow), but not to Handler 2's `stderr` stream (red arrow). For Handlers, writing the `LogRecord` to their stream is called **emitting** it, which is captured in their [`.emit\(\)`](#).

Next, let's further dissect everything from above.

The LogRecord Class

What is a `LogRecord`? When you log a message, an instance of the `LogRecord` class is the object you send to be logged. It's created for you by a `Logger` instance and encapsulates all the pertinent info about that event. Internally, it's little more than a wrapper around a `dict` that contains attributes for the record. A `Logger` instance sends a `LogRecord` instance to zero or more `Handler` instances.

The `LogRecord` contains some metadata, such as the following:

1. A name
2. The creation time as a Unix timestamp
3. The message itself
4. Information on what function made the logging call

Here's a peek into the metadata that it carries with it, which you can introspect by stepping through a `logging.error()` call with the [pdb module](#):

Python

>>>

```
>>> import logging
>>> import pdb

>>> def f(x):
...     logging.error("bad vibes")
...     return x / 0
```

```
...  
=> pdb.run("f(1)")
```

After stepping through some higher-level functions, you end up [at line 1517](#):

Shell

```
(Pdb) 1  
1514         exc_info = (type(exc_info), exc_info, exc_info.__traceback__)  
1515     elif not isinstance(exc_info, tuple):  
1516         exc_info = sys.exc_info()  
1517     record = self.makeRecord(self.name, level, fn, lno, msg, args,  
1518                             exc_info, func, extra, sinfo)  
1519 ->     self.handle(record)  
1520  
1521     def handle(self, record):  
1522         """  
1523             Call the handlers for the specified record.  
1524  
1525         from pprint import pprint  
1526         pprint(vars(record))  
{'args': (),  
 'created': 1550671851.660067,  
 'exc_info': None,  
 'exc_text': None,  
 'filename': '<stdin>',  
 'funcName': 'f',  
 'levelname': 'ERROR',  
 'levelno': 40,  
 'lineno': 2,  
 'module': '<stdin>',  
 'msecs': 660.067081451416,  
 'msg': 'bad vibes',  
 'name': 'root',  
 'pathname': '<stdin>',  
 'process': 2360,  
 'processName': 'MainProcess',  
 'relativeCreated': 295145.5490589142,  
 'stack_info': None,  
 'thread': 4372293056,  
 'threadName': 'MainThread'}
```

A LogRecord, internally, contains a trove of metadata that's used in one way or another.

You'll rarely need to deal with a LogRecord directly, since the Logger and Handler do this for you. It's still worthwhile to know what information is wrapped up in a LogRecord, because this is where all that useful info, like the timestamp, come from when you see record log messages.

Note: Below the LogRecord class, you'll also find the `setLogRecordFactory()`, `getLogRecordFactory()`, and `makeLogRecord()` [factory functions](#). You won't need these unless you want to use a custom class instead of LogRecord to encapsulate log messages and their metadata.

The Logger and Handler Classes

The Logger and Handler classes are both central to how logging works, and they interact with each other frequently. A Logger, a Handler, and a LogRecord each have a `.level` associated with them.

The Logger takes the LogRecord and passes it off to the Handler, but only if the effective level of the LogRecord is equal to or higher than that of the Logger. The same goes for the LogRecord versus Handler test. This is called **level-based filtering**, which Logger and Handler implement in slightly different ways.

In other words, there is an (at least) two-step test applied before the message that you log gets to go anywhere. In order to be fully passed from a logger to handler and then logged to the end stream (which could be `sys.stdout`, a file, or an email via SMTP), a LogRecord must have a level at least as high as *both* the logger and handler.

PEP 282 describes how this works:

Each `Logger` object keeps track of a log level (or threshold) that it is interested in, and discards log requests below that level. ([Source](#))

So where does this level-based filtering actually occur for both `Logger` and `Handler`?

For the `Logger` class, it's a reasonable first assumption that the logger would compare its `.level` attribute to the level of the `LogRecord`, and be done there. However, it's slightly more involved than that.

Level-based filtering for loggers occurs in `.isEnabledFor()`, which in turn calls `.getEffectiveLevel()`. Always use `logger.getEffectiveLevel()` rather than just consulting `logger.level`. The reason has to do with the organization of `Logger` objects in a hierarchical namespace. (You'll see more on this later.)

By default, a `Logger` instance has a level of 0 (NOTSET). However, loggers also have **parent loggers**, one of which is the root logger, which functions as the parent of all other loggers. A `Logger` will walk upwards in its hierarchy and get its effective level vis-à-vis its parent (which ultimately may be root if no other parents are found).

Here's [where this happens](#) in the `Logger` class:

Python

```
class Logger(Filterer):
    ...
    def getEffectiveLevel(self):
        logger = self
        while logger:
            if logger.level:
                return logger.level
            logger = logger.parent
        return NOTSET

    def isEnabledFor(self, level):
        try:
            return self._cache[level]
        except KeyError:
            _acquireLock()
            if self.manager.disable >= level:
                is_enabled = self._cache[level] = False
            else:
                is_enabled = self._cache[level] = level >= self.getEffectiveLevel()
            _releaseLock()
        return is_enabled
```

Correspondingly, here's an example that calls the source code you see above:

Python

>>>

```
>>> import logging
>>> logger = logging.getLogger("app")
>>> logger.level # No!
0
>>> logger.getEffectiveLevel()
30
>>> logger.parent
<RootLogger root (WARNING)>
>>> logger.parent.level
30
```

Here's the takeaway: don't rely on `.level`. If you haven't explicitly set a level on your `logger` object, and you're depending on `.level` for some reason, then your logging setup will likely behave differently than you expected it to.

What about `Handler`? For handlers, the level-to-level comparison is simpler, though it actually happens [in `.callHandlers\(\)`](#) from the `Logger` class:

Python

```
class Logger(Filterer):
    ...
    def callHandlers(self, record):
        c = self
        found = 0
        while c:
            for hdlr in c.handlers:
```

```
    found = found + 1
    if record.levelno >= hdlr.level:
        hdlr.handle(record)
```

For a given `LogRecord` instance (named `record` in the source code above), a logger checks with each of its registered handlers and does a quick check on the `.level` attribute of that `Handler` instance. If the `.levelno` of the `LogRecord` is greater than or equal to that of the handler, only then does the record get passed on. A [docstring](#) in `logging` refers to this as “conditionally emit[ting] the specified logging record.”

The Filter and Filterer Classes

Above, we asked the question, “Where does level-based filtering happen?” In answering this question, it’s easy to get distracted by the `Filter` and `Filterer` classes. Paradoxically, level-based filtering for `Logger` and `Handler` instances occurs without the help of either of the `Filter` or `Filterer` classes.

`Filter` and `Filterer` are designed to let you add additional function-based filters on top of the level-based filtering that is done by default. I like to think of it as *à la carte* filtering.

`Filterer` is the base class for `Logger` and `Handler` because both of these classes are eligible for receiving additional custom filters that you specify. You add instances of `Filter` to them with `logger.addFilter()` or `handler.addFilter()`, which is what `self.filters` refers to in the following method:

Python

```
class Filterer(object):
    ...
    def filter(self, record):
        rv = True
        for f in self.filters:
            if hasattr(f, 'filter'):
                result = f.filter(record)
            else:
                result = f(record)
            if not result:
                rv = False
                break
        return rv
```

Given a `record` (which is a `LogRecord` instance), `.filter()` returns `True` or `False` depending on whether that record gets the okay from this class’s filters.

Here is `.handle()` in turn, for the `Logger` and `Handler` classes:

Python

```
class Logger(Filterer):
    ...
    def handle(self, record):
        if (not self.disabled) and self.filter(record):
            self.callHandlers(record)

    ...
class Handler(Filterer):
    ...
```

```

# ...
def handle(self, record):
    rv = self.filter(record)
    if rv:
        self.acquire()
        try:
            self.emit(record)
        finally:
            self.release()
    return rv

```

Neither Logger nor Handler come with any additional filters by default, but here's a quick example of how you could add one:

```

Python >>>
>>> import logging

>>> logger = logging.getLogger("rp")
>>> logger.setLevel(logging.INFO)
>>> logger.addHandler(logging.StreamHandler())
>>> logger.filters # Initially empty
[]
>>> class ShortMsgFilter(logging.Filter):
...     """Only allow records that contain long messages (> 25 chars)."""
...     def filter(self, record):
...         msg = record.msg
...         if isinstance(msg, str):
...             return len(msg) > 25
...         return False
...
>>> logger.addFilter(ShortMsgFilter())
>>> logger.filters
[<__main__.ShortMsgFilter object at 0x10c28b208>]
>>> logger.info("Reeeeeaaaalllllly long message") # Length: 31
Reeeeeaaaalllllly long message
>>> logger.info("Done") # Length: <25, no output

```

Above, you define a class `ShortMsgFilter` and override its `.filter()`. In `.addHandler()`, you could also just pass a callable, such as a function or lambda or a class that defines `__call__()`.

The Manager Class

There's one more behind-the-scenes actor of logging that is worth touching on: the Manager class. What matters most is not the Manager class but a single instance of it that acts as a container for the growing hierarchy of loggers that are defined across packages. You'll see in the next section how just a single instance of this class is central to gluing the module together and allowing its parts to talk to each other.

The All-Important Root Logger

When it comes to Logger instances, one stands out. It's called the root logger:

Python

```
class RootLogger(Logger):
    def __init__(self, level):
        Logger.__init__(self, "root", level)

    ...

root = RootLogger(WARNING)
Logger.root = root
Logger.manager = Manager(Logger.root)
```

The last three lines of this code block are one of the ingenious tricks employed by the `logging` package. Here are a few points:

- The root logger is just a no-frills Python object with the identifier `root`. It has a level of `logging.WARNING` and a `.name` of "root". As far as the class `RootLogger` is concerned, this unique name is all that's special about it.
- The `root` object in turn becomes a **class attribute** for the `Logger` class. This means that all instances of `Logger`, and the `Logger` class itself, all have a `.root` attribute that is the root logger. This is another example of a singleton-like pattern being enforced in the `logging` package.
- A `Manager` instance is set as the `.manager` class attribute for `Logger`. This eventually comes into play in `logging.getLogger("name")`. The `.manager` does all the facilitation of searching for existing loggers with the name "name" and creating them if they don't exist.

The Logger Hierarchy

Everything is a child of `root` in the logger namespace, and I mean everything. That includes loggers that you specify yourself as well as those from third-party libraries that you import.

Remember earlier how the `.getEffectiveLevel()` for our logger instances was 30 (`WARNING`) even though we had not explicitly set it? That's because the root logger sits at the top of the hierarchy, and its level is a fallback if any nested loggers have a null level of `NOTSET`:

Python

>>>

```
>>> root = logging.getLogger() # Or getLogger("")
>>> root
<RootLogger root (WARNING)>
>>> root.parent is None
True
>>> root.root is root # Self-referential
True
>>> root is logging.root
True
>>> root.getEffectiveLevel()
30
```

The same logic applies to the search for a logger's handlers. The search is effectively a reverse-order search up the tree of a logger's parents.

A Multi-Handler Design

The logger hierarchy may seem neat in theory, but how beneficial is it in practice?

Let's take a break from exploring the `logging` code and foray into writing our own mini-application—one that takes advantage of the logger hierarchy in a way that reduces boilerplate code and keeps things scalable if the project's codebase grows.

Here's the project structure:

```
project/
|
```

```
project/
└── __init__.py
└── utils.py
└── base.py
```

Don't worry about the application's main functions in `utils.py` and `base.py`. What we're paying more attention to here is the interaction in logging objects between the modules in `project/`.

In this case, say that you want to design a multipronged logging setup:

- Each module gets a logger with multiple handlers.
- Some of the handlers are shared between different logger instances in different modules. These handlers only care about level-based filtering, not the module where the log record emanated from. There is a handler for `DEBUG` messages, one for `INFO`, one for `WARNING`, and so on.
- Each logger is also tied to one more additional handler that only receives `LogRecord` instances from that lone logger. You can call this a module-based file handler.

Visually, what we're shooting for would look something like this:

A multipronged logging design (Image: Real Python)

The two turquoise objects are instances of `Logger`, established with `logging.getLogger(__name__)` for each module in a package. Everything else is a `Handler` instance.

The thinking behind this design is that it's neatly compartmentalized. You can conveniently look at messages coming from a single logger, or look at messages of a certain level and above coming from any logger or module.

The properties of the logger hierarchy make it suitable for setting up this multipronged logger-handler layout. What does that mean? Here's a concise explanation from the Django documentation:

Why is the hierarchy important? Well, because loggers can be set to propagate their logging calls to their parents. In this way, you can define a single set of handlers at the root of a logger tree, and capture all logging calls in the subtree of loggers. A logging handler defined in the project namespace will catch all logging messages issued on the `project.interesting` and `project.interesting.stuff` loggers. ([Source](#))

The term **propagate** refers to how a logger keeps walking up its chain of parents looking for handlers. The `.propagate` attribute is `True` for a `Logger` instance by default:

```
>>> logger = logging.getLogger(__name__)
>>> logger.propagate
True
```

In `.callHandlers()`, if `propagate` is `True`, each successive parent gets reassigned to the local variable `c` until the hierarchy is exhausted:

Python

```
class Logger(Filterer):
    ...
    def callHandlers(self, record):
        c = self
        found = 0
        while c:
            for hdlr in c.handlers:
                found = found + 1
                if record.levelno >= hdlr.level:
                    hdlr.handle(record)
            if not c.propagate:
                c = None
            else:
                c = c.parent
```

Here's what this means: because the `__name__` dunder variable within a package's `__init__.py` module is just the name of the package, a logger there becomes a parent to any loggers present in other modules in the same package.

Here are the resulting `.name` attributes from assigning to `logger` with `logging.getLogger(__name__)`:

Module	<code>.name</code> Attribute
<code>project/__init__.py</code>	'project'
<code>project/utils.py</code>	'project.utils'
<code>project/base.py</code>	'project.base'

Because the 'project.utils' and 'project.base' loggers are children of 'project', they will latch onto not only their own direct handlers but whatever handlers are attached to 'project'.

Let's build out the modules. First comes `__init__.py`:

Python

```
# __init__.py
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

levels = ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
for level in levels:
```

```

handler = logging.FileHandler(f"/tmp/level-{level.lower()}.log")
handler.setLevel(getattr(logging, level))
logger.addHandler(handler)

def add_module_handler(logger, level=logging.DEBUG):
    handler = logging.FileHandler(
        f"/tmp/module-{logger.name.replace('.', '-')}.log"
    )
    handler.setLevel(level)
    logger.addHandler(handler)

```

This module is imported when the project package is imported. You add a handler for each level in DEBUG through CRITICAL, then attach it to a single logger at the top of the hierarchy.

You also define a utility function that adds one more `FileHandler` to a logger, where the `filename` of the handler corresponds to the module name where the logger is defined. (This assumes the logger is defined with `__name__`.)

You can then add some minimal boilerplate logger setup in `base.py` and `utils.py`. Notice that you only need to add one additional handler with `add_module_handler()` from `__init__.py`. You don't need to worry about the level-oriented handlers because they are already added to their parent logger named 'project':

Python

```

# base.py
import logging

from project import add_module_handler

logger = logging.getLogger(__name__)
add_module_handler(logger)

def func1():
    logger.debug("debug called from base.func1()")
    logger.critical("critical called from base.func1()")

```

Here's `utils.py`:

Python

```

# utils.py
import logging

from project import add_module_handler

logger = logging.getLogger(__name__)
add_module_handler(logger)

def func2():
    logger.debug("debug called from utils.func2()")
    logger.critical("critical called from utils.func2()")

```

Let's see how all of this works together from a fresh Python session:

Python

>>>

```

>>> from pprint import pprint
>>> import project
>>> from project import base, utils

>>> project.logger
<Logger project (DEBUG)>
>>> base.logger, utils.logger
(<Logger project.base (DEBUG)>, <Logger project.utils (DEBUG)>)
>>> base.logger.handlers
[<FileHandler /tmp/module-project-base.log (DEBUG)>]
>>> pprint(base.logger.parent.handlers)
[<FileHandler /tmp/level-debug.log (DEBUG)>,
 <FileHandler /tmp/level-info.log (INFO)>,
 <FileHandler /tmp/level-warning.log (WARNING)>,
 <FileHandler /tmp/level-error.log (ERROR)>,

```

```
<FileHandler /tmp/level-critical.log (CRITICAL)>
>>> base.func1()
>>> utils.func2()
```

You'll see in the resulting log files that our filtration system works as intended. Module-oriented handlers direct one logger to a specific file, while level-oriented handlers direct multiple loggers to a different file:

Shell

```
$ cat /tmp/level-debug.log
debug called from base.func1()
critical called from base.func1()
debug called from utils.func2()
critical called from utils.func2()

$ cat /tmp/level-critical.log
critical called from base.func1()
critical called from utils.func2()

$ cat /tmp/module-project-base.log
debug called from base.func1()
critical called from base.func1()

$ cat /tmp/module-project-utils.log
debug called from utils.func2()
critical called from utils.func2()
```

A drawback worth mentioning is that this design introduces a lot of redundancy. One `LogRecord` instance may go to no less than six files. That's also a non-negligible amount of file I/O that may add up in a performance-critical application.

Now that you've seen a practical example, let's switch gears and delve into a possible source of confusion in logging.

The “Why Didn’t My Log Message Go Anywhere?” Dilemma

There are two common situations with logging when it's easy to get tripped up:

1. You logged a message that seemingly went nowhere, and you're not sure why.
2. Instead of being suppressed, a log message appeared in a place that you didn't expect it to.

Each of these has a reason or two commonly associated with it.

You logged a message that seemingly went nowhere, and you're not sure why.

Don't forget that the **effective** level of a logger for which you don't otherwise set a custom level is `WARNING`, because a logger will walk up its hierarchy until it finds the root logger with its own `WARNING` level:

Python

>>>

```
>>> import logging
>>> logger = logging.getLogger("xyz")
>>> logger.debug("mind numbing info here")
>>> logger.critical("storm is coming")
storm is coming
```

Because of this default, the `.debug()` call goes nowhere.

Instead of being suppressed, a log message appeared in a place that you didn't expect it to.

When you defined your logger above, you didn't add any handlers to it. So, why is it writing to the console?

The reason for this is that logging [sneakily uses](#) a handler called `lastResort` that writes to `sys.stderr` if no other handlers are found:

Python

```
class _StderrHandler(StreamHandler):
    ...
    @property
    def stream(self):
        return sys.stderr

_defaultLastResort = _StderrHandler(WARNING)
lastResort = _defaultLastResort
```

This kicks in when a logger goes to find its handlers:

Python

```
class Logger(Filterer):
    ...
    def callHandlers(self, record):
        c = self
        found = 0
        while c:
            for hdlr in c.handlers:
                found = found + 1
                if record.levelno >= hdlr.level:
                    hdlr.handle(record)
            if not c.propagate:
                c = None
            else:
                c = c.parent
        if (found == 0):
            if lastResort:
                if record.levelno >= lastResort.level:
                    lastResort.handle(record)
```

If the logger gives up on its search for handlers (both its own direct handlers and attributes of parent loggers), then it picks up the `lastResort` handler and uses that.

There's one more subtle detail worth knowing about. This section has largely talked about the instance methods (methods that a class defines) rather than the module-level functions of the `logging` package that carry the same name.

If you use the functions, such as `logging.info()` rather than `logger.info()`, then something slightly different happens internally. The function calls `logging.basicConfig()`, which adds a `StreamHandler` that writes to `sys.stderr`. In the end, the behavior is virtually the same:

Python

>>>

```
>>> import logging
>>> root = logging.getLogger("")
>>> root.handlers
[]
>>> root.hasHandlers()
False
>>> logging.basicConfig()
>>> root.handlers
[<StreamHandler <stderr> (NOTSET)>]
>>> root.hasHandlers()
True
```

Taking Advantage of Lazy Formatting

It's time to switch gears and take a closer look at how messages themselves are joined with their data. While it's been supplanted by `str.format()` and `f-strings`, you've probably used Python's percent-style formatting to do something like this:

Python

```
>>> print("To iterate is %s, to recurse %s" % ("human", "divine"))
To iterate is human, to recurse divine
```

>>>

As a result, you may be tempted to do the same thing in a logging call:

Python

```
>>> # Bad! Check out a more efficient alternative below.
>>> logging.warning("To iterate is %s, to recurse %s" % ("human", "divine"))
WARNING:root:To iterate is human, to recurse divine
```

>>>

This uses the entire format string and its arguments as the `msg` argument to `logging.warning()`.

Here is the recommended alternative, [straight from the logging docs](#):

Python

```
>>> # Better: formatting doesn't occur until it really needs to.
>>> logging.warning("To iterate is %s, to recurse %s", "human", "divine")
WARNING:root:To iterate is human, to recurse divine
```

>>>

It looks a little weird, right? This seems to defy the conventions of how percent-style string formatting works, but it's a more efficient function call because the format string gets formatted [lazily rather than greedily](#). Here's what that means.

The method signature for `Logger.warning()` looks like this:

Python

```
def warning(self, msg, *args, **kwargs)
```

The same applies to the other methods, such as `.debug()`. When you call `warning("To iterate is %s, to recurse %s", "human", "divine")`, both "human" and "divine" get caught as `*args` and, within the scope of the method's body, `args` is equal to `("human", "divine")`.

Contrast this to the first call above:

Python

```
logging.warning("To iterate is %s, to recurse %s" % ("human", "divine"))
```

In this form, everything in the parentheses gets immediately merged together into "To iterate is human, to recurse divine" and passed as `msg`, while `args` is an empty tuple.

Why does this matter? Repeated logging calls can degrade runtime performance slightly, but the `logging` package does its very best to control that and keep it in check. By not merging the format string with its arguments right away, `logging` is delaying the string formatting until the `LogRecord` is requested by a `Handler`.

This happens in `LogRecord.getMessage()`, so only after `logging` deems that the `LogRecord` will actually be passed to a handler does it become its fully merged self.

All that is to say that the `logging` package makes some very fine-tuned performance optimizations in the right places. This may seem like minutia, but if you're making the same `logging.debug()` call a million times inside a loop, and the args are function calls, then the lazy nature of how `logging` does string formatting can make a difference.

Before doing any merging of `msg` and `args`, a `Logger` instance will check its `.isEnabledFor()` to see if that merging should be done in the first place.

Functions vs Methods

Towards the bottom of `logging/__init__.py` sit the module-level functions that are advertised up front in the public API of `logging`. You already saw the `Logger` methods such as `.debug()`, `.info()`, and `.warning()`. The top-level functions are wrappers around the corresponding methods of the same name, but they have two important features:

1. They always call their corresponding method from the root logger, `root`.
2. Before calling the root logger methods, they call `logging.basicConfig()` with no arguments if `root` doesn't have any handlers. As you saw earlier, it is this call that sets a `sys.stdout` handler for the root logger.

For illustration, here's `logging.error()`:

Python

```
def error(msg, *args, **kwargs):
    if len(root.handlers) == 0:
        basicConfig()
    root.error(msg, *args, **kwargs)
```

You'll find the same pattern for `logging.debug()`, `logging.info()`, and the others as well. Tracing the chain of commands is interesting. Eventually, you'll end up at the same place, which is where the internal `Logger._log()` is called.

The calls to `debug()`, `info()`, `warning()`, and the other level-based functions all route to here. `_log()` primarily has two purposes:

1. **Call `self.makeRecord()`:** Make a `LogRecord` instance from the `msg` and other arguments you pass to it.
2. **Call `self.handle()`:** This determines what actually gets done with the record. Where does it get sent? Does it make it there or get filtered out?

Here's that entire process in one diagram:

A complex diagram showing the internal structure of a logging call. It features a central node labeled 'logging' which branches into 'logger' and 'manager'. 'logger' further branches into 'Handler' and 'Filter'. 'Handler' leads to 'StreamHandler' and 'FileHandler'. 'Filter' leads to 'LevelFilter'. 'manager' branches into 'dictConfig' and 'makeLogRecord'. 'dictConfig' leads to 'ConfigParser'. 'makeLogRecord' leads to 'LogRecord'.

Internals of a logging call (Image: Real Python)

You can also trace the call stack with pdb.

Tracing the Call to logging.warning()

Show/Hide

What Does getLogger() Really Do?

Also hiding in this section of the source code is the top-level `getLogger()`, which wraps `Logger.manager.getLogger()`:

Python

```
def getLogger(name=None):
    if name:
        return Logger.manager.getLogger(name)
    else:
        return root
```

This is the entry point for enforcing the singleton logger design:

- If you specify a name, then the underlying `.getLogger()` does a dict lookup on the string name. What this comes down to is a lookup in the `loggerDict` of `logging.Manager`. This is a dictionary of all registered loggers, including the intermediate `PlaceHolder` instances that are generated when you reference a logger far down in the hierarchy before referencing its parents.
- Otherwise, `root` is returned. There is only one `root`—the instance of `RootLogger` discussed above.

This feature is what lies behind a trick that can let you peek into all of the registered loggers:

Python

```
>>> import logging
```

>>>

```

>>> logging.Logger.manager.loggerDict
{}

>>> from pprint import pprint
>>> import asyncio
>>> pprint(logging.Logger.manager.loggerDict)
{'asyncio': <Logger asyncio (WARNING)>,
 'concurrent': <logging.PlaceHolder object at 0x10d153710>,
 'concurrent.futures': <Logger concurrent.futures (WARNING)>}

```

Whoa, hold on a minute. What's happening here? It looks like something changed internally to the `logging` package as a result of an import of another library, and that's exactly what happened.

Firstly, recall that `Logger.manager` is a class attribute, where an instance of `Manager` is tacked onto the `Logger` class. The `manager` is designed to track and manage all of the singleton instances of `Logger`. These are housed in `.loggerDict`.

Now, when you initially import `logging`, this dictionary is empty. But after you import `asyncio`, the same dictionary gets populated with three loggers. This is an example of one module setting the attributes of another module in-place. Sure enough, inside of `asyncio/log.py`, you'll find the following:

Python

```

import logging

logger = logging.getLogger(__package__) # "asyncio"

```

The key-value pair is `set` in `Logger.getLogger()` so that the `manager` can oversee the entire namespace of loggers. This means that the object `asyncio.log.logger` gets registered in the logger dictionary that belongs to the `logging` package. Something similar happens in the `concurrent.futures` package as well, which is imported by `asyncio`.

You can see the power of the singleton design in an equivalence test:

Python

>>>

```

>>> obj1 = logging.getLogger("asyncio")
>>> obj2 = logging.Logger.manager.loggerDict["asyncio"]
>>> obj1 is obj2
True

```

This comparison illustrates (glossing over a few details) what `getLogger()` ultimately does.

Library vs Application Logging: What Is `NullHandler`?

That brings us to the final hundred or so lines in the `logging/__init__.py` source, where `NullHandler` is defined. Here's the definition in all its glory:

Python

```

class NullHandler(Handler):
    def handle(self, record):
        pass

    def emit(self, record):
        pass

    def createLock(self):
        self.lock = None

```

The `NullHandler` is all about the distinctions between logging in a library versus an application. Let's see what that means.

A **library** is an extensible, generalizable Python package that is intended for other users to install and set up. It is built by a developer with the express purpose of being distributed to users. Examples include popular open-source projects like [NumPy](#), [dateutil](#), and [cryptography](#).

An **application** (or app, or program) is designed for a more specific purpose and a much smaller set of users

(possibly just one user). It's a program or set of programs highly tailored by the user to do a limited set of things. An example of an application is a Django app that sits behind a web page. Applications commonly use (`import`) libraries and the tools they contain.

When it comes to logging, there are different best practices in a library versus an app.

That's where `NullHandler` fits in. It's basically a do-nothing stub class.

If you're writing a Python library, you really need to do this one minimalist piece of setup in your package's `__init__.py`:

Python

```
# Place this in your library's uppermost `__init__.py`  
# Nothing else!  
  
import logging  
  
logging.getLogger(__name__).addHandler(NullHandler())
```

This serves two critical purposes.

Firstly, a library logger that is declared with `logger = logging.getLogger(__name__)` (without any further configuration) will log to `sys.stderr` by default, even if that's not what the end user wants. This could be described as an opt-out approach, where the end user of the library has to go in and disable logging to their console if they don't want it.

Common wisdom says to use an opt-in approach instead: don't emit any log messages by default, and let the end users of the library determine if they want to further configure the library's loggers and add handlers to them. Here's that philosophy worded more bluntly by the author of the `logging` package, Vinay Sajip:

A third party library which uses `logging` should not spew logging output by default which may not be wanted by a developer/user of an application which uses it. ([Source](#))

This leaves it up to the library user, not library developer, to incrementally call methods such as `logger.addHandler()` or `logger.setLevel()`.

The second reason that `NullHandler` exists is more archaic. In Python 2.7 and earlier, trying to log a `LogRecord` from a logger that has no handler set would [raise a warning](#). Adding the no-op class `NullHandler` will avert this.

Here's what specifically happens in the line `logging.getLogger(__name__).addHandler(NullHandler())` from above:

1. Python gets (creates) the `Logger` instance with the same name as your package. If you're designing the `calculus` package, within `__init__.py`, then `__name__` will be equal to '`calculus`'.
2. A `NullHandler` instance gets attached to this logger. That means that Python will not default to using the `lastResort` handler.

Keep in mind that any logger created in any of the other `.py` modules of the package will be children of this logger in the logger hierarchy and that, because this handler also belongs to them, they won't need to use the `lastResort` handler and won't default to logging to standard error (`stderr`).

As a quick example, let's say your library has the following structure:

```
calculus/  
|  
└── __init__.py  
└── integration.py
```

In `integration.py`, as the library developer you are free to do the following:

Python

```
# calculus/integration.py
```

```

import logging

logger = logging.getLogger(__name__)

def func(x):
    logger.warning("Look!")
    # Do stuff
    return None

```

Now, a user comes along and installs your library from PyPI via `pip install calculus`. They use `from calculus.integration import func` in some application code. This user is free to manipulate and configure the `logger` object from the library like any other Python object, to their heart's content.

What Logging Does With Exceptions

One thing that you may be wary of is the danger of exceptions that stem from your calls to `logging`. If you have a `logging.error()` call that is designed to give you some more verbose debugging information, but that call itself for some reason raises an exception, that would be the height of irony, right?

Cleverly, if the `logging` package encounters an exception that has to do with logging itself, then it will print the [traceback](#) but not raise the exception itself.

Here's an example that deals with a common typo: passing two arguments to a format string that is only expecting one argument. The important distinction is that what you see below is *not* an exception being raised, but rather a prettified printed traceback of the internal exception, which itself was suppressed:

```

Python >>>
>>> logging.critical("This %s has too many arguments", "msg", "other")
--- Logging error ---
Traceback (most recent call last):
  File "lib/python3.7/logging/_init_.py", line 1034, in emit
    msg = self.format(record)
  File "lib/python3.7/logging/_init_.py", line 880, in format
    return fmt.format(record)
  File "lib/python3.7/logging/_init_.py", line 619, in format
    record.message = record.getMessage()
  File "lib/python3.7/logging/_init_.py", line 380, in getMessage
    msg = msg % self.args
TypeError: not all arguments converted during string formatting
Call stack:
  File "<stdin>", line 1, in <module>
Message: 'This %s has too many arguments'
Arguments: ('msg', 'other')

```

This lets your program gracefully carry on with its actual program flow. The rationale is that you wouldn't want an uncaught exception to come from a logging call itself and stop a program dead in its tracks.

[Tracebacks](#) can be messy, but this one is informative and relatively straightforward. What enables the suppression of exceptions related to logging is `Handler.handleError()`. When the handler calls `.emit()`, which is the method where it attempts to log the record, it falls back to `.handleError()` if something goes awry. Here's the

implementation of `.emit()` for the `StreamHandler` class:

```

Python
def emit(self, record):
    try:
        msg = self.format(record)
        stream = self.stream
        stream.write(msg + self.terminator)
        self.flush()
    except Exception:
        self.handleError(record)

```

Any exception related to the formatting and writing gets caught rather than being raised, and `handleError` gracefully writes the traceback to `sys.stderr`.

Logging Python Tracebacks

Speaking of exceptions and their tracebacks, what about cases where your program encounters them but should log the exception and keep chugging along in its execution?

Let's walk through a couple of ways to do this.

Here's a contrived example of a lottery simulator using code that isn't Pythonic on purpose. You're developing an online lottery game where users can wager on their lucky number:

Python

```
import random

class Lottery(object):
    def __init__(self, n):
        self.n = n

    def make_tickets(self):
        for i in range(self.n):
            yield i

    def draw(self):
        pool = self.make_tickets()
        random.shuffle(pool)
        return next(pool)
```

Behind the frontend application sits the critical code below. You want to make sure that you keep track of any errors caused by the site that may make a user lose their money. The first (suboptimal) way is to use `logging.error()` and log the `str` form of the exception instance itself:

Python

```
try:
    lucky_number = int(input("Enter your ticket number: "))
    drawn = Lottery(n=20).draw()
    if lucky_number == drawn:
        print("Winner chicken dinner!")
except Exception as e:
    # NOTE: See below for a better way to do this.
    logging.error("Could not draw ticket: %s", e)
```

This will only get you the actual exception message, rather than the traceback. You check the logs on your website's server and find this cryptic message:

Text

```
ERROR:root:Could not draw ticket: object of type 'generator' has no len()
```

Hmm. As the application developer, you've got a serious problem, and a user got ripped off as a result. But maybe this exception message itself isn't very informative. Wouldn't it be nice to see the lineage of the traceback that led to this exception?

The proper solution is to use `logging.exception()`, which logs a message with level `ERROR` and also displays the exception traceback. Replace the two final lines above with these:

Python

```
except Exception:
    logging.exception("Could not draw ticket")
```

Now you get a better indication of what's going on:

Python

>>>

```
ERROR:root:Could not draw ticket
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "<stdin>", line 9, in draw
  File "lib/python3.7/random.py", line 275, in shuffle
```

```
for i in reversed(range(1, len(x))):  
TypeError: object of type 'generator' has no len()
```

Using `exception()` saves you from having to reference the exception yourself because logging pulls it in with `sys.exc_info()`.

This makes things clearer that the problem stems from `random.shuffle()`, which needs to know the length of the object it is shuffling. Because our `Lottery` class passes a generator to `shuffle()`, it gets held up and raises before the pool can be shuffled, much less generate a winning ticket.

In large, full-blown applications, you'll find `logging.exception()` to be even more useful when deep, multi-library tracebacks are involved, and you can't step into them with a live debugger like `pdb`.

The code for `logging.Logger.exception()`, and hence `logging.exception()`, is just a single line:

Python

```
def exception(self, msg, *args, exc_info=True, **kwargs):  
    self.error(msg, *args, exc_info=exc_info, **kwargs)
```

That is, `logging.exception()` just calls `logging.error()` with `exc_info=True`, which is otherwise `False` by default. If you want to log an exception traceback but at a level different than `logging.ERROR`, just call that function or method with `exc_info=True`.

Keep in mind that `exception()` should only be called in the context of an exception handler, inside of an `except` block:

Python

```
for i in data:  
    try:  
        result = my_longwinded_nested_function(i)  
    except ValueError:  
        # We are in the context of exception handler now.  
        # If it's unclear exactly *why* we couldn't process  
        # `i`, then log the traceback and move on rather than  
        # ditching completely.  
        logger.exception("Could not process %s", i)  
        continue
```

Use this pattern sparingly rather than as a means to suppress any exception. It can be most helpful when you're debugging a long function call stack where you're otherwise seeing an ambiguous, unclear, and hard-to-track error.

Conclusion

Pat yourself on the back, because you've just walked through almost 2,000 lines of dense source code. You're now better equipped to deal with the `logging` package!

Keep in mind that this tutorial has been far from exhaustive in covering all of the classes found in the `logging` package. There's even more machinery that glues everything together. If you'd like to learn more, then you can look into the `Formatter` classes and the separate modules `logging/config.py` and `logging/handlers.py`.

About Brad Solomon

Brad is a software engineer and a member of the Real Python Tutorial Team.

[» More about Brad](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Jim](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#)



Real Python

What is the Python Global Interpreter Lock (GIL)?

by Abhinav Ajitsaria · 28 Comments ·

Tweet Share Email

Table of Contents

- [What problem did the GIL solve for Python?](#)
- [Why was the GIL chosen as the solution?](#)
- [The impact on multi-threaded Python programs](#)
- [Why hasn't the GIL been removed yet?](#)
- [Why wasn't it removed in Python 3?](#)
- [How to deal with Python's GIL](#)

Managing Python Dependencies

With Pip and Virtual Environments

realpython.com



The Python Global Interpreter Lock or [GIL](#), in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an “infamous” feature of Python.

In this article you'll learn how the GIL affects the performance of your Python programs, and how you can mitigate the impact it might have on your code.

What problem did the GIL solve for Python?

Python uses reference counting for memory management. It means that objects created in Python have a reference count variable that keeps track of the number of references that point to the object. When this count reaches zero, the memory occupied by the object is released.

Let's take a look at a brief code example to demonstrate how reference counting works:

```
>>> import sys  
>>> a = []  
>>> b = a  
>>> sys.getrefcount(a)  
3
```

In the above example, the reference count for the empty list object [] was 3. The list object was referenced by a, b and the argument passed to sys.getrefcount().

Back to the GIL:

The problem was that this reference count variable needed protection from race conditions where two threads increase or decrease its value simultaneously. If this happens, it can cause either leaked memory that is never released or, even worse, incorrectly release the memory while a reference to that object still exists. This can cause crashes or other “weird” bugs in your Python programs.

This reference count variable can be kept safe by adding *locks* to all data structures that are shared across threads so that they are not modified inconsistently.

But adding a lock to each object or groups of objects means multiple locks will exist which can cause another problem—Deadlocks (deadlocks can only happen if there is more than one lock). Another side effect would be decreased performance caused by the repeated acquisition and release of locks.

The GIL is a single lock on the interpreter itself which adds a rule that execution of any Python bytecode requires acquiring the interpreter lock. This prevents deadlocks (as there is only one lock) and doesn’t introduce much performance overhead. But it effectively makes any CPU-bound Python program single-threaded.

The GIL, although used by interpreters for other languages like Ruby, is not the only solution to this problem. Some languages avoid the requirement of a GIL for thread-safe memory management by using approaches other than reference counting, such as garbage collection.

On the other hand, this means that those languages often have to compensate for the loss of single threaded performance benefits of a GIL by adding other performance boosting features like JIT compilers.

Why was the GIL chosen as the solution?

So, why was an approach that is seemingly so obstructing used in Python? Was it a bad decision by the developers of Python?

Well, in the [words of Larry Hastings](#), the design decision of the GIL is one of the things that made Python as popular as it is today.

Python has been around since the days when operating systems did not have a concept of threads. Python was designed to be easy-to-use in order to make development quicker and more and more developers started using it.

A lot of extensions were being written for the existing C libraries whose features were needed in Python. To prevent inconsistent changes, these C extensions required a thread-safe memory management which the GIL provided.

The GIL is simple to implement and was easily added to Python. It provides a performance increase to single-threaded programs as only one lock needs to be managed.

C libraries that were not thread-safe became easier to integrate. And these C extensions became one of the reasons why Python was readily adopted by different communities.

As you can see, the GIL was a pragmatic solution to a difficult problem that the CPython developers faced early on in Python’s life.

The impact on multi-threaded Python programs

When you look at a typical Python program—or any computer program for that matter—there’s a difference between those that are CPU-bound in their performance and those that are I/O-bound.

CPU-bound programs are those that are pushing the CPU to its limit. This includes programs that do mathematical

computations like matrix multiplications, searching, image processing, etc.

I/O-bound programs are the ones that spend time waiting for Input/Output which can come from a user, file, database, network, etc. I/O-bound programs sometimes have to wait for a significant amount of time till they get what they need from the source due to the fact that the source may need to do its own processing before the input/output is ready, for example, a user thinking about what to enter into an input prompt or a database query running in its own process.

Let's have a look at a simple CPU-bound program that performs a countdown:

Python

```
# single_threaded.py
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

start = time.time()
countdown(COUNT)
end = time.time()

print('Time taken in seconds -', end - start)
```

Running this code on my system with 4 cores gave the following output:

Shell

```
$ python single_threaded.py
Time taken in seconds - 6.20024037361145
```

Now I modified the code a bit to do the same countdown using two threads in parallel:

Python

```
# multi_threaded.py
import time
from threading import Thread

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()

print('Time taken in seconds -', end - start)
```

And when I ran it again:

Shell

```
$ python multi_threaded.py
Time taken in seconds - 6.924342632293701
```

As you can see, both versions take almost same amount of time to finish. In the multi-threaded version the GIL prevented the CPU-bound threads from executing in parallel.

The GIL does not have much impact on the performance of I/O-bound multi-threaded programs as the lock is shared between threads while they are waiting for I/O.

But a program whose threads are entirely CPU-bound, e.g., a program that processes an image in parts using threads, would not only become single threaded due to the lock but will also see an increase in execution time, as seen in the above example, in comparison to a scenario where it was written to be entirely single-threaded.

This increase is the result of acquire and release overheads added by the lock.

Why hasn't the GIL been removed yet?

The developers of Python receive a lot of complaints regarding this but a language as popular as Python cannot bring a change as significant as the removal of GIL without causing backward incompatibility issues.

The GIL can obviously be removed and this has been done multiple times in the past by the developers and researchers but all those attempts broke the existing C extensions which depend heavily on the solution that the GIL provides.

Of course, there are other solutions to the problem that the GIL solves but some of them decrease the performance of single-threaded and multi-threaded I/O-bound programs and some of them are just too difficult. After all, you wouldn't want your existing Python programs to run slower after a new version comes out, right?

The creator and BDFL of Python, Guido van Rossum, gave an answer to the community in September 2007 in his article [“It isn't Easy to remove the GIL”](#):

“I'd welcome a set of patches into Py3k *only if* the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) *does not decrease*”

And this condition hasn't been fulfilled by any of the attempts made since.

Why wasn't it removed in Python 3?

Python 3 did have a chance to start a lot of features from scratch and in the process, broke some of the existing C extensions which then required changes to be updated and ported to work with Python 3. This was the reason why the early versions of Python 3 saw slower adoption by the community.

But why wasn't GIL removed alongside?

Removing the GIL would have made Python 3 slower in comparison to Python 2 in single-threaded performance and you can imagine what that would have resulted in. You can't argue with the single-threaded performance benefits of the GIL. So the result is that Python 3 still has the GIL.

But Python 3 did bring a major improvement to the existing GIL—

We discussed the impact of GIL on “only CPU-bound” and “only I/O-bound” multi-threaded programs but what about the programs where some threads are I/O-bound and some are CPU-bound?

In such programs, Python's GIL was known to starve the I/O-bound threads by not giving them a chance to acquire the GIL from CPU-bound threads.

This was because of a mechanism built into Python that forced threads to release the GIL **after a fixed interval** of continuous use and if nobody else acquired the GIL, the same thread could continue its use.

```
Python >>>
>>> import sys
>>> # The interval is set to 100 instructions:
>>> sys.getcheckinterval()
100
```

The problem in this mechanism was that most of the time the CPU-bound thread would reacquire the GIL itself before other threads could acquire it. This was researched by David Beazley and visualizations can be found [here](#).

This problem was fixed in Python 3.2 in 2009 by Antoine Pitrou who [added a mechanism](#) of looking at the number of

GIL acquisition requests by other threads that got dropped and not allowing the current thread to reacquire GIL before other threads got a chance to run.

How to deal with Python's GIL

If the GIL is causing you problems, here a few approaches you can try:

Multi-processing vs multi-threading: The most popular way is to use a multi-processing approach where you use multiple processes instead of threads. Each Python process gets its own Python interpreter and memory space so the GIL won't be a problem. Python has a [multiprocessing](#) module which lets us create processes easily like this:

Python

```
from multiprocessing import Pool
import time

COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

if __name__ == '__main__':
    pool = Pool(processes=2)
    start = time.time()
    r1 = pool.apply_async(countdown, [COUNT//2])
    r2 = pool.apply_async(countdown, [COUNT//2])
    pool.close()
    pool.join()
    end = time.time()
    print('Time taken in seconds - ', end - start)
```

Running this on my system gave this output:

Shell

```
$ python multiprocess.py
Time taken in seconds - 4.060242414474487
```

A decent performance increase compared to the multi-threaded version, right?

The time didn't drop to half of what we saw above because process management has its own overheads. Multiple processes are heavier than multiple threads, so, keep in mind that this could become a scaling bottleneck.

Alternative Python interpreters: Python has multiple interpreter implementations. CPython, Jython, IronPython and PyPy, written in C, Java, C# and Python respectively, are the most popular ones. GIL exists only in the original Python implementation that is CPython. If your program, with its libraries, is available for one of the other implementations then you can try them out as well.

Just wait it out: While many Python users take advantage of the single-threaded performance benefits of GIL. The multi-threading programmers don't have to fret as some of the brightest minds in the Python community are working to remove the GIL from CPython. One such attempt is known as the [Gilectomy](#).

The Python GIL is often regarded as a mysterious and difficult topic. But keep in mind that as a Pythonista you're usually only affected by it if you are writing C extensions or if you're using CPU-bound multi-threading in your programs.

In that case, this article should give you everything you need to understand what the GIL is and how to deal with it in your own projects. And if you want to understand the low-level inner workings of GIL, I'd recommend you watch the [Understanding the Python GIL](#) talk by David Beazley.

About Abhinav Ajitsaria

Abhinav is a Software Engineer from India. He loves to talk about system design, machine

learning, AWS and of course, Python.

» [More about Abhinav](#)

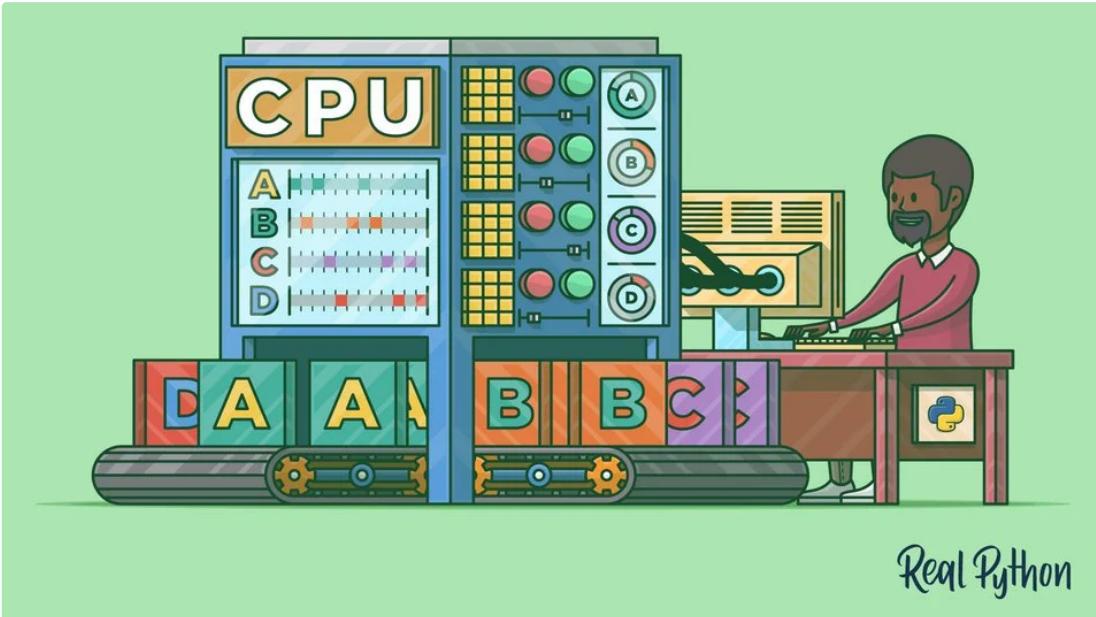
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Dan](#)

Keep Learning

Related Tutorial Categories: [advanced](#) [python](#)



Speed Up Your Python Program With Concurrency

by Jim Anderson 94 Comments advanced best-practices

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Is Concurrency?](#)
- [What Is Parallelism?](#)
- [When Is Concurrency Useful?](#)
- [How to Speed Up an I/O-Bound Program](#)
 - [Synchronous Version](#)
 - [threading Version](#)
 - [asyncio Version](#)
 - [multiprocessing Version](#)
- [How to Speed Up a CPU-Bound Program](#)
 - [CPU-Bound Synchronous Version](#)
 - [threading and asyncio Versions](#)
 - [CPU-Bound multiprocessing Version](#)
- [When to Use Concurrency](#)
- [Conclusion](#)

 **blackfire.io**
Profile & Optimize Python Apps Performance



Now available as
Public Beta
Sign-up for free and
install in minutes!

If you've heard lots of talk about [asyncio being added to Python](#) but are curious how it compares to other concurrency methods or are wondering what concurrency is and how it might speed up your program, you've come to the right place.

In this article, you'll learn the following:

- What **concurrency** is
- What **parallelism** is
- How some of **Python's concurrency methods** compare, including `threading`, `asyncio`, and `multiprocessing`
- **When to use concurrency** in your program and which module to use

This article assumes that you have a basic understanding of Python and that you're using at least version 3.6 to run the examples. You can download the examples from the [Real Python GitHub repo](#).

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

 **Take the Quiz:** Test your knowledge with our interactive “Python Concurrency” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

What Is Concurrency?

The dictionary definition of concurrency is simultaneous occurrence. In Python, the things that are occurring simultaneously are called by different names (thread, task, process) but at a high level, they all refer to a sequence of instructions that run in order.

I like to think of them as different trains of thought. Each one can be stopped at certain points, and the CPU or brain that is processing them can switch to a different one. The state of each one is saved so it can be restarted right where it was interrupted.

You might wonder why Python uses different words for the same concept. It turns out that threads, tasks, and processes are only the same if you view them from a high level. Once you start digging into the details, they all represent slightly different things. You’ll see more of how they are different as you progress through the examples.

Now let’s talk about the simultaneous part of that definition. You have to be a little careful because, when you get down to the details, only `multiprocessing` actually runs these trains of thought at literally the same time.

`Threading` and `asyncio` both run on a single processor and therefore only run one at a time. They just cleverly find ways to take turns to speed up the overall process. Even though they don’t run different trains of thought simultaneously, we still call this concurrency.

The way the threads or tasks take turns is the big difference between `threading` and `asyncio`. In `threading`, the operating system actually knows about each thread and can interrupt it at any time to start running a different thread. This is called [pre-emptive multitasking](#) since the operating system can pre-empt your thread to make the switch.

`Pre-emptive multitasking` is handy in that the code in the thread doesn’t need to do anything to make the switch. It can also be difficult because of that “at any time” phrase. This switch can happen in the middle of a single Python statement, even a trivial one like `x = x + 1`.

`Asyncio`, on the other hand, uses [cooperative multitasking](#). The tasks must cooperate by announcing when they are ready to be switched out. That means that the code in the task has to change slightly to make this happen.

The benefit of doing this extra work up front is that you always know where your task will be swapped out. It will not be swapped out in the middle of a Python statement unless that statement is marked. You’ll see later how this can simplify parts of your design.

What Is Parallelism?

So far, you’ve looked at concurrency that happens on a single processor. What about all of those CPU cores your cool, new laptop has? How can you make use of them? `multiprocessing` is the answer.

With `multiprocessing`, Python creates new processes. A process here can be thought of as almost a completely different program, though technically they’re usually defined as a collection of resources where the resources include memory, file handles and things like that. One way to think about it is that each process runs in its own Python interpreter.

Because they are different processes, each of your trains of thought in a `multiprocessing` program can run on a different core. Running on a different core means that they actually can run at the same time, which is fabulous. There are some complications that arise from doing this, but Python does a pretty good job of smoothing them over most of the time.

Now that you have an idea of what concurrency and parallelism are, let’s review their differences, and then we can

look at why they can be useful:

Concurrency Type	Switching Decision	Number of Processors
Pre-emptive multitasking (threading)	The operating system decides when to switch tasks external to Python.	1
Cooperative multitasking (asyncio)	The tasks decide when to give up control.	1
Multiprocessing (multiprocessing)	The processes all run at the same time on different processors.	Many

Each of these types of concurrency can be useful. Let's take a look at what types of programs they can help you speed up.

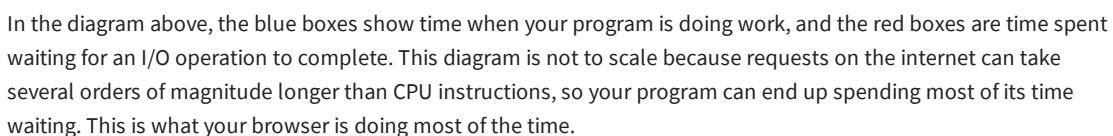
When Is Concurrency Useful?

Concurrency can make a big difference for two types of problems. These are generally called CPU-bound and I/O-bound.

I/O-bound problems cause your program to slow down because it frequently must wait for input/output (I/O) from some external resource. They arise frequently when your program is working with things that are much slower than your CPU.

Examples of things that are slower than your CPU are legion, but your program thankfully does not interact with most of them. The slow things your program will interact with most frequently are the file system and network connections.

Let's see what that looks like:



In the diagram above, the blue boxes show time when your program is doing work, and the red boxes are time spent waiting for an I/O operation to complete. This diagram is not to scale because requests on the internet can take several orders of magnitude longer than CPU instructions, so your program can end up spending most of its time waiting. This is what your browser is doing most of the time.

On the flip side, there are classes of programs that do significant computation without talking to the network or accessing a file. These are the CPU-bound programs, because the resource limiting the speed of your program is the CPU, not the network or the file system.

Here's a corresponding diagram for a CPU-bound program:

As you work through the examples in the following section, you'll see that different forms of concurrency work better or worse with CPU-bound and I/O-bound programs. Adding concurrency to your program adds extra code and complications, so you'll need to decide if the potential speed up is worth the extra effort. By the end of this article, you should have enough info to start making that decision.

Here's a quick summary to clarify this concept:

I/O-Bound Process	CPU-Bound Process
Your program spends most of its time talking to a slow device, like a network connection, a hard drive, or a printer.	You program spends most of its time doing CPU operations.
Speeding it up involves overlapping the times spent waiting for these devices.	Speeding it up involves finding ways to do more computations in the same amount of time.

You'll look at I/O-bound programs first. Then, you'll get to see some code dealing with CPU-bound programs.

How to Speed Up an I/O-Bound Program

Let's start by focusing on I/O-bound programs and a common problem: downloading content over the network. For our example, you will be downloading web pages from a few sites, but it really could be any network traffic. It's just easier to visualize and set up with web pages.

Synchronous Version

We'll start with a non-concurrent version of this task. Note that this program requires the [requests](#) module. You should run `pip install requests` before running it, probably using a [virtualenv](#). This version does not use concurrency at all:

Python

```
import requests
import time

def download_site(url, session):
    with session.get(url) as response:
        print(f"Read {len(response.content)} from {url}")

def download_all_sites(sites):
    with requests.Session() as session:
        for url in sites:
            download_site(url, session)
```

```

if __name__ == "__main__":
    sites = [
        "https://www.python.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    download_all_sites(sites)
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} in {duration} seconds")

```

As you can see, this is a fairly short program. `download_site()` just downloads the contents from a URL and prints the size. One small thing to point out is that we're using a `Session` object from `requests`.

It is possible to simply use `get()` from `requests` directly, but creating a `Session` object allows `requests` to do some fancy networking tricks and really speed things up.

`download_all_sites()` creates the `Session` and then walks through the list of sites, downloading each one in turn. Finally, it prints out how long this process took so you can have the satisfaction of seeing how much concurrency has helped us in the following examples.

The processing diagram for this program will look much like the I/O-bound diagram in the last section.

Note: Network traffic is dependent on many factors that can vary from second to second. I've seen the times of these tests double from one run to another due to network issues.

Why the Synchronous Version Rocks

The great thing about this version of code is that, well, it's easy. It was comparatively easy to write and debug. It's also more straight-forward to think about. There's only one train of thought running through it, so you can predict what the next step is and how it will behave.

The Problems With the Synchronous Version

The big problem here is that it's relatively slow compared to the other solutions we'll provide. Here's an example of what the final output gave on my machine:

Shell

```

$ ./io_non_concurrent.py
[most output skipped]
Downloaded 160 in 14.289619207382202 seconds

```

Note: Your results may vary significantly. When running this script, I saw the times vary from 14.2 to 21.9 seconds. For this article, I took the fastest of three runs as the time. The differences between the methods will still be clear.

Being slower isn't always a big issue, however. If the program you're running takes only 2 seconds with a synchronous version and is only run rarely, it's probably not worth adding concurrency. You can stop here.

What if your program is run frequently? What if it takes hours to run? Let's move on to concurrency by rewriting this program using `threading`.

threading Version

As you probably guessed, writing a threaded program takes more effort. You might be surprised at how little extra effort it takes for simple cases, however. Here's what the same program looks like with `threading`:

Python

```

import concurrent.futures
import requests
import threading
import time

```

```

thread_local = threading.local()

def get_session():
    if not hasattr(thread_local, "session"):
        thread_local.session = requests.Session()
    return thread_local.session

def download_site(url):
    session = get_session()
    with session.get(url) as response:
        print(f"Read {len(response.content)} from {url}")

def download_all_sites(sites):
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        executor.map(download_site, sites)

if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    download_all_sites(sites)
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} in {duration} seconds")

```

When you add threading, the overall structure is the same and you only needed to make a few changes. `download_all_sites()` changed from calling the function once per site to a more complex structure.

In this version, you're creating a `ThreadPoolExecutor`, which seems like a complicated thing. Let's break that down: `ThreadPoolExecutor = Thread + Pool + Executor`.

You already know about the `Thread` part. That's just a train of thought we mentioned earlier. The `Pool` portion is where it starts to get interesting. This object is going to create a pool of threads, each of which can run concurrently. Finally, the `Executor` is the part that's going to control how and when each of the threads in the pool will run. It will execute the request in the pool.

Helpfully, the standard library implements `ThreadPoolExecutor` as a context manager so you can use the `with` syntax to manage creating and freeing the pool of `Threads`.

Once you have a `ThreadPoolExecutor`, you can use its handy `.map()` method. This method runs the passed-in function on each of the sites in the list. The great part is that it automatically runs them concurrently using the pool of threads it is managing.

Those of you coming from other languages, or even Python 2, are probably wondering where the usual objects and functions are that manage the details you're used to when dealing with threading, things like `Thread.start()`, `Thread.join()`, and `Queue`.

These are all still there, and you can use them to achieve fine-grained control of how your threads are run. But, starting with Python 3.2, the standard library added a higher-level abstraction called `Executors` that manage many of the details for you if you don't need that fine-grained control.

The other interesting change in our example is that each thread needs to create its own `requests.Session()` object. When you're looking at the documentation for `requests`, it's not necessarily easy to tell, but reading [this issue](#), it seems fairly clear that you need a separate `Session` for each thread.

This is one of the interesting and difficult issues with threading. Because the operating system is in control of when your task gets interrupted and another task starts, any data that is shared between the threads needs to be protected, or thread-safe. Unfortunately `requests.Session()` is not thread-safe.

There are several strategies for making data accesses thread-safe depending on what the data is and how you're using it. One of them is to use thread-safe data structures like `Queue` from Python's `queue` module.

These objects use low-level primitives like `threading.Lock` to ensure that only one thread can access a block of code or a bit of memory at the same time. You are using this strategy indirectly by way of the `ThreadPoolExecutor`

object.

Another strategy to use here is something called thread local storage. `Threading.local()` creates an object that look like a global but is specific to each individual thread. In your example, this is done with `threadLocal` and `get_session()`:

Python

```
threadLocal = threading.local()

def get_session():
    if not hasattr(threadLocal, "session"):
        threadLocal.session = requests.Session()
    return threadLocal.session
```

`ThreadLocal` is in the `threading` module to specifically solve this problem. It looks a little odd, but you only want to create one of these objects, not one for each thread. The object itself takes care of separating accesses from different threads to different data.

When `get_session()` is called, the session it looks up is specific to the particular thread on which it's running. So each thread will create a single session the first time it calls `get_session()` and then will simply use that session on each subsequent call throughout its lifetime.

Finally, a quick note about picking the number of threads. You can see that the example code uses 5 threads. Feel free to play around with this number and see how the overall time changes. You might expect that having one thread per download would be the fastest but, at least on my system it was not. I found the fastest results somewhere between 5 and 10 threads. If you go any higher than that, then the extra overhead of creating and destroying the threads erases any time savings.

The difficult answer here is that the correct number of threads is not a constant from one task to another. Some experimentation is required.

Why the `threading` Version Rocks

It's fast! Here's the fastest run of my tests. Remember that the non-concurrent version took more than 14 seconds:

Shell

```
$ ./io_threading.py
[most output skipped]
Downloaded 160 in 3.7238826751708984 seconds
```

Here's what its execution timing diagram looks like:

It uses multiple threads to have multiple open requests out to web sites at the same time, allowing your program to overlap the waiting times and get the final result faster! Yippee! That was the goal.

The Problems with the `threading` Version

Well, as you can see from the example, it takes a little more code to make this happen, and you really have to give

some thought to what data is shared between threads.

Threads can interact in ways that are subtle and hard to detect. These interactions can cause race conditions that frequently result in random, intermittent bugs that can be quite difficult to find. Those of you who are unfamiliar with the concept of race conditions might want to expand and read the section below.

asyncio Version

Before you jump into examining the asyncio example code, let's talk more about how asyncio works.

asyncio Basics

This will be a simplified version of asyncio. There are many details that are glossed over here, but it still conveys the idea of how it works.

The general concept of asyncio is that a single Python object, called the event loop, controls how and when each task gets run. The event loop is aware of each task and knows what state it's in. In reality, there are many states that tasks could be in, but for now let's imagine a simplified event loop that just has two states.

The ready state will indicate that a task has work to do and is ready to be run, and the waiting state means that the task is waiting for some external thing to finish, such as a network operation.

Your simplified event loop maintains two lists of tasks, one for each of these states. It selects one of the ready tasks and starts it back to running. That task is in complete control until it cooperatively hands the control back to the event loop.

When the running task gives control back to the event loop, the event loop places that task into either the ready or waiting list and then goes through each of the tasks in the waiting list to see if it has become ready by an I/O operation completing. It knows that the tasks in the ready list are still ready because it knows they haven't run yet.

Once all of the tasks have been sorted into the right list again, the event loop picks the next task to run, and the process repeats. Your simplified event loop picks the task that has been waiting the longest and runs that. This process repeats until the event loop is finished.

An important point of asyncio is that the tasks never give up control without intentionally doing so. They never get interrupted in the middle of an operation. This allows us to share resources a bit more easily in asyncio than in threading. You don't have to worry about making your code thread-safe.

That's a high-level view of what's happening with asyncio. If you want more detail, [this StackOverflow answer](#) provides some good details if you want to dig deeper.

async and await

Now let's talk about two new keywords that were added to Python: `async` and `await`. In light of the discussion above, you can view `await` as the magic that allows the task to hand control back to the event loop. When your code awaits a function call, it's a signal that the call is likely to be something that takes a while and that the task should give up control.

It's easiest to think of `async` as a flag to Python telling it that the function about to be defined uses `await`. There are some cases where this is not strictly true, like [asynchronous generators](#), but it holds for many cases and gives you a simple model while you're getting started.

One exception to this that you'll see in the next code is the `async with` statement, which creates a context manager from an object you would normally await. While the semantics are a little different, the idea is the same: to flag this context manager as something that can get swapped out.

As I'm sure you can imagine, there's some complexity in managing the interaction between the event loop and the tasks. For developers starting out with asyncio, these details aren't important, but you do need to remember that any function that calls `await` needs to be marked with `async`. You'll get a syntax error otherwise.

Back to Code

Now that you've got a basic understanding of what `asyncio` is, let's walk through the `asyncio` version of the

example code and figure out how it works. Note that this version adds [aiohttp](#). You should run `pip install aiohttp` before running it:

Python

```
import asyncio
import time
import aiohttp

async def download_site(session, url):
    async with session.get(url) as response:
        print("Read {} from {}".format(response.content_length, url))

async def download_all_sites(sites):
    async with aiohttp.ClientSession() as session:
        tasks = []
        for url in sites:
            task = asyncio.ensure_future(download_site(session, url))
            tasks.append(task)
        await asyncio.gather(*tasks, return_exceptions=True)

if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    asyncio.get_event_loop().run_until_complete(download_all_sites(sites))
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} sites in {duration} seconds")
```

This version is a bit more complex than the previous two. It has a similar structure, but there's a bit more work setting up the tasks than there was creating the `ThreadPoolExecutor`. Let's start at the top of the example.

`download_site()`

`download_site()` at the top is almost identical to the threading version with the exception of the `async` keyword on the function definition line and the `async with` keywords when you actually call `session.get()`. You'll see later why `Session` can be passed in here rather than using thread-local storage.

`download_all_sites()`

`download_all_sites()` is where you will see the biggest change from the threading example.

You can share the session across all tasks, so the session is created here as a context manager. The tasks can share the session because they are all running on the same thread. There is no way one task could interrupt another while the session is in a bad state.

Inside that context manager, it creates a list of tasks using `asyncio.ensure_future()`, which also takes care of starting them. Once all the tasks are created, this function uses `asyncio.gather()` to keep the session context alive until all of the tasks have completed.

The threading code does something similar to this, but the details are conveniently handled in the `ThreadPoolExecutor`. There currently is not an `AsyncioPoolExecutor` class.

There is one small but important change buried in the details here, however. Remember how we talked about the number of threads to create? It wasn't obvious in the threading example what the optimal number of threads was.

One of the cool advantages of `asyncio` is that it scales far better than `threading`. Each task takes far fewer resources and less time to create than a thread, so creating and running more of them works well. This example just creates a separate task for each site to download, which works out quite well.

`__main__`

Finally, the nature of `asyncio` means that you have to start up the event loop and tell it which tasks to run. The `__main__` section at the bottom of the file contains the code to `get_event_loop()` and then `run_until_complete()`. If nothing else, they've done an excellent job in naming those functions.

If you've updated to [Python 3.7](#), the Python core developers simplified this syntax for you. Instead of the `asyncio.get_event_loop().run_until_complete()` tongue-twister, you can just use `asyncio.run()`.

Why the `asyncio` Version Rocks

It's really fast! In the tests on my machine, this was the fastest version of the code by a good margin:

Shell

```
$ ./io_asyncio.py
[most output skipped]
Downloaded 160 in 2.5727896690368652 seconds
```

The execution timing diagram looks quite similar to what's happening in the `threading` example. It's just that the I/O requests are all done by the same thread:

The lack of a nice wrapper like the `ThreadPoolExecutor` makes this code a bit more complex than the `threading` example. This is a case where you have to do a little extra work to get much better performance.

Also there's a common argument that having to add `async` and `await` in the proper locations is an extra complication. To a small extent, that is true. The flip side of this argument is that it forces you to think about when a given task will get swapped out, which can help you create a better, faster, design.

The scaling issue also looms large here. Running the `threading` example above with a thread for each site is noticeably slower than running it with a handful of threads. Running the `asyncio` example with hundreds of tasks didn't slow it down at all.

The Problems With the `asyncio` Version

There are a couple of issues with `asyncio` at this point. You need special `async` versions of libraries to gain the full advantage of `asyncio`. Had you just used `requests` for downloading the sites, it would have been much slower because `requests` is not designed to notify the event loop that it's blocked. This issue is getting smaller and smaller as time goes on and more libraries embrace `asyncio`.

Another, more subtle, issue is that all of the advantages of cooperative multitasking get thrown away if one of the tasks doesn't cooperate. A minor mistake in code can cause a task to run off and hold the processor for a long time, starving other tasks that need running. There is no way for the event loop to break in if a task does not hand control

[back to top](#)

With that in mind, let's step up to a radically different approach to concurrency, `multiprocessing`.

`multiprocessing` Version

Unlike the previous approaches, the `multiprocessing` version of the code takes full advantage of the multiple CPUs that your cool, new computer has. Or, in my case, that my clunky, old laptop has. Let's start with the code:

```
Python
import requests
import multiprocessing
import time

session = None

def set_global_session():
    global session
    if not session:
        session = requests.Session()

def download_site(url):
    with session.get(url) as response:
        name = multiprocessing.current_process().name
        print(f"{name}:Read {len(response.content)} from {url}")

def download_all_sites(sites):
    with multiprocessing.Pool(initializer=set_global_session) as pool:
        pool.map(download_site, sites)

if __name__ == "__main__":
    sites = [
        "https://www.jython.org",
        "http://olympus.realpython.org/dice",
    ] * 80
    start_time = time.time()
    download_all_sites(sites)
    duration = time.time() - start_time
    print(f"Downloaded {len(sites)} in {duration} seconds")
```

This is much shorter than the `asyncio` example and actually looks quite similar to the `threading` example, but before we dive into the code, let's take a quick tour of what `multiprocessing` does for you.

`multiprocessing` in a Nutshell

Up until this point, all of the examples of concurrency in this article run only on a single CPU or core in your computer. The reasons for this have to do with the current design of CPython and something called the Global Interpreter Lock, or GIL.

This article won't dive into the hows and whys of the [GIL](#). It's enough for now to know that the synchronous, threading, and asyncio versions of this example all run on a single CPU.

`multiprocessing` in the standard library was designed to break down that barrier and run your code across multiple CPUs. At a high level, it does this by creating a new instance of the Python interpreter to run on each CPU and then farming out part of your program to run on it.

As you can imagine, bringing up a separate Python interpreter is not as fast as starting a new thread in the current Python interpreter. It's a heavyweight operation and comes with some restrictions and difficulties, but for the correct problem, it can make a huge difference.

`multiprocessing` Code

The code has a few small changes from our synchronous version. The first one is in `download_all_sites()`. Instead of simply calling `download_site()` repeatedly, it creates a `multiprocessing.Pool` object and has it map `download_site` to the iterable `sites`. This should look familiar from the `threading` example.

What happens here is that the `Pool` creates a number of separate Python interpreter processes and has each one run the specified function on some of the items in the iterable, which in our case is the list of sites. The communication between the main process and the other processes is handled by the `multiprocessing` module for you.

The line that creates `Pool` is worth your attention. First off, it does not specify how many processes to create in the `Pool`, although that is an optional parameter. By default, `multiprocessing.Pool()` will determine the number of CPUs in your computer and match that. This is frequently the best answer, and it is in our case.

For this problem, increasing the number of processes did not make things faster. It actually slowed things down because the cost for setting up and tearing down all those processes was larger than the benefit of doing the I/O requests in parallel.

Next we have the `initializer=set_global_session` part of that call. Remember that each process in our `Pool` has its own memory space. That means that they cannot share things like a `Session` object. You don't want to create a new `Session` each time the function is called, you want to create one for each process.

The `initializer` function parameter is built for just this case. There is not a way to pass a return value back from the `initializer` to the function called by the process `download_site()`, but you can initialize a global `session` variable to hold the single session for each process. Because each process has its own memory space, the global for each one will be different.

That's really all there is to it. The rest of the code is quite similar to what you've seen before.

Why the `multiprocessing` Version Rocks

The `multiprocessing` version of this example is great because it's relatively easy to set up and requires little extra code. It also takes full advantage of the CPU power in your computer. The execution timing diagram for this code looks like this:

The Problems With the `multiprocessing` Version

This version of the example does require some extra setup, and the global `session` object is strange. You have to spend some time thinking about which variables will be accessed in each process.

Finally, it is clearly slower than the `asyncio` and `threading` versions in this example:

Sneil

```
$ ./io_mp.py
[most output skipped]
Downloaded 160 in 5.718175172805786 seconds
```

That's not surprising, as I/O-bound problems are not really why `multiprocessing` exists. You'll see more as you step into the next section and look at CPU-bound examples.

How to Speed Up a CPU-Bound Program

Let's shift gears here a little bit. The examples so far have all dealt with an I/O-bound problem. Now, you'll look into a CPU-bound problem. As you saw, an I/O-bound problem spends most of its time waiting for external operations, like a network call, to complete. A CPU-bound problem, on the other hand, does few I/O operations, and its overall execution time is a factor of how fast it can process the required data.

For the purposes of our example, we'll use a somewhat silly function to create something that takes a long time to run on the CPU. This function computes the sum of the squares of each number from 0 to the passed-in value:

Python

```
def cpu_bound(number):
    return sum(i * i for i in range(number))
```

You'll be passing in large numbers, so this will take a while. Remember, this is just a placeholder for your code that actually does something useful and requires significant processing time, like computing the roots of equations or [sorting](#) a large data structure.

CPU-Bound Synchronous Version

Now let's look at the non-concurrent version of the example:

Python

```
import time

def cpu_bound(number):
    return sum(i * i for i in range(number))

def find_sums(numbers):
    for number in numbers:
        cpu_bound(number)

if __name__ == "__main__":
    numbers = [5_000_000 + x for x in range(20)]

    start_time = time.time()
    find_sums(numbers)
    duration = time.time() - start_time
    print(f"Duration {duration} seconds")
```

This code calls `cpu_bound()` 20 times with a different large number each time. It does all of this on a single thread in a single process on a single CPU. The execution timing diagram looks like this:

Unlike the I/O-bound examples, the CPU-bound examples are usually fairly consistent in their run times. This one takes about 7.8 seconds on my machine:

Shell

```
$ ./cpu_non_concurrent.py  
Duration 7.834432125091553 seconds
```

Clearly we can do better than this. This is all running on a single CPU with no concurrency. Let's see what we can do to make it better.

threading and asyncio Versions

How much do you think rewriting this code using `threading` or `asyncio` will speed this up?

If you answered "Not at all," give yourself a cookie. If you answered, "It will slow it down," give yourself two cookies.

Here's why: In your I/O-bound example above, much of the overall time was spent waiting for slow operations to finish. `threading` and `asyncio` sped this up by allowing you to overlap the times you were waiting instead of doing them sequentially.

On a CPU-bound problem, however, there is no waiting. The CPU is cranking away as fast as it can to finish the problem. In Python, both threads and tasks run on the same CPU in the same process. That means that the one CPU is doing all of the work of the non-concurrent code plus the extra work of setting up threads or tasks. It takes more than 10 seconds:

Shell

```
$ ./cpu_threading.py  
Duration 10.407078266143799 seconds
```

I've written up a `threading` version of this code and placed it with the other example code in the [GitHub repo](#) so you can go test this yourself. Let's not look at that just yet, however.

CPU-Bound multiprocessing Version

Now you've finally reached where `multiprocessing` really shines. Unlike the other concurrency libraries, `multiprocessing` is explicitly designed to share heavy CPU workloads across multiple CPUs. Here's what its execution timing diagram looks like:

Here's what the code looks like:

Python

```
import multiprocessing
import time

def cpu_bound(number):
    return sum(i * i for i in range(number))

def find_sums(numbers):
    with multiprocessing.Pool() as pool:
        pool.map(cpu_bound, numbers)

if __name__ == "__main__":
    numbers = [5_000_000 + x for x in range(20)]

    start_time = time.time()
    find_sums(numbers)
    duration = time.time() - start_time
    print(f"Duration {duration} seconds")
```

Little of this code had to change from the non-concurrent version. You had to import `multiprocessing` and then just change from looping through the numbers to creating a `multiprocessing.Pool` object and using its `.map()` method to send individual numbers to worker-processes as they become free.

This was just what you did for the I/O-bound `multiprocessing` code, but here you don't need to worry about the `Session` object.

As mentioned above, the `processes` optional parameter to the `multiprocessing.Pool()` constructor deserves some attention. You can specify how many `Process` objects you want created and managed in the `Pool`. By default, it will determine how many CPUs are in your machine and create a process for each one. While this works great for our simple example, you might want to have a little more control in a production environment.

Also, as we mentioned in the first section about threading, the `multiprocessing.Pool` code is built upon building blocks like `Queue` and `Semaphore` that will be familiar to those of you who have done multithreaded and `multiprocessing` code in other languages.

Why the `multiprocessing` Version Rocks

The `multiprocessing` version of this example is great because it's relatively easy to set up and requires little extra code. It also takes full advantage of the CPU power in your computer.

Hey, that's exactly what I said the last time we looked at `multiprocessing`. The big difference is that this time it is clearly the best option. It takes 2.5 seconds on my machine:

Shell

```
$ ./cpu_mp.py
Duration 2.5175397396087646 seconds
```

That's much better than we saw with the other options.

The Problems With the `multiprocessing` Version

There are some drawbacks to using `multiprocessing`. They don't really show up in this simple example, but splitting your problem up so each processor can work independently can sometimes be difficult.

Also, many solutions require more communication between the processes. This can add some complexity to your solution that a non-concurrent program would not need to deal with.

When to Use Concurrency

You've covered a lot of ground here, so let's review some of the key ideas and then discuss some decision points that will help you determine which, if any, concurrency module you want to use in your project.

The first step of this process is deciding if you *should* use a concurrency module. While the examples here make each of the libraries look pretty simple, concurrency always comes with extra complexity and can often result in bugs that are difficult to find.

Hold out on adding concurrency until you have a known performance issue and *then* determine which type of concurrency you need. As [Donald Knuth](#) has said, "Premature optimization is the root of all evil (or at least most of it) in programming."

Once you've decided that you should optimize your program, figuring out if your program is CPU-bound or I/O-bound is a great next step. Remember that I/O-bound programs are those that spend most of their time waiting for something to happen while CPU-bound programs spend their time processing data or crunching numbers as fast as they can.

As you saw, CPU-bound problems only really gain from using `multiprocessing`. `threading` and `asyncio` did not help this type of problem at all.

For I/O-bound problems, there's a general rule of thumb in the Python community: "Use `asyncio` when you can, `threading` when you must." `asyncio` can provide the best speed up for this type of program, but sometimes you will require critical libraries that have not been ported to take advantage of `asyncio`. Remember that any task that doesn't give up control to the event loop will block all of the other tasks.

Conclusion

You've now seen the basic types of concurrency available in Python:

- `threading`
- `asyncio`
- `multiprocessing`

You've got the understanding to decide which concurrency method you should use for a given problem, or if you should use any at all! In addition, you've achieved a better understanding of some of the problems that can arise when you're using concurrency.

I hope you've learned a lot from this article and that you find a great use for concurrency in your own projects! Be sure to take our "Python Concurrency" quiz linked below to check your learning:

 **Take the Quiz:** Test your knowledge with our interactive "Python Concurrency" quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

About Jim Anderson

Jim has been programming for a long time in a variety of languages. He has worked on embedded systems, built distributed build systems, done off-shore vendor management, and s in many, many meetings.

[» More about Jim](#)

worked on this tutorial are:

[Aldren](#)

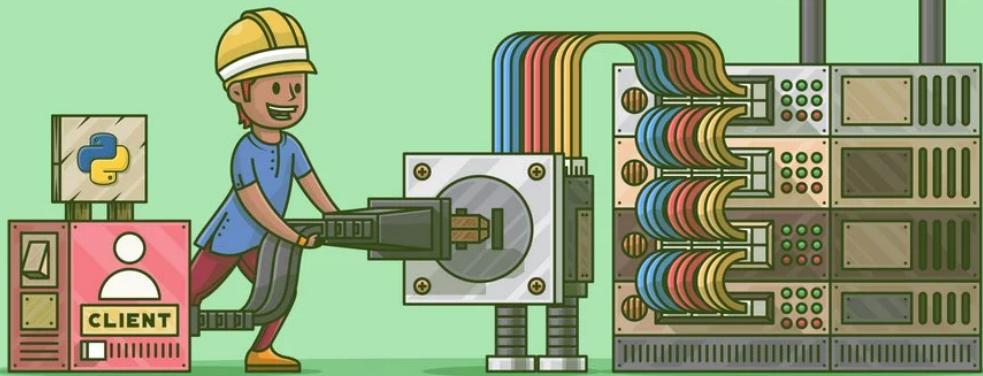
[Brad](#)

[David](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [advanced](#) [best-practices](#)



Real Python

Socket Programming in Python (Guide)

by [Nathan Jennings](#) 87 Comments [advanced](#) [python](#) [web-dev](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Background](#)
- [Socket API Overview](#)
- [TCP Sockets](#)
- [Echo Client and Server](#)
 - [Echo Server](#)
 - [Echo Client](#)
 - [Running the Echo Client and Server](#)
 - [Viewing Socket State](#)
- [Communication Breakdown](#)
- [Handling Multiple Connections](#)
- [Multi-Connection Client and Server](#)
 - [Multi-Connection Server](#)
 - [Multi-Connection Client](#)
 - [Running the Multi-Connection Client and Server](#)
- [Application Client and Server](#)
 - [Application Protocol Header](#)
 - [Sending an Application Message](#)
 - [Application Message Class](#)
 - [Running the Application Client and Server](#)
- [Troubleshooting](#)
 - [ping](#)
 - [netstat](#)
 - [Windows](#)
 - [Wireshark](#)
- [Reference](#)
 - [Python Documentation](#)
 - [Errors](#)
 - [Socket Address Families](#)
 - [Using Hostnames](#)
 - [Blocking Calls](#)
 - [Closing Connections](#)

- [Byte Eniganness](#)
- [Conclusion](#)



Your Guided Tour Through the Python 3.9 Interpreter »

Sockets and the socket API are used to send messages across a network. They provide a form of [inter-process communication \(IPC\)](#). The network can be a logical, local network to the computer, or one that's physically connected to an external network, with its own connections to other networks. The obvious example is the Internet, which you connect to via your ISP.

This tutorial has three different iterations of building a socket server and client with Python:

1. We'll start the tutorial by looking at a simple socket server and client.
2. Once you've seen the API and how things work in this initial example, we'll look at an improved version that handles multiple connections simultaneously.
3. Finally, we'll progress to building an example server and client that functions like a full-fledged socket application, complete with its own custom header and content.

By the end of this tutorial, you'll understand how to use the main functions and methods in Python's [socket module](#) to write your own client-server applications. This includes showing you how to use a custom class to send messages and data between endpoints that you can build upon and utilize for your own applications.

The examples in this tutorial use Python 3.6. You can find the [source code on GitHub](#).

Networking and sockets are large subjects. Literal volumes have been written about them. If you're new to sockets or networking, it's completely normal if you feel overwhelmed with all of the terms and pieces. I know I did!

Don't be discouraged though. I've written this tutorial for you. As we do with Python, we can learn a little bit at a time. Use your browser's bookmark feature and come back when you're ready for the next section.

Let's get started!

Background

Sockets have a long history. Their use [originated with ARPANET](#) in 1971 and later became an API in the Berkeley Software Distribution (BSD) operating system released in 1983 called [Berkeley sockets](#).

When the Internet took off in the 1990s with the World Wide Web, so did network programming. Web servers and browsers weren't the only applications taking advantage of newly connected networks and using sockets. Client-server applications of all types and sizes came into widespread use.

Today, although the underlying protocols used by the socket API have evolved over the years, and we've seen new ones, the low-level API has remained the same.

The most common type of socket applications are client-server applications, where one side acts as the server and waits for connections from clients. This is the type of application that I'll be covering in this tutorial. More specifically, we'll look at the socket API for [Internet sockets](#), sometimes called Berkeley or BSD sockets. There are also [Unix domain sockets](#), which can only be used to communicate between processes on the same host.

Socket API Overview

Python's [socket module](#) provides an interface to the [Berkeley sockets API](#). This is the module that we'll use and discuss in this tutorial.

The primary socket API functions and methods in this module are:

- `socket()`
- `bind()`

- `listen()`
- `accept()`
- `connect()`
- `connect_ex()`
- `send()`
- `recv()`
- `close()`

Python provides a convenient and consistent API that maps directly to these system calls, their C counterparts. We'll look at how these are used together in the next section.

As part of its standard library, Python also has classes that make using these low-level socket functions easier. Although it's not covered in this tutorial, see the [socketserver module](#), a framework for network servers. There are also many modules available that implement higher-level Internet protocols like HTTP and SMTP. For an overview, see [Internet Protocols and Support](#).

TCP Sockets

As you'll see shortly, we'll create a socket object using `socket.socket()` and specify the socket type as `socket.SOCK_STREAM`. When you do that, the default protocol that's used is the [Transmission Control Protocol \(TCP\)](#). This is a good default and probably what you want.

Why should you use TCP? The Transmission Control Protocol (TCP):

- **Is reliable:** packets dropped in the network are detected and retransmitted by the sender.
- **Has in-order data delivery:** data is read by your application in the order it was written by the sender.

In contrast, [User Datagram Protocol \(UDP\)](#) sockets created with `socket.SOCK_DGRAM` aren't reliable, and data read by the receiver can be out-of-order from the sender's writes.

Why is this important? Networks are a best-effort delivery system. There's no guarantee that your data will reach its destination or that you'll receive what's been sent to you.

Network devices (for example, routers and switches), have finite bandwidth available and their own inherent system limitations. They have CPUs, memory, buses, and interface packet buffers, just like our clients and servers. TCP relieves you from having to worry about [packet loss](#), data arriving out-of-order, and many other things that invariably happen when you're communicating across a network.

In the diagram below, let's look at the sequence of socket API calls and data flow for TCP:



TCP Socket Flow ([Image source](#))

The left-hand column represents the server. On the right-hand side is the client.

Starting in the top left-hand column, note the API calls the server makes to setup a “listening” socket:

- `socket()`
- `bind()`
- `listen()`
- `accept()`

A listening socket does just what it sounds like. It listens for connections from clients. When a client connects, the server calls `accept()` to accept, or complete, the connection.

The client calls `connect()` to establish a connection to the server and initiate the three-way handshake. The handshake step is important since it ensures that each side of the connection is reachable in the network, in other words that the client can reach the server and vice-versa. It may be that only one host, client or server, can reach the other.

In the middle is the round-trip section, where data is exchanged between the client and server using calls to `send()` and `recv()`.

At the bottom, the client and server `close()` their respective sockets.

Echo Client and Server

Now that you’ve seen an overview of the socket API and how the client and server communicate, let’s create our first client and server. We’ll begin with a simple implementation. The server will simply echo whatever it receives back to the client.

Echo Server

Here’s the server, `echo-server.py`:

Python

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1' # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
```

```
    ...
    break
conn.sendall(data)
```

Note: Don't worry about understanding everything above right now. There's a lot going on in these few lines of code. This is just a starting point so you can see a basic server in action.

There's a [reference section](#) at the end of this tutorial that has more information and links to additional resources. I'll link to these and other resources throughout the tutorial.

Let's walk through each API call and see what's happening.

`socket.socket()` creates a socket object that supports the [context manager type](#), so you can use it in a [with statement](#). There's no need to call `s.close()`:

Python

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    pass # Use the socket object without calling s.close().
```

The arguments passed to `socket()` specify the [address family](#) and socket type. `AF_INET` is the Internet address family for [IPv4](#). `SOCK_STREAM` is the socket type for [TCP](#), the protocol that will be used to transport our messages in the network.

`bind()` is used to associate the socket with a specific network interface and port number:

Python

```
HOST = '127.0.0.1' # Standard loopback interface address (localhost)
PORT = 65432         # Port to listen on (non-privileged ports are > 1023)

# ...

s.bind((HOST, PORT))
```

The values passed to `bind()` depend on the [address family](#) of the socket. In this example, we're using `socket.AF_INET` ([IPv4](#)). So it expects a 2-tuple: (host, port).

`host` can be a hostname, [IP address](#), or empty string. If an IP address is used, `host` should be an IPv4-formatted address string. The IP address `127.0.0.1` is the standard IPv4 address for the [loopback](#) interface, so only processes on the host will be able to connect to the server. If you pass an empty string, the server will accept connections on all available IPv4 interfaces.

`port` should be an integer from 1-65535 (0 is reserved). It's the [TCP port](#) number to accept connections on from clients. Some systems may require superuser privileges if the port is < 1024.

Here's a note on using hostnames with `bind()`:

"If you use a hostname in the host portion of IPv4/v6 socket address, the program may show a non-deterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion." ([Source](#))

I'll discuss this more later in [Using Hostnames](#), but it's worth mentioning here. For now, just understand that when using a hostname, you could see different results depending on what's returned from the name resolution process.

It could be anything. The first time you run your application, it might be the address `10.1.2.3`. The next time it's a different address, `192.168.0.1`. The third time, it could be `172.16.7.8`, and so on.

Continuing with the server example, `listen()` enables a server to accept() connections. It makes it a "listening" socket:

Python

```
s.listen()
conn, addr = s.accept()
```

`listen()` has a `backlog` parameter. It specifies the number of unaccepted connections that the system will allow before refusing new connections. Starting in Python 3.5, it's optional. If not specified, a default backlog value is chosen.

If your server receives a lot of connection requests simultaneously, increasing the `backlog` value may help by setting the maximum length of the queue for pending connections. The maximum value is system dependent. For example, on Linux, see [`/proc/sys/net/core/somaxconn`](#).

`accept()` [blocks](#) and waits for an incoming connection. When a client connects, it returns a new socket object representing the connection and a tuple holding the address of the client. The tuple will contain `(host, port)` for IPv4 connections or `(host, port, flowinfo, scopeid)` for IPv6. See [Socket Address Families](#) in the reference section for details on the tuple values.

One thing that's imperative to understand is that we now have a new socket object from `accept()`. This is important since it's the socket that you'll use to communicate with the client. It's distinct from the listening socket that the server is using to accept new connections:

Python

```
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
```

After getting the client socket object `conn` from `accept()`, an infinite `while` loop is used to loop over [blocking calls](#) to `conn.recv()`. This reads whatever data the client sends and echoes it back using `conn.sendall()`.

If `conn.recv()` returns an empty [bytes](#) object, `b''`, then the client closed the connection and the loop is terminated. The `with` statement is used with `conn` to automatically close the socket at the end of the block.

Echo Client

Now let's look at the client, `echo-client.py`:

Python

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1' # The server's hostname or IP address
PORT = 65432 # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

In comparison to the server, the client is pretty simple. It creates a socket object, connects to the server and calls `s.sendall()` to send its message. Lastly, it calls `s.recv()` to read the server's reply and then [prints it](#).

Running the Echo Client and Server

Let's run the client and server to see how they behave and inspect what's happening.

Note: If you're having trouble getting the examples or your own code to run from the command line, read [How Do I Make My Own Command-Line Commands Using Python?](#) If you're on Windows, check the [Python Windows FAQ](#).

Open a terminal or command prompt, navigate to the directory that contains your scripts, and run the server:

Shell

```
$ ./echo-server.py
```

Your terminal will appear to hang. That's because the server is [blocked](#) (suspended) in a call:

Python

```
conn, addr = s.accept()
```

It's waiting for a client connection. Now open another terminal window or command prompt and run the client:

Shell

```
$ ./echo-client.py
Received b'Hello, world'
```

In the server window, you should see:

Shell

```
$ ./echo-server.py
Connected by ('127.0.0.1', 64623)
```

In the output above, the server printed the `addr` tuple returned from `s.accept()`. This is the client's IP address and TCP port number. The port number, 64623, will most likely be different when you run it on your machine.

Viewing Socket State

To see the current state of sockets on your host, use `netstat`. It's available by default on macOS, Linux, and Windows.

Here's the `netstat` output from macOS after starting the server:

Shell

```
$ netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp4       0      0  127.0.0.1.65432        *.*                  LISTEN
```

Notice that `Local Address` is `127.0.0.1.65432`. If `echo-server.py` had used `HOST = ''` instead of `HOST = '127.0.0.1'`, `netstat` would show this:

Shell

```
$ netstat -an
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp4       0      0  *.65432              *.*                  LISTEN
```

`Local Address` is `*.65432`, which means all available host interfaces that support the address family will be used to accept incoming connections. In this example, in the call to `socket()`, `socket.AF_INET` was used (IPv4). You can see this in the `Proto` column: `tcp4`.

I've trimmed the output above to show the echo server only. You'll likely see much more output, depending on the system you're running it on. The things to notice are the columns `Proto`, `Local Address`, and `(state)`. In the last example above, `netstat` shows the echo server is using an IPv4 TCP socket (`tcp4`), on port 65432 on all interfaces (`*.65432`), and it's in the listening state (`LISTEN`).

Another way to see this, along with additional helpful information, is to use `lsof` (list open files). It's available by

default on macOS and can be installed on Linux using your package manager, if it's not already:

Shell

```
$ lsof -i -n
COMMAND   PID  USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
Python    67982 nathan  3u  IPv4  0xecf272      0t0  TCP *:65432 (LISTEN)
```

`lsof` gives you the COMMAND, PID (process id), and USER (user id) of open Internet sockets when used with the `-i` option. Above is the echo server process.

`netstat` and `lsof` have a lot of options available and differ depending on the OS you're running them on. Check the `man` page or documentation for both. They're definitely worth spending a little time with and getting to know. You'll be rewarded. On macOS and Linux, use `man netstat` and `man lsof`. For Windows, use `netstat /?`.

Here's a common error you'll see when a connection attempt is made to a port with no listening socket:

Shell

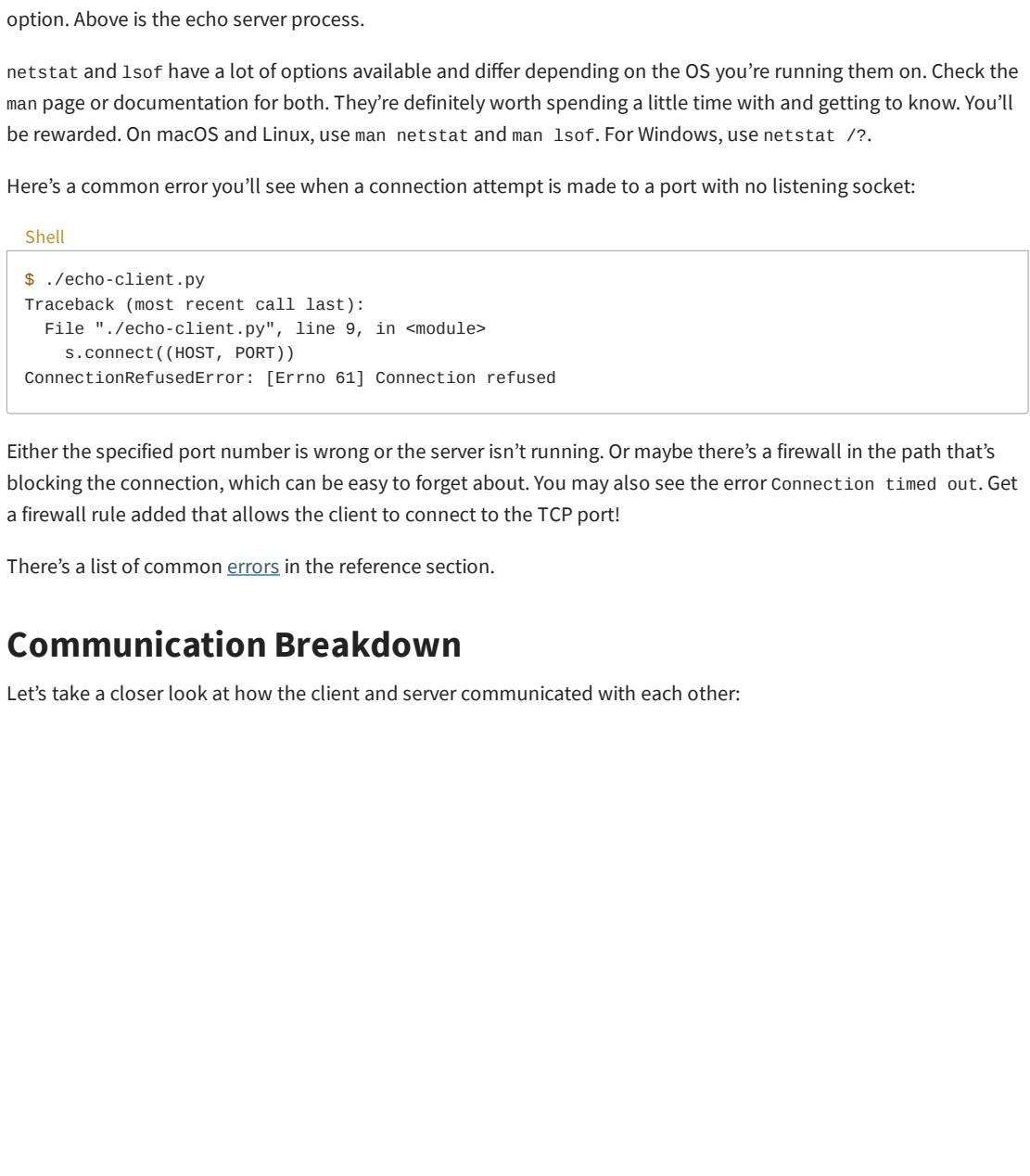
```
$ ./echo-client.py
Traceback (most recent call last):
  File "./echo-client.py", line 9, in <module>
    s.connect((HOST, PORT))
ConnectionRefusedError: [Errno 61] Connection refused
```

Either the specified port number is wrong or the server isn't running. Or maybe there's a firewall in the path that's blocking the connection, which can be easy to forget about. You may also see the error `Connection timed out`. Get a firewall rule added that allows the client to connect to the TCP port!

There's a list of common [errors](#) in the reference section.

Communication Breakdown

Let's take a closer look at how the client and server communicated with each other:



When using the `loopback` interface (IPv4 address `127.0.0.1` or IPv6 address `::1`), data never leaves the host or touches the external network. In the diagram above, the loopback interface is contained inside the host. This represents the internal nature of the loopback interface and that connections and data that transit it are local to the host. This is why you'll also hear the loopback interface and IP address `127.0.0.1` or `::1` referred to as "localhost."

Applications use the loopback interface to communicate with other processes running on the host and for security and isolation from the external network. Since it's internal and accessible only from within the host, it's not exposed.

You can see this in action if you have an application server that uses its own private database. If it's not a database used by other servers, it's probably configured to listen for connections on the loopback interface only. If this is the case, other hosts on the network can't connect to it.

When you use an IP address other than `127.0.0.1` or `::1` in your applications, it's probably bound to an [Ethernet](#) interface that's connected to an external network. This is your gateway to other hosts outside of your "localhost" kingdom:

Be careful out there. It's a nasty, cruel world. Be sure to read the section [Using Hostnames](#) before venturing from the safe confines of "localhost." There's a security note that applies even if you're not using hostnames and using IP addresses only.

Handling Multiple Connections

The echo server definitely has its limitations. The biggest being that it serves only one client and then exits. The echo client has this limitation too, but there's an additional problem. When the client makes the following call, it's possible that `s.recv()` will return only one byte, `b'H'` from `b'Hello, world'`:

Python

```
data = s.recv(1024)
```

The `bufsize` argument of `1024` used above is the maximum amount of data to be received at once. It doesn't mean that `recv()` will return `1024` bytes.

`send()` also behaves this way. `send()` returns the number of bytes sent, which may be less than the size of the data passed in. You're responsible for checking this and calling `send()` as many times as needed to send all of the data:

"Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data." ([Source](#))

We avoided having to do this by using `sendall()`:

"Unlike `send()`, this method continues to send data from bytes until either all data has been sent or an error occurs. None is returned on success." ([Source](#))

We have two problems at this point:

- How do we handle multiple connections concurrently?
- We need to call `send()` and `recv()` until all data is sent or received.

What do we do? There are many approaches to [concurrency](#). More recently, a popular approach is to use [Asynchronous I/O](#). `asyncio` was introduced into the standard library in Python 3.4. The traditional choice is to use [threads](#).

The trouble with concurrency is it's hard to get right. There are many subtleties to consider and guard against. All it takes is for one of these to manifest itself and your application may suddenly fail in not-so-subtle ways.

I don't say this to scare you away from learning and using concurrent programming. If your application needs to scale, it's a necessity if you want to use more than one processor or one core. However, for this tutorial, we'll use

something that's more traditional than threads and easier to reason about. We're going to use the granddaddy of system calls: [select\(\)](#).

`select()` allows you to check for I/O completion on more than one socket. So you can call `select()` to see which sockets have I/O ready for reading and/or writing. But this is Python, so there's more. We're going to use the [selectors](#) module in the standard library so the most efficient implementation is used, regardless of the operating system we happen to be running on:

“This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.” ([Source](#))

Even though, by using `select()`, we're not able to run concurrently, depending on your workload, this approach may still be plenty fast. It depends on what your application needs to do when it services a request and the number of clients it needs to support.

[asyncio](#) uses single-threaded cooperative multitasking and an event loop to manage tasks. With `select()`, we'll be writing our own version of an event loop, albeit more simply and synchronously. When using multiple threads, even though you have concurrency, we currently have to use the [GIL](#) with [CPython and PyPy](#). This effectively limits the amount of work we can do in parallel anyway.

I say all of this to explain that using `select()` may be a perfectly fine choice. Don't feel like you have to use `asyncio`, threads, or the latest asynchronous library. Typically, in a network application, your application is I/O bound: it could be waiting on the local network, endpoints on the other side of the network, on a disk, and so forth.

If you're getting requests from clients that initiate CPU bound work, look at the [concurrent.futures](#) module. It contains the class [ProcessPoolExecutor](#) that uses a pool of processes to execute calls asynchronously.

If you use multiple processes, the operating system is able to schedule your Python code to run in parallel on multiple processors or cores, without the GIL. For ideas and inspiration, see the PyCon talk [John Reese - Thinking Outside the GIL with AsyncIO and Multiprocessing - PyCon 2018](#).

In the next section, we'll look at examples of a server and client that address these problems. They use `select()` to handle multiple connections simultaneously and call `send()` and `recv()` as many times as needed.

Multi-Connection Client and Server

In the next two sections, we'll create a server and client that handles multiple connections using a selector object created from the [selectors](#) module.

Multi-Connection Server

First, let's look at the multi-connection server, `multiconn-server.py`. Here's the first part that sets up the listening socket:

Python

```
import selectors
sel = selectors.DefaultSelector()
# ...
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print('listening on', (host, port))
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ, data=None)
```

The biggest difference between this server and the echo server is the call to `lsock.setblocking(False)` to configure the socket in non-blocking mode. Calls made to this socket will no longer [block](#). When it's used with `sel.select()`, as you'll see below, we can wait for events on one or more sockets and then read and write data when it's ready.

`sel.register()` registers the socket to be monitored with `sel.select()` for the events you're interested in. For the

`sel.register()` registers the socket to be monitored with `sel.select()` for the events you're interested in. For the listening socket, we want read events: `selectors.EVENT_READ`.

`data` is used to store whatever arbitrary data you'd like along with the socket. It's returned when `select()` returns. We'll use `data` to keep track of what's been sent and received on the socket.

Next is the event loop:

Python

```
import selectors
sel = selectors.DefaultSelector()

# ...

while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        if key.data is None:
            accept_wrapper(key.fileobj)
        else:
            service_connection(key, mask)
```

`sel.select(timeout=None).blocks` until there are sockets ready for I/O. It returns a list of (`key, events`) tuples, one for each socket. `key` is a `SelectorKey` namedtuple that contains a `fileobj` attribute. `key.fileobj` is the socket object, and `mask` is an event mask of the operations that are ready.

If `key.data` is `None`, then we know it's from the listening socket and we need to `accept()` the connection. We'll call our own `accept()` wrapper function to get the new socket object and register it with the selector. We'll look at it in a moment.

If `key.data` is not `None`, then we know it's a client socket that's already been accepted, and we need to service it. `service_connection()` is then called and passed `key` and `mask`, which contains everything we need to operate on the socket.

Let's look at what our `accept_wrapper()` function does:

Python

```
def accept_wrapper(sock):
    conn, addr = sock.accept() # Should be ready to read
    print('accepted connection from', addr)
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b'', outb=b'')
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)
```

Since the listening socket was registered for the event `selectors.EVENT_READ`, it should be ready to read. We call `sock.accept()` and then immediately call `conn.setblocking(False)` to put the socket in non-blocking mode.

Remember, this is the main objective in this version of the server since we don't want it to block. If it blocks, then the entire server is stalled until it returns. Which means other sockets are left waiting. This is the dreaded "hang" state that you don't want your server to be in.

Next, we create an object to hold the data we want included along with the socket using the class `types.SimpleNamespace`. Since we want to know when the client connection is ready for reading and writing, both of those events are set using the following:

Python

```
events = selectors.EVENT_READ | selectors.EVENT_WRITE
```

The `events`, `socket`, and `data` objects are then passed to `sel.register()`.

Now let's look at `service_connection()` to see how a client connection is handled when it's ready:

Python

```
def service_connection(key, mask):
    sock = key.fileobj
```

```

data = key.data
if mask & selectors.EVENT_READ:
    recv_data = sock.recv(1024) # Should be ready to read
    if recv_data:
        data.outb += recv_data
    else:
        print('closing connection to', data.addr)
        sel.unregister(sock)
        sock.close()
if mask & selectors.EVENT_WRITE:
    if data.outb:
        print('echoing', repr(data.outb), 'to', data.addr)
        sent = sock.send(data.outb) # Should be ready to write
        data.outb = data.outb[sent:]

```

This is the heart of the simple multi-connection server. `key` is the namedtuple returned from `select()` that contains the socket object (`fileobj`) and data object. `mask` contains the events that are ready.

If the socket is ready for reading, then `mask & selectors.EVENT_READ` is true, and `sock.recv()` is called. Any data that's read is appended to `data.outb` so it can be sent later.

Note the `else:` block if no data is received:

Python

```

if recv_data:
    data.outb += recv_data
else:
    print('closing connection to', data.addr)
    sel.unregister(sock)
    sock.close()

```

This means that the client has closed their socket, so the server should too. But don't forget to first call `sel.unregister()` so it's no longer monitored by `select()`.

When the socket is ready for writing, which should always be the case for a healthy socket, any received data stored in `data.outb` is echoed to the client using `sock.send()`. The bytes sent are then removed from the send buffer:

Python

```
data.outb = data.outb[sent:]
```

Multi-Connection Client

Now let's look at the multi-connection client, `multiconn-client.py`. It's very similar to the server, but instead of listening for connections, it starts by initiating connections via `start_connections()`:

Python

```

messages = [b'Message 1 from client.', b'Message 2 from client.']

def start_connections(host, port, num_conns):
    server_addr = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
        print('starting connection', connid, 'to', server_addr)
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setblocking(False)
        sock.connect_ex(server_addr)
        events = selectors.EVENT_READ | selectors.EVENT_WRITE
        data = types.SimpleNamespace(connid=connid,
                                    msg_total=sum(len(m) for m in messages),
                                    recv_total=0,
                                    messages=list(messages),
                                    outb=b'')
        sel.register(sock, events, data=data)

```

`num_conns` is read from the command-line, which is the number of connections to create to the server. Just like the server, each socket is set to non-blocking mode.

`connect_ex()` is used instead of `connect()` since `connect()` would immediately raise a `BlockingIOError` exception. `connect_ex()` initially returns an error indicator, `errno.EINPROGRESS`, instead of raising an exception while the connection is in progress. Once the connection is completed, the socket is ready for reading and writing and is returned as such by `select()`.

After the socket is setup, the data we want stored with the socket is created using the class `types.SimpleNamespace`. The messages the client will send to the server are copied using `list(messages)` since each connection will call `socket.send()` and modify the list. Everything needed to keep track of what the client needs to send, has sent and received, and the total number of bytes in the messages is stored in the object data.

Let's look at `service_connection()`. It's fundamentally the same as the server:

Python

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024) # Should be ready to read
        if recv_data:
            print('received', repr(recv_data), 'from connection', data.connid)
            data.recv_total += len(recv_data)
        if not recv_data or data.recv_total == data.msg_total:
            print('closing connection', data.connid)
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if not data.outb and data.messages:
            data.outb = data.messages.pop(0)
        if data.outb:
            print('sending', repr(data.outb), 'to connection', data.connid)
            sent = sock.send(data.outb) # Should be ready to write
            data.outb = data.outb[sent:]
```

There's one important difference. It keeps track of the number of bytes it's received from the server so it can close its side of the connection. When the server detects this, it closes its side of the connection too.

Note that by doing this, the server depends on the client being well-behaved: the server expects the client to close its side of the connection when it's done sending messages. If the client doesn't close, the server will leave the connection open. In a real application, you may want to guard against this in your server and prevent client connections from accumulating if they don't send a request after a certain amount of time.

Running the Multi-Connection Client and Server

Now let's run `multiconn-server.py` and `multiconn-client.py`. They both use command-line arguments. You can run them without arguments to see the options.

For the server, pass a host and port number:

Shell

```
$ ./multiconn-server.py
usage: ./multiconn-server.py <host> <port>
```

For the client, also pass the number of connections to create to the server, `num_connections`:

Shell

```
$ ./multiconn-client.py
usage: ./multiconn-client.py <host> <port> <num_connections>
```

Below is the server output when listening on the loopback interface on port 65432:

Shell

```
$ ./multiconn-server.py 127.0.0.1 65432
listening on ('127.0.0.1', 65432)
accepted connection from ('127.0.0.1', 61354)
accepted connection from ('127.0.0.1', 61355)
echoing b'Message 1 from client. Message 2 from client.' to ('127.0.0.1', 61354)
echoing b'Message 1 from client. Message 2 from client.' to ('127.0.0.1', 61355)
closing connection to ('127.0.0.1', 61354)
closing connection to ('127.0.0.1', 61355)
```

Below is the client output when it creates two connections to the server above:

Shell

```
$ ./multiconn-client.py 127.0.0.1 65432 2
starting connection 1 to ('127.0.0.1', 65432)
starting connection 2 to ('127.0.0.1', 65432)
sending b'Message 1 from client.' to connection 1
sending b'Message 2 from client.' to connection 1
sending b'Message 1 from client.' to connection 2
sending b'Message 2 from client.' to connection 2
received b'Message 1 from client. Message 2 from client.' from connection 1
closing connection 1
received b'Message 1 from client. Message 2 from client.' from connection 2
closing connection 2
```

Application Client and Server

The multi-connection client and server example is definitely an improvement compared with where we started. However, let's take one more step and address the shortcomings of the previous "multiconn" example in a final implementation: the application client and server.

We want a client and server that handles errors appropriately so other connections aren't affected. Obviously, our client or server shouldn't come crashing down in a ball of fury if an exception isn't caught. This is something we haven't discussed up until now. I've intentionally left out error handling for brevity and clarity in the examples.

Now that you're familiar with the basic API, non-blocking sockets, and `select()`, we can add some error handling and discuss the "elephant in the room" that I've kept hidden from you behind that large curtain over there. Yes, I'm talking about the custom class I mentioned way back in the introduction. I knew you wouldn't forget.

First, let's address the errors:

"All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise `OSError` or one of its subclasses." ([Source](#))

We need to catch `OSError`. Another thing I haven't mentioned in relation to errors is timeouts. You'll see them discussed in many places in the documentation. Timeouts happen and are a "normal" error. Hosts and routers are rebooted, switch ports go bad, cables go bad, cables get unplugged, you name it. You should be prepared for these and other errors and handle them in your code.

What about the "elephant in the room?" As hinted by the socket type `socket.SOCK_STREAM`, when using TCP, you're reading from a continuous stream of bytes. It's like reading from a file on disk, but instead you're reading bytes from the network.

However, unlike reading a file, there's no `f.seek()`. In other words, you can't reposition the socket pointer, if there was one, and move randomly around the data reading whatever, whenever you'd like.

When bytes arrive at your socket, there are network buffers involved. Once you've read them, they need to be saved somewhere. Calling `recv()` again reads the next stream of bytes available from the socket.

What this means is that you'll be reading from the socket in chunks. You need to call `recv()` and save the data in a buffer until you've read enough bytes to have a complete message that makes sense to your application.

It's up to you to define and keep track of where the message boundaries are. As far as the TCP socket is concerned, it's just sending and receiving raw bytes to and from the network. It knows nothing about what those raw bytes mean.

This brings us to defining an application-layer protocol. What's an application-layer protocol? Put simply, your application will send and receive messages. These messages are your application's protocol.

In other words, the length and format you choose for these messages define the semantics and behavior of your application. This is directly related to what I explained in the previous paragraph regarding reading bytes from the socket. When you're reading bytes with `recv()`, you need to keep up with how many bytes were read and figure out where the message boundaries are.

How is this done? One way is to always send fixed-length messages. If they're always the same size, then it's easy. When you've read that number of bytes into a buffer, then you know you have one complete message.

However, using fixed-length messages is inefficient for small messages where you'd need to use padding to fill them out. Also, you're still left with the problem of what to do about data that doesn't fit into one message.

In this tutorial, we'll take a generic approach. An approach that's used by many protocols, including HTTP. We'll prefix messages with a header that includes the content length as well as any other fields we need. By doing this, we'll only need to keep up with the header. Once we've read the header, we can process it to determine the length of the message's content and then read that number of bytes to consume it.

We'll implement this by creating a custom class that can send and receive messages that contain text or binary data. You can improve and extend it for your own applications. The most important thing is that you'll be able to see an example of how this is done.

I need to mention something regarding sockets and bytes that may affect you. As we talked about earlier, when sending and receiving data via sockets, you're sending and receiving raw bytes.

If you receive data and want to use it in a context where it's interpreted as multiple bytes, for example a 4-byte integer, you'll need to take into account that it could be in a format that's not native to your machine's CPU. The client or server on the other end could have a CPU that uses a different byte order than your own. If this is the case, you'll need to convert it to your host's native byte order before using it.

This byte order is referred to as a CPU's [endianness](#). See [Byte Endianness](#) in the reference section for details. We'll avoid this issue by taking advantage of [Unicode](#) for our message header and using the encoding UTF-8. Since UTF-8 uses an 8-bit encoding, there are no byte ordering issues.

You can find an explanation in Python's [Encodings and Unicode](#) documentation. Note that this applies to the text header only. We'll use an explicit type and encoding defined in the header for the content that's being sent, the message payload. This will allow us to transfer any data we'd like (text or binary), in any format.

You can easily determine the byte order of your machine by using `sys.byteorder`. For example, on my Intel laptop, this happens:

Shell

```
$ python3 -c 'import sys; print(repr(sys.byteorder))'  
'little'
```

If I run this in a virtual machine that [emulates](#) a big-endian CPU (PowerPC), then this happens:

Shell

```
$ python3 -c 'import sys; print(repr(sys.byteorder))'  
'big'
```

In this example application, our application-layer protocol defines the header as Unicode text with a UTF-8 encoding. For the actual content in the message, the message payload, you'll still have to swap the byte order manually if needed.

This will depend on your application and whether or not it needs to process multi-byte binary data from a machine

with a different endianness. You can help your client or server implement binary support by adding additional headers and using them to pass parameters, similar to HTTP.

Don't worry if this doesn't make sense yet. In the next section, you'll see how all of this works and fits together.

Application Protocol Header

Let's fully define the protocol header. The protocol header is:

- Variable-length text
- Unicode with the encoding UTF-8
- A Python dictionary serialized using [JSON](#)

The required headers, or sub-headers, in the protocol header's dictionary are as follows:

Name	Description
byteorder	The byte order of the machine (uses <code>sys.byteorder</code>). This may not be required for your application.
content-length	The length of the content in bytes.
content-type	The type of content in the payload, for example, <code>text/json</code> or <code>binary/my-binary-type</code> .
content-encoding	The encoding used by the content, for example, <code>utf-8</code> for Unicode text or <code>binary</code> for binary data.

These headers inform the receiver about the content in the payload of the message. This allows you to send arbitrary data while providing enough information so the content can be decoded and interpreted correctly by the receiver. Since the headers are in a dictionary, it's easy to add additional headers by inserting key/value pairs as needed.

Sending an Application Message

There's still a bit of a problem. We have a variable-length header, which is nice and flexible, but how do you know the length of the header when reading it with `recv()`?

When we previously talked about using `recv()` and message boundaries, I mentioned that fixed-length headers can be inefficient. That's true, but we're going to use a small, 2-byte, fixed-length header to prefix the JSON header that contains its length.

You can think of this as a hybrid approach to sending messages. In effect, we're bootstrapping the message receive process by sending the length of the header first. This makes it easy for our receiver to deconstruct the message.

To give you a better idea of the message format, let's look at a message in its entirety:

A message starts with a fixed-length header of 2 bytes that's an integer in network byte order. This is the length of the next header, the variable-length JSON header. Once we've read 2 bytes with `recv()`, then we know we can process the 2 bytes as an integer and then read that number of bytes before decoding the UTF-8 JSON header.

The [JSON header](#) contains a dictionary of additional headers. One of those is `content-length`, which is the number of bytes of the message's content (not including the JSON header). Once we've called `recv()` and read `content-length` bytes, we've reached a message boundary and read an entire message.

Application Message Class

Finally, the payoff! Let's look at the `Message` class and see how it's used with `select()` when read and write events happen on the socket.

For this example application, I had to come up with an idea for what types of messages the client and server would use. We're far beyond toy echo clients and servers at this point.

To keep things simple and still demonstrate how things would work in a real application, I created an application protocol that implements a basic search feature. The client sends a search request and the server does a lookup for a match. If the request sent by the client isn't recognized as a search, the server assumes it's a binary request and returns a binary response.

After reading the following sections, running the examples, and experimenting with the code, you'll see how things work. You can then use the `Message` class as a starting point and modify it for your own use.

We're really not that far off from the "multicomm" client and server example. The event loop code stays the same in `app-client.py` and `app-server.py`. What I've done is move the message code into a class named `Message` and added methods to support reading, writing, and processing of the headers and content. This is a great example for using a class.

As we discussed before and you'll see below, working with sockets involves keeping state. By using a class, we keep all of the state, data, and code bundled together in an organized unit. An instance of the class is created for each socket in the client and server when a connection is started or accepted.

The class is mostly the same for both the client and the server for the wrapper and utility methods. They start with an underscore, like `Message._json_encode()`. These methods simplify working with the class. They help other methods by allowing them to stay shorter and support the [DRY](#) principle.

The server's `Message` class works in essentially the same way as the client's and vice-versa. The difference being that the client initiates the connection and sends a request message, followed by processing the server's response

message. Conversely, the server waits for a connection, processes the client's request message, and then sends a response message.

It looks like this:

Step	Endpoint	Action / Message Content
1	Client	Sends a Message containing request content
2	Server	Receives and processes client request Message
3	Server	Sends a Message containing response content
4	Client	Receives and processes server response Message

Here's the file and code layout:

Application	File	Code
Server	app-server.py	The server's main script
Server	libserver.py	The server's Message class
Client	app-client.py	The client's main script
Client	libclient.py	The client's Message class

Message Entry Point

I'd like to discuss how the Message class works by first mentioning an aspect of its design that wasn't immediately obvious to me. Only after refactoring it at least five times did I arrive at what it is currently. Why? Managing state.

After a Message object is created, it's associated with a socket that's monitored for events using `selector.register()`:

Python

```
message = libserver.Message(sel, conn, addr)
sel.register(conn, selectors.EVENT_READ, data=message)
```

Note: Some of the code examples in this section are from the server's main script and Message class, but this section and discussion applies equally to the client as well. I'll show and explain the client's version when it differs.

When events are ready on the socket, they're returned by `selector.select()`. We can then get a reference back to the message object using the `data` attribute on the key object and call a method in Message:

Python

```
while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        # ...
        message = key.data
        message.process_events(mask)
```

Looking at the event loop above, you'll see that `sel.select()` is in the driver's seat. It's blocking, waiting at the top of the loop for events. It's responsible for waking up when read and write events are ready to be processed on the socket. Which means, indirectly, it's also responsible for calling the method `process_events()`. This is what I mean when I say the method `process_events()` is the entry point.

Let's see what the `process_events()` method does.

Let's see what the `process_events()` method does.

Python

```
def process_events(self, mask):
    if mask & selectors.EVENT_READ:
        self.read()
    if mask & selectors.EVENT_WRITE:
        self.write()
```

That's good: `process_events()` is simple. It can only do two things: call `read()` and `write()`.

This brings us back to managing state. After a few refactorings, I decided that if another method depended on state variables having a certain value, then they would only be called from `read()` and `write()`. This keeps the logic as simple as possible as events come in on the socket for processing.

This may seem obvious, but the first few iterations of the class were a mix of some methods that checked the current state and, depending on their value, called other methods to process data outside `read()` or `write()`. In the end, this proved too complex to manage and keep up with.

You should definitely modify the class to suit your own needs so it works best for you, but I'd recommend that you keep the state checks and the calls to methods that depend on that state to the `read()` and `write()` methods if possible.

Let's look at `read()`. This is the server's version, but the client's is the same. It just uses a different method name, `process_response()` instead of `process_request()`:

Python

```
def read(self):
    self._read()

    if self._jsonheader_len is None:
        self._process_protoheader()

    if self._jsonheader_len is not None:
        if self.jsonheader is None:
            self._process_jsonheader()

    if self.jsonheader:
        if self.request is None:
            self._process_request()
```

The `_read()` method is called first. It calls `socket.recv()` to read data from the socket and store it in a receive buffer.

Remember that when `socket.recv()` is called, all of the data that makes up a complete message may not have arrived yet. `socket.recv()` may need to be called again. This is why there are state checks for each part of the message before calling the appropriate method to process it.

Before a method processes its part of the message, it first checks to make sure enough bytes have been read into the receive buffer. If there are, it processes its respective bytes, removes them from the buffer and writes its output to a variable that's used by the next processing stage. Since there are three components to a message, there are three state checks and process method calls:

Message Component	Method	Output
Fixed-length header	<code>process_protoheader()</code>	<code>self._jsonheader_len</code>
JSON header	<code>process_jsonheader()</code>	<code>self.jsonheader</code>
Content	<code>process_request()</code>	<code>self.request</code>

Next, let's look at `write()`. This is the server's version:

Python

```
def write(self):
    if self.request:
        if not self.response_created:
            self.create_response()

    self._write()
```

`write()` checks first for a request. If one exists and a response hasn't been created, `create_response()` is called. `create_response()` sets the state variable `response_created` and writes the response to the send buffer.

The `_write()` method calls `socket.send()` if there's data in the send buffer.

Remember that when `socket.send()` is called, all of the data in the send buffer may not have been queued for transmission. The network buffers for the socket may be full, and `socket.send()` may need to be called again. This is why there are state checks. `create_response()` should only be called once, but it's expected that `_write()` will need to be called multiple times.

The client version of `write()` is similar:

Python

```
def write(self):
    if not self._request_queued:
        self.queue_request()

    self._write()

    if self._request_queued:
        if not self._send_buffer:
            # Set selector to listen for read events, we're done writing.
            self._set_selector_events_mask('r')
```

Since the client initiates a connection to the server and sends a request first, the state variable `_request_queued` is checked. If a request hasn't been queued, it calls `queue_request()`. `queue_request()` creates the request and writes it to the send buffer. It also sets the state variable `_request_queued` so it's only called once.

Just like the server, `_write()` calls `socket.send()` if there's data in the send buffer.

The notable difference in the client's version of `write()` is the last check to see if the request has been queued. This will be explained more in the section [Client Main Script](#), but the reason for this is to tell `selector.select()` to stop monitoring the socket for write events. If the request has been queued and the send buffer is empty, then we're done writing and we're only interested in read events. There's no reason to be notified that the socket is writable.

I'll wrap up this section by leaving you with one thought. The main purpose of this section was to explain that `selector.select()` is calling into the `Message` class via the method `process_events()` and to describe how state is managed.

This is important because `process_events()` will be called many times over the life of the connection. Therefore, make sure that any methods that should only be called once are either checking a state variable themselves, or the state variable set by the method is checked by the caller.

Server Main Script

In the server's main script `app-server.py`, arguments are read from the command line that specify the interface and port to listen on:

Shell

```
$ ./app-server.py
usage: ./app-server.py <host> <port>
```

For example, to listen on the loopback interface on port 65432, enter:

Shell

```
$ ./app-server.py 127.0.0.1 65432
```

```
listening on ('127.0.0.1', 65432)
```

Use an empty string for <host> to listen on all interfaces.

After creating the socket, a call is made to `socket.setsockopt()` with the option `socket.SO_REUSEADDR`:

Python

```
# Avoid bind() exception: OSError: [Errno 48] Address already in use
lsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Setting this socket option avoids the error `Address already in use`. You'll see this when starting the server and a previously used TCP socket on the same port has connections in the [TIME_WAIT](#) state.

For example, if the server actively closed a connection, it will remain in the `TIME_WAIT` state for two minutes or more, depending on the operating system. If you try to start the server again before the `TIME_WAIT` state expires, you'll get an `OSError` exception of `Address already in use`. This is a safeguard to make sure that any delayed packets in the network aren't delivered to the wrong application.

The event loop catches any errors so the server can stay up and continue to run:

Python

```
while True:
    events = sel.select(timeout=None)
    for key, mask in events:
        if key.data is None:
            accept_wrapper(key.fileobj)
        else:
            message = key.data
            try:
                message.process_events(mask)
            except Exception:
                print('main: error: exception for',
                      f'{message.addr}: {traceback.format_exc()}')
            message.close()
```

When a client connection is accepted, a `Message` object is created:

Python

```
def accept_wrapper(sock):
    conn, addr = sock.accept() # Should be ready to read
    print('accepted connection from', addr)
    conn.setblocking(False)
    message = libserver.Message(sel, conn, addr)
    sel.register(conn, selectors.EVENT_READ, data=message)
```

The `Message` object is associated with the socket in the call to `sel.register()` and is initially set to be monitored for read events only. Once the request has been read, we'll modify it to listen for write events only.

An advantage of taking this approach in the server is that in most cases, when a socket is healthy and there are no network issues, it will always be writable.

If we told `sel.register()` to also monitor `EVENT_WRITE`, the event loop would immediately wakeup and notify us that this is the case. However, at this point, there's no reason to wake up and call `send()` on the socket. There's no response to send since a request hasn't been processed yet. This would consume and waste valuable CPU cycles.

Server Message Class

In the section [Message Entry Point](#), we looked at how the `Message` object was called into action when socket events were ready via `process_events()`. Now let's look at what happens as data is read on the socket and a component, or piece, of the message is ready to be processed by the server.

The server's message class is in `libserver.py`. You can find the [source code on GitHub](#).

The methods appear in the class in the order in which processing takes place for a message.

When the server has read at least 2 bytes, the fixed-length header can be processed:

Python

```
def process_protoheader(self):
    hdrlen = 2
    if len(self._recv_buffer) >= hdrlen:
        self._jsonheader_len = struct.unpack('>H',
                                              self._recv_buffer[:hdrlen])[0]
        self._recv_buffer = self._recv_buffer[hdrlen:]
```

The fixed-length header is a 2-byte integer in network (big-endian) byte order that contains the length of the JSON header. `struct.unpack()` is used to read the value, decode it, and store it in `self._jsonheader_len`. After processing the piece of the message it's responsible for, `process_protoheader()` removes it from the receive buffer.

Just like the fixed-length header, when there's enough data in the receive buffer to contain the JSON header, it can be processed as well:

Python

```
def process_jsonheader(self):
    hdrlen = self._jsonheader_len
    if len(self._recv_buffer) >= hdrlen:
        self.jsonheader = self._json_decode(self._recv_buffer[:hdrlen],
                                             'utf-8')
        self._recv_buffer = self._recv_buffer[hdrlen:]
        for reqhdr in ('byteorder', 'content-length', 'content-type',
                       'content-encoding'):
            if reqhdr not in self.jsonheader:
                raise ValueError(f'Missing required header "{reqhdr}".')
```

The method `self._json_decode()` is called to decode and deserialize the JSON header into a dictionary. Since the JSON header is defined as Unicode with a UTF-8 encoding, `utf-8` is hardcoded in the call. The result is saved to `self.jsonheader`. After processing the piece of the message it's responsible for, `process_jsonheader()` removes it from the receive buffer.

Next is the actual content, or payload, of the message. It's described by the JSON header in `self.jsonheader`. When `content-length` bytes are available in the receive buffer, the request can be processed:

Python

```
def process_request(self):
    content_len = self.jsonheader['content-length']
    if not len(self._recv_buffer) >= content_len:
        return
    data = self._recv_buffer[:content_len]
    self._recv_buffer = self._recv_buffer[content_len:]
    if self.jsonheader['content-type'] == 'text/json':
        encoding = self.jsonheader['content-encoding']
        self.request = self._json_decode(data, encoding)
        print('received request', repr(self.request), 'from', self.addr)
    else:
        # Binary or unknown content-type
        self.request = data
        print(f'received {self.jsonheader["content-type"]} request from',
              self.addr)
    # Set selector to listen for write events, we're done reading.
    self._set_selector_events_mask('w')
```

After saving the message content to the `data` variable, `process_request()` removes it from the receive buffer. Then, if the content type is JSON, it decodes and deserializes it. If it's not, for this example application, it assumes it's a binary request and simply prints the content type.

The last thing process_request() does is modify the selector to monitor write events only. In the server's main script, app-server.py, the socket is initially set to monitor read events only. Now that the request has been fully processed, we're no longer interested in reading.

A response can now be created and written to the socket. When the socket is writable, create_response() is called from write():

Python

```
def create_response(self):
    if self.jsonheader['content-type'] == 'text/json':
        response = self._create_response_json_content()
    else:
        # Binary or unknown content-type
        response = self._create_response_binary_content()
    message = self._create_message(**response)
    self.response_created = True
    self._send_buffer += message
```

A response is created by calling other methods, depending on the content type. In this example application, a simple dictionary lookup is done for JSON requests when action == 'search'. You can define other methods for your own applications that get called here.

After creating the response message, the state variable self.response_created is set so write() doesn't call create_response() again. Finally, the response is appended to the send buffer. This is seen by and sent via _write().

One tricky bit to figure out was how to close the connection after the response is written. I put the call to close() in the method _write():

Python

```
def _write(self):
    if self._send_buffer:
        print('sending', repr(self._send_buffer), 'to', self.addr)
        try:
            # Should be ready to write
            sent = self.sock.send(self._send_buffer)
        except BlockingIOError:
            # Resource temporarily unavailable (errno EWOULDBLOCK)
            pass
        else:
            self._send_buffer = self._send_buffer[sent:]
            # Close when the buffer is drained. The response has been sent.
            if sent and not self._send_buffer:
                self.close()
```

Although it's somewhat "hidden," I think it's an acceptable trade-off given that the Message class only handles one message per connection. After the response is written, there's nothing left for the server to do. It's completed its work.

Client Main Script

In the client's main script app-client.py, arguments are read from the command line and used to create requests and start connections to the server:

Shell

```
$ ./app-client.py
usage: ./app-client.py <host> <port> <action> <value>
```

Here's an example:

Shell

```
$ ./app-client.py 127.0.0.1 65432 search needle
```

After creating a dictionary representing the request from the command-line arguments, the host, port, and request dictionary are passed to `start_connection()`:

Python

```
def start_connection(host, port, request):
    addr = (host, port)
    print('starting connection to', addr)
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setblocking(False)
    sock.connect_ex(addr)
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    message = libclient.Message(sel, sock, addr, request)
    sel.register(sock, events, data=message)
```

A socket is created for the server connection as well as a `Message` object using the `request` dictionary.

Like the server, the `Message` object is associated with the socket in the call to `sel.register()`. However, for the client, the socket is initially set to be monitored for both read and write events. Once the request has been written, we'll modify it to listen for read events only.

This approach gives us the same advantage as the server: not wasting CPU cycles. After the request has been sent, we're no longer interested in write events, so there's no reason to wake up and process them.

Client Message Class

In the section [Message Entry Point](#), we looked at how the `message` object was called into action when socket events were ready via `process_events()`. Now let's look at what happens after data is read and written on the socket and a message is ready to be processed by the client.

The client's message class is in `libclient.py`. You can find the [source code on GitHub](#).

The methods appear in the class in the order in which processing takes place for a message.

The first task for the client is to queue the request:

Python

```
def queue_request(self):
    content = self.request['content']
    content_type = self.request['type']
    content_encoding = self.request['encoding']
    if content_type == 'text/json':
        req = {
            'content_bytes': self._json_encode(content, content_encoding),
            'content_type': content_type,
            'content_encoding': content_encoding
        }
    else:
        req = {
            'content_bytes': content,
            'content_type': content_type,
            'content_encoding': content_encoding
        }
    message = self._create_message(**req)
    self._send_buffer += message
    self._request_queued = True
```

The dictionaries used to create the request, depending on what was passed on the command line, are in the client's main script, `app-client.py`. The request dictionary is passed as an argument to the class when a `Message` object is created.

The request message is created and appended to the send buffer, which is then seen by and sent via `_write()`. The state variable `self._request_queued` is set so `queue_request()` isn't called again.

After the request has been sent, the client waits for a response from the server.

The methods for reading and processing a message in the client are the same as the server. As response data is read from the socket, the process header methods are called: `process_protoheader()` and `process_jsonheader()`.

The difference is in the naming of the final process methods and the fact that they're processing a response, not creating one: `process_response()`, `_process_response_json_content()`, and `_process_response_binary_content()`.

Last, but certainly not least, is the final call for `process_response()`:

Python

```
def process_response(self):
    ...
    # Close when response has been processed
    self.close()
```

Message Class Wrapup

I'll conclude the `Message` class discussion by mentioning a couple of things that are important to notice with a few of the supporting methods.

Any exceptions raised by the class are caught by the main script in its `except` clause:

Python

```
try:
    message.process_events(mask)
except Exception:
    print('main: error: exception for',
          f'{message.addr}: {traceback.format_exc()}')
    message.close()
```

Note the last line: `message.close()`.

This is a really important line, for more than one reason! Not only does it make sure that the socket is closed, but `message.close()` also removes the socket from being monitored by `select()`. This greatly simplifies the code in the class and reduces complexity. If there's an exception or we explicitly raise one ourselves, we know `close()` will take care of the cleanup.

The methods `Message._read()` and `Message._write()` also contain something interesting:

Python

```
def _read(self):
    try:
        # Should be ready to read
        data = self.sock.recv(4096)
    except BlockingIOError:
        # Resource temporarily unavailable (errno EWOULDBLOCK)
        pass
    else:
        if data:
            self._recv_buffer += data
        else:
            raise RuntimeError('Peer closed.')
```

Note the `except` line: `except BlockingIOError:`.

`_write()` has one too. These lines are important because they catch a temporary error and skip over it using `pass`. The temporary error is when the socket would [block](#), for example if it's waiting on the network or the other end of the connection (its peer).

By catching and skipping over the exception with `pass`, `select()` will eventually call us again, and we'll get another chance to read or write the data.

Running the Application Client and Server

After all of this hard work, let's have some fun and run some searches!

In these examples, I'll run the server so it listens on all interfaces by passing an empty string for the host argument. This will allow me to run the client and connect from a virtual machine that's on another network. It emulates a big-endian PowerPC machine.

First, let's start the server:

Shell

```
$ ./app-server.py '' 65432
listening on ('', 65432)
```

Now let's run the client and enter a search. Let's see if we can find him:

Shell

```
$ ./app-client.py 10.0.1.1 65432 search morpheus
starting connection to ('10.0.1.1', 65432)
sending b'\x00d{"byteorder": "big", "content-type": "text/json", "content-encoding": "utf-8", "conte
received response {'result': 'Follow the white rabbit. 🐰'} from ('10.0.1.1', 65432)
got result: Follow the white rabbit. 🐰
closing connection to ('10.0.1.1', 65432)
```

My terminal is running a shell that's using a text encoding of Unicode (UTF-8), so the output above prints nicely with emojis.

Let's see if we can find the puppies:

Shell

```
$ ./app-client.py 10.0.1.1 65432 search 🐶
starting connection to ('10.0.1.1', 65432)
sending b'\x00d{"byteorder": "big", "content-type": "text/json", "content-encoding": "utf-8", "conte
received response {'result': '🐶 Playing ball! 🐶'} from ('10.0.1.1', 65432)
got result: 🐶 Playing ball! 🐶
closing connection to ('10.0.1.1', 65432)
```

Notice the byte string sent over the network for the request in the sending line. It's easier to see if you look for the bytes printed in hex that represent the puppy emoji: `\xf0\x9f\x90\xb6`. I was able to [enter the emoji](#) for the search since my terminal is using Unicode with the encoding UTF-8.

This demonstrates that we're sending raw bytes over the network and they need to be decoded by the receiver to be interpreted correctly. This is why we went to all of the trouble to create a header that contains the content type and encoding.

Here's the server output from both client connections above:

Shell

```
accepted connection from ('10.0.2.2', 55340)
received request {'action': 'search', 'value': 'morpheus'} from ('10.0.2.2', 55340)
sending b'\x00g{"byteorder": "little", "content-type": "text/json", "content-encoding": "utf-8", "co
closing connection to ('10.0.2.2', 55340)
```

```
CLOSING CONNECTION TO ('10.0.2.2', 55340)
accepted connection from ('10.0.2.2', 55338)
received request {'action': 'search', 'value': ' '} from ('10.0.2.2', 55338)
sending b'\x00{"byteorder": "little", "content-type": "text/json", "content-encoding": "utf-8", "co
closing connection to ('10.0.2.2', 55338)
```

Look at the sending line to see the bytes that were written to the client's socket. This is the server's response message.

You can also test sending binary requests to the server if the action argument is anything other than search:

Shell

```
$ ./app-client.py 10.0.1.1 65432 binary ⊕
starting connection to ('10.0.1.1', 65432)
sending b'\x00{"byteorder": "big", "content-type": "binary/custom-client-binary-type", "content-enc
received binary/custom-server-binary-type response from ('10.0.1.1', 65432)
got response: b'First 10 bytes of request: binary\xf0\x9f\x98\x83'
closing connection to ('10.0.1.1', 65432)
```

Since the request's content-type is not text/json, the server treats it as a custom binary type and doesn't perform JSON decoding. It simply prints the content-type and returns the first 10 bytes to the client:

Shell

```
$ ./app-server.py 1 65432
listening on ('', 65432)
accepted connection from ('10.0.2.2', 55320)
received binary/custom-client-binary-type request from ('10.0.2.2', 55320)
sending b'\x00\x7f{"byteorder": "little", "content-type": "binary/custom-server-binary-type", "conte
closing connection to ('10.0.2.2', 55320)
```

Troubleshooting

Inevitably, something won't work, and you'll be wondering what to do. Don't worry, it happens to all of us.

Hopefully, with the help of this tutorial, your debugger, and favorite search engine, you'll be able to get going again with the source code part.

If not, your first stop should be Python's [socket module](#) documentation. Make sure you read all of the documentation for each function or method you're calling. Also, read through the [Reference](#) section for ideas. In particular, check the [Errors](#) section.

Sometimes, it's not all about the source code. The source code might be correct, and it's just the other host, the client or server. Or it could be the network, for example, a router, firewall, or some other networking device that's playing man-in-the-middle.

For these types of issues, additional tools are essential. Below are a few tools and utilities that might help or at least provide some clues.

ping

ping will check if a host is alive and connected to the network by sending an [ICMP](#) echo request. It communicates directly with the operating system's TCP/IP protocol stack, so it works independently from any application running on the host.

Below is an example of running ping on macOS:

Shell

```
$ ping -c 3 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.058 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.165 ms
```

```

64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.103 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.164 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.058/0.129/0.165/0.050 ms

```

Note the statistics at the end of the output. This can be helpful when you're trying to discover intermittent connectivity problems. For example, is there any packet loss? How much latency is there (see the round-trip times)?

If there's a firewall between you and the other host, a ping's echo request may not be allowed. Some firewall administrators implement policies that enforce this. The idea being that they don't want their hosts to be discoverable. If this is the case and you have firewall rules added to allow the hosts to communicate, make sure that the rules also allow ICMP to pass between them.

ICMP is the protocol used by ping, but it's also the protocol TCP and other lower-level protocols use to communicate error messages. If you're experiencing strange behavior or slow connections, this could be the reason.

ICMP messages are identified by type and code. To give you an idea of the important information they carry, here are a few:

ICMP Type	ICMP Code	Description
8	0	Echo request
0	0	Echo reply
3	0	Destination network unreachable
3	1	Destination host unreachable
3	2	Destination protocol unreachable
3	3	Destination port unreachable
3	4	Fragmentation required, and DF flag set
11	0	TTL expired in transit

See the article [Path MTU Discovery](#) for information regarding fragmentation and ICMP messages. This is an example of something that can cause strange behavior that I mentioned previously.

netstat

In the section [Viewing Socket State](#), we looked at how netstat can be used to display information about sockets and their current state. This utility is available on macOS, Linux, and Windows.

I didn't mention the columns Recv-Q and Send-Q in the example output. These columns will show you the number of bytes that are held in network buffers that are queued for transmission or receipt, but for some reason haven't been read or written by the remote or local application.

In other words, the bytes are waiting in network buffers in the operating system's queues. One reason could be the application is CPU bound or is otherwise unable to call socket.recv() or socket.send() and process the bytes. Or there could be network issues affecting communications like congestion or failing network hardware or cabling.

To demonstrate this and see how much data I could send before seeing an error, I wrote a test client that connects to a test server and repeatedly calls socket.send(). The test server never calls socket.recv(). It just accepts the connection. This causes the network buffers on the server to fill, which eventually raises an error on the client.

First, I started the server:

[Shell](#)

```
$ ./app-server-test.py 127.0.0.1 65432
listening on ('127.0.0.1', 65432)
```

Then I ran the client. Let's see what the error is:

Shell

```
$ ./app-client-test.py 127.0.0.1 65432 binary test
error: socket.send() blocking io exception for ('127.0.0.1', 65432):
BlockingIOError(35, 'Resource temporarily unavailable')
```

Here's the netstat output while the client and server were still running, with the client printing out the error message above multiple times:

Shell

```
$ netstat -an | grep 65432
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp4   408300      0 127.0.0.1.65432      127.0.0.1.53225      ESTABLISHED
tcp4      0 269868 127.0.0.1.53225      127.0.0.1.65432      ESTABLISHED
tcp4      0      0 127.0.0.1.65432      *.*                  LISTEN
```

The first entry is the server (Local Address has port 65432):

Shell

```
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp4   408300      0 127.0.0.1.65432      127.0.0.1.53225      ESTABLISHED
```

Notice the Recv-Q: 408300.

The second entry is the client (Foreign Address has port 65432):

Shell

```
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp4      0 269868 127.0.0.1.53225      127.0.0.1.65432      ESTABLISHED
```

Notice the Send-Q: 269868.

The client sure was trying to write bytes, but the server wasn't reading them. This caused the server's network buffer queue to fill on the receive side and the client's network buffer queue to fill on the send side.

Windows

If you work with Windows, there's a suite of utilities that you should definitely check out if you haven't already: [Windows Sysinternals](#).

One of them is `TCPView.exe`. `TCPView` is a graphical netstat for Windows. In addition to addresses, port numbers, and socket state, it will show you running totals for the number of packets and bytes, sent and received. Like the Unix utility `lsof`, you also get the process name and ID. Check the menus for other display options.

Wireshark

Sometimes you need to see what's happening on the wire. Forget about what the application log says or what the value is that's being returned from a library call. You want to see what's actually being sent or received on the network. Just like debuggers, when you need to see it, there's no substitute.

[Wireshark](#) is a network protocol analyzer and traffic capture application that runs on macOS, Linux, and Windows, among others. There's a GUI version named `wireshark`, and also a terminal, text-based version named `tshark`.

Running a traffic capture is a great way to watch how an application behaves on the network and gather evidence about what it sends and receives, and how often and how much. You'll also be able to see when a client or server closes or aborts a connection or stops responding. This information can be extremely helpful when you're troubleshooting.

There are many good tutorials and other resources on the web that will walk you through the basics of using Wireshark and TShark.

Here's an example of a traffic capture using Wireshark on the loopback interface:

Here's the same example shown above using tshark:

Shell

```
$ tshark -i lo0 'tcp port 65432'
Capturing on 'Loopback'
  1  0.000000  127.0.0.1 → 127.0.0.1      TCP  68 53942 → 65432 [SYN] Seq=0 Win=65535 Len=16
  2  0.000057  127.0.0.1 → 127.0.0.1      TCP  68 65432 → 53942 [SYN, ACK] Seq=0 Ack=1 Win=65535 L
  3  0.000068  127.0.0.1 → 127.0.0.1      TCP  56 53942 → 65432 [ACK] Seq=1 Ack=1 Win=408288 Len=0
  4  0.000075  127.0.0.1 → 127.0.0.1      TCP  56 [TCP Window Update] 65432 → 53942 [ACK] Seq=1 Ac
  5  0.000216  127.0.0.1 → 127.0.0.1      TCP 202 53942 → 65432 [PSH, ACK] Seq=1 Ack=1 Win=408288
  6  0.000234  127.0.0.1 → 127.0.0.1      TCP  56 65432 → 53942 [ACK] Seq=1 Ack=147 Win=408128 Len
  7  0.000627  127.0.0.1 → 127.0.0.1      TCP 204 65432 → 53942 [PSH, ACK] Seq=1 Ack=147 Win=4081
  8  0.000649  127.0.0.1 → 127.0.0.1      TCP  56 53942 → 65432 [ACK] Seq=147 Ack=149 Win=408128 L
  9  0.000668  127.0.0.1 → 127.0.0.1      TCP  56 65432 → 53942 [FIN, ACK] Seq=149 Ack=147 Win=408
 10 0.000682  127.0.0.1 → 127.0.0.1      TCP  56 53942 → 65432 [ACK] Seq=147 Ack=150 Win=408128 L
 11 0.000687  127.0.0.1 → 127.0.0.1      TCP  56 [TCP Dup ACK 6#1] 65432 → 53942 [ACK] Seq=150 Ac
 12 0.000848  127.0.0.1 → 127.0.0.1      TCP  56 53942 → 65432 [FIN, ACK] Seq=147 Ack=150 Win=408
 13 0.001004  127.0.0.1 → 127.0.0.1      TCP  56 65432 → 53942 [ACK] Seq=150 Ack=148 Win=408128 L
  0 packets captured
```

Reference

This section serves as a general reference with additional information and links to external resources.

Python Documentation

- Python's [socket module](#)
- Python's [Socket Programming HOWTO](#)

Errors

The following is from Python's socket module documentation:

"All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise `OSError` or one of its subclasses." ([Source](#))

Here are some common errors you'll probably encounter when working with sockets:

Exception	errno Constant	Description
BlockingIOError	EWOULDBLOCK	Resource temporarily unavailable. For example, in non-blocking mode when calling <code>send()</code> and the peer is busy and not reading, the send queue (network buffer) is full. Or there are issues with the network. Hopefully this is a temporary condition.
OSError	EADDRINUSE	Address already in use. Make sure there's not another process running that's using the same port number and your server is setting the socket option <code>SO_REUSEADDR</code> : <code>socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,</code>
ConnectionResetError	ECONNRESET	Connection reset by peer. The remote process crashed or did not close its socket properly (unclean shutdown). Or there's a firewall or other device in the network path that's missing rules or misbehaving.
TimeoutError	ETIMEDOUT	Operation timed out. No response from peer.
ConnectionRefusedError	ECONNREFUSED	Connection refused. No application listening on specified port.

Socket Address Families

`socket.AF_INET` and `socket.AF_INET6` represent the address and protocol families used for the first argument to `socket.socket()`. APIs that use an address expect it to be in a certain format, depending on whether the socket was created with `socket.AF_INET` or `socket.AF_INET6`.

Address Family	Protocol	Address Tuple	Description
<code>socket.AF_INET</code>	IPv4	(host, port)	host is a string with a hostname like ' <code>www.example.com</code> ' or an IPv4 address like ' <code>10.1.2.3</code> '. port is an integer.
<code>socket.AF_INET6</code>	IPv6	(host, port, flowinfo, scopeid)	host is a string with a hostname like ' <code>www.example.com</code> ' or an IPv6 address like ' <code>fe80::6203:7ab:fe88:9c23</code> '. port is an integer. flowinfo and

Address Family	Protocol	Address Tuple	scope_id represent the sin6_flowinfo Description and sin6_scope_id members in the C struct sockaddr_in6.
----------------	----------	---------------	---

Note the excerpt below from Python's socket module documentation regarding the host value of the address tuple:

"For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents INADDR_ANY, and the string '<broadcast>' represents INADDR_BROADCAST. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs." ([Source](#))

See Python's [Socket families documentation](#) for more information.

I've used IPv4 sockets in this tutorial, but if your network supports it, try testing and using IPv6 if possible. One way to support this easily is by using the function [socket.getaddrinfo\(\)](#). It translates the host and port arguments into a sequence of 5-tuples that contains all of the necessary arguments for creating a socket connected to that service. `socket.getaddrinfo()` will understand and interpret passed-in IPv6 addresses and hostnames that resolve to IPv6 addresses, in addition to IPv4.

The following example returns address information for a TCP connection to example.org on port 80:

```
Python >>>
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '',
 6, ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '',
 6, ('93.184.216.34', 80))]
```

Results may differ on your system if IPv6 isn't enabled. The values returned above can be used by passing them to `socket.socket()` and `socket.connect()`. There's a client and server example in the [Example section](#) of Python's socket module documentation.

Using Hostnames

For context, this section applies mostly to using hostnames with `bind()` and `connect()`, or `connect_ex()`, when you intend to use the loopback interface, "localhost." However, it applies any time you're using a hostname and there's an expectation of it resolving to a certain address and having a special meaning to your application that affects its behavior or assumptions. This is in contrast to the typical scenario of a client using a hostname to connect to a server that's resolved by DNS, like `www.example.com`.

The following is from Python's socket module documentation:

"If you use a hostname in the host portion of IPv4/v6 socket address, the program may show a non-deterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion." ([Source](#))

The standard convention for the name "`localhost`" is for it to resolve to `127.0.0.1` or `::1`, the loopback interface. This will more than likely be the case for you on your system, but maybe not. It depends on how your system is configured for name resolution. As with all things IT, there are always exceptions, and there are no guarantees that using the name "localhost" will connect to the loopback interface.

For example, on Linux, see `man nsswitch.conf`, the Name Service Switch configuration file. Another place to check on macOS and Linux is the file `/etc/hosts`. On Windows, see `C:\Windows\System32\drivers\etc\hosts`. The `hosts` file contains a static table of name to address mappings in a simple text format. [DNS](#) is another piece of the puzzle altogether.

Interestingly enough, as of this writing (June 2018), there's an RFC draft [Let 'localhost' be localhost](#) that discusses the conventions, assumptions and security around using the name "localhost."

What's important to understand is that when you use hostnames in your application, the returned address(es) could

literally be anything. Don't make assumptions regarding a name if you have a security-sensitive application. Depending on your application and environment, this may or may not be a concern for you.

Note: Security precautions and best practices still apply, even if your application isn't "security-sensitive." If your application accesses the network, it should be secured and maintained. This means, at a minimum:

- System software updates and security patches are applied regularly, including Python. Are you using any third party libraries? If so, make sure those are checked and updated too.
- If possible, use a dedicated or host-based firewall to restrict connections to trusted systems only.
- What DNS servers are configured? Do you trust them and their administrators?
- Make sure that request data is sanitized and validated as much as possible prior to calling other code that processes it. Use (fuzz) tests for this and run them regularly.

Regardless of whether or not you're using hostnames, if your application needs to support secure connections (encryption and authentication), you'll probably want to look into using [TLS](#). This is its own separate topic and beyond the scope of this tutorial. See Python's [ssl module documentation](#) to get started. This is the same protocol that your web browser uses to connect securely to web sites.

With interfaces, IP addresses, and name resolution to consider, there are many variables. What should you do? Here are some recommendations that you can use if you don't have a network application review process:

Application	Usage	Recommendation
Server	loopback interface	Use an IP address, for example, 127.0.0.1 or ::1.
Server	ethernet interface	Use an IP address, for example, 10.1.2.3. To support more than one interface, use an empty string for all interfaces/addresses. See the security note above.
Client	loopback interface	Use an IP address, for example, 127.0.0.1 or ::1.
Client	ethernet interface	Use an IP address for consistency and non-reliance on name resolution. For the typical case, use a hostname. See the security note above.

For clients or servers, if you need to authenticate the host you're connecting to, look into using TLS.

Blocking Calls

A socket function or method that temporarily suspends your application is a blocking call. For example, `accept()`, `connect()`, `send()`, and `recv()` "block." They don't return immediately. Blocking calls have to wait on system calls (I/O) to complete before they can return a value. So you, the caller, are blocked until they're done or a timeout or other error occurs.

Blocking socket calls can be set to non-blocking mode so they return immediately. If you do this, you'll need to at least refactor or redesign your application to handle the socket operation when it's ready.

Since the call returns immediately, data may not be ready. The callee is waiting on the network and hasn't had time to complete its work. If this is the case, the current status is the `errno` value `socket.EWOULDBLOCK`. Non-blocking mode is supported with [setblocking\(\)](#).

By default, sockets are always created in blocking mode. See [Notes on socket timeouts](#) for a description of the three modes.

Closing Connections

An interesting thing to note with TCP is it's completely legal for the client or server to close their side of the connection while the other side remains open. This is referred to as a "half-open" connection. It's the application's decision whether or not this is desirable. In general, it's not. In this state, the side that's closed their end of the connection can no longer send data. They can only receive it.

I'm not advocating that you take this approach, but as an example, HTTP uses a header named "Connection" that's used to standardize how applications should close or persist open connections. For details, see [section 6.3 in RFC 7230, Hypertext Transfer Protocol \(HTTP/1.1\): Message Syntax and Routing](#).

When designing and writing your application and its application-layer protocol, it's a good idea to go ahead and work out how you expect connections to be closed. Sometimes this is obvious and simple, or it's something that can take some initial prototyping and testing. It depends on the application and how the message loop is processed with its expected data. Just make sure that sockets are always closed in a timely manner after they complete their work.

Byte Endianness

See [Wikipedia's article on endianness](#) for details on how different CPUs store byte orderings in memory. When interpreting individual bytes, this isn't a problem. However, when handling multiple bytes that are read and processed as a single value, for example a 4-byte integer, the byte order needs to be reversed if you're communicating with a machine that uses a different endianness.

Byte order is also important for text strings that are represented as multi-byte sequences, like Unicode. Unless you're always using "true," strict [ASCII](#) and control the client and server implementations, you're probably better off using Unicode with an encoding like UTF-8 or one that supports a [byte order mark \(BOM\)](#).

It's important to explicitly define the encoding used in your application-layer protocol. You can do this by mandating that all text is UTF-8 or using a "content-encoding" header that specifies the encoding. This prevents your application from having to detect the encoding, which you should avoid if possible.

This becomes problematic when there is data involved that's stored in files or a database and there's no metadata available that specifies its encoding. When the data is transferred to another endpoint, it will have to try to detect the encoding. For a discussion, see [Wikipedia's Unicode article](#) that references [RFC 3629: UTF-8, a transformation format of ISO 10646](#):

"However RFC 3629, the UTF-8 standard, recommends that byte order marks be forbidden in protocols using UTF-8, but discusses the cases where this may not be possible. In addition, the large restriction on possible patterns in UTF-8 (for instance there cannot be any lone bytes with the high bit set) means that it should be possible to distinguish UTF-8 from other character encodings without relying on the BOM." ([Source](#))

The takeaway from this is to always store the encoding used for data that's handled by your application if it can vary. In other words, try to somehow store the encoding as metadata if it's not always UTF-8 or some other encoding with a BOM. Then you can send that encoding in a header along with the data to tell the receiver what it is.

The byte ordering used in TCP/IP is [big-endian](#) and is referred to as network order. Network order is used to represent integers in lower layers of the protocol stack, like IP addresses and port numbers. Python's socket module includes functions that convert integers to and from network and host byte order:

Function	Description
<code>socket.ntohl(x)</code>	Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.
<code>socket ntohs(x)</code>	Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.
<code>socket htonl(x)</code>	Convert 32-bit positive integers from host to network byte order. On machines where the

Function	Description
socket.htons(x)	Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

You can also use the [struct module](#) to pack and unpack binary data using format strings:

Python

```
import struct
network_bytorder_int = struct.pack('>H', 256)
python_int = struct.unpack('>H', network_bytorder_int)[0]
```

Conclusion

We covered a lot of ground in this tutorial. Networking and sockets are large subjects. If you're new to networking or sockets, don't be discouraged by all of the terms and acronyms.

There are a lot of pieces to become familiar with in order to understand how everything works together. However, just like Python, it will start to make more sense as you get to know the individual pieces and spend more time with them.

We looked at the low-level socket API in Python's socket module and saw how it can be used to create client-server applications. We also created our own custom class and used it as an application-layer protocol to exchange messages and data between endpoints. You can use this class and build upon it to learn and help make creating your own socket applications easier and faster.

You can find the [source code on GitHub](#).

Congratulations on making it to the end! You are now well on your way to using sockets in your own applications.

I hope this tutorial has given you the information, examples, and inspiration needed to start you on your sockets development journey.

About Nathan Jennings

Nathan is a member of the Real Python tutorial team who started his programmer career with C long time ago, but eventually found Python. From web applications and data collection to networking and network security, he enjoys all things Pythonic.

[» More about Nathan](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

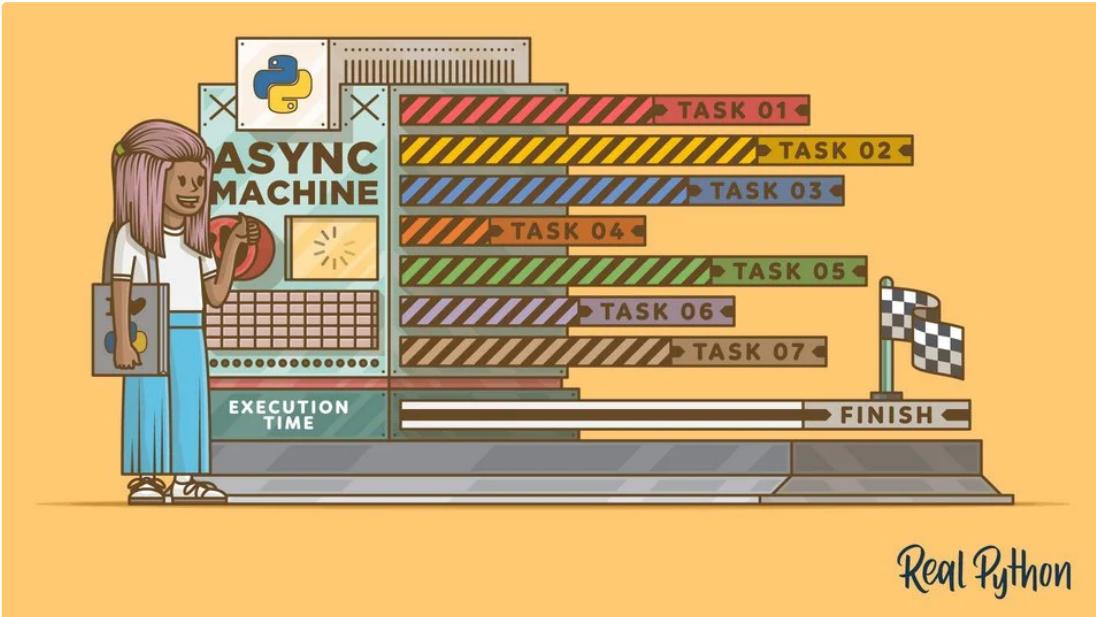
[Brad](#)

[Jim](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [advanced](#) [python](#) [web-dev](#)



Real Python

Getting Started With Async Features in Python

by [Doug Farrell](#) ⌂ Sep 23, 2019 🗣 8 Comments 🎧 [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Understanding Asynchronous Programming](#)
 - [Building a Synchronous Web Server](#)
 - [Thinking Differently About Programming](#)
- [Programming Parents: Not as Easy as It Looks!](#)
 - [Thought Experiment #1: The Synchronous Parent](#)
 - [Thought Experiment #2: The Polling Parent](#)
 - [Thought Experiment #3: The Threading Parent](#)
- [Using Python Async Features in Practice](#)
 - [Synchronous Programming](#)
 - [Simple Cooperative Concurrency](#)
 - [Cooperative Concurrency With Blocking Calls](#)
 - [Cooperative Concurrency With Non-Blocking Calls](#)
 - [Synchronous \(Blocking\) HTTP Calls](#)
 - [Asynchronous \(Non-Blocking\) HTTP Calls](#)
- [Conclusion](#)



Have you heard of asynchronous programming in Python? Are you curious to know more about Python async features and how you can use them in your work? Perhaps you've even tried to write threaded programs and run into some issues. If you're looking to understand how to use Python async features, then you've come to the right place.

In this article, you'll learn:

- What a **synchronous program** is
- What an **asynchronous program** is
- Why you might want to write an asynchronous program
- How to use Python async features

All of the example code in this article have been tested with Python 3.7.2. You can grab a copy to follow along by clicking the link below:

Download Code: [Click here to download the code you'll use](#) to learn about async features in Python in this tutorial.

[Remove ads](#)

Understanding Asynchronous Programming

A **synchronous program** is executed one step at a time. Even with conditional branching, loops and function calls, you can still think about the code in terms of taking one execution step at a time. When each step is complete, the program moves on to the next one.

Here are two examples of programs that work this way:

- **Batch processing programs** are often created as synchronous programs. You get some input, process it, and create some output. Steps follow one after the other until the program reaches the desired output. The program only needs to pay attention to the steps and their order.
- **Command-line programs** are small, quick processes that run in a terminal. These scripts are used to create something, transform one thing into something else, generate a report, or perhaps list out some data. This can be expressed as a series of program steps that are executed sequentially until the program is done.

An **asynchronous program** behaves differently. It still takes one execution step at a time. The difference is that the system may not wait for an execution step to be completed before moving on to the next one.

This means that the program will move on to future execution steps even though a previous step hasn't yet finished and is still running elsewhere. This also means that the program knows what to do when a previous step does finish running.

Why would you want to write a program in this manner? The rest of this article will help you answer that question and give you the tools you need to elegantly solve interesting asynchronous problems.

Building a Synchronous Web Server

A web server's basic unit of work is, more or less, the same as batch processing. The server will get some input, process it, and create the output. Written as a synchronous program, this would create a working web server.

It would also be an *absolutely terrible* web server.

Why? In this case, one unit of work (input, process, output) is not the only purpose. The real purpose is to handle hundreds or even thousands of units of work as quickly as possible. This can happen over long periods of time, and several work units may even arrive all at once.

Can a synchronous web server be made better? Sure, you could optimize the execution steps so that all the work coming in is handled as quickly as possible. Unfortunately, there are limitations to this approach. The result could be a web server that doesn't respond fast enough, can't handle enough work, or even one that times out when work gets stacked up.

NOTE: There are other limitations you might see if you tried to optimize the above approach. These include network speed, file IO speed, database query speed, and the speed of other connected services, to name a few. What these all have in common is that they are all IO functions. All of these items are orders of magnitude slower than the CPU's processing speed.

In a synchronous program, if an execution step starts a database query, then the CPU is essentially idle until the database query is returned. For batch-oriented programs, this isn't a priority most of the time. Processing the results of that IO operation is the goal. Often, this can take longer than the IO operation itself. Any optimization efforts would be focused on the processing work, not the IO.

Asynchronous programming techniques allow your programs to take advantage of relatively slow IO processes by freeing the CPU to do other work.

Thinking Differently About Programming

When you start trying to understand asynchronous programming, you might see a lot of discussion about the importance of blocking, or writing [non-blocking code](#). (Personally, I struggled to get a good grasp of these concepts from the people I asked and the documentation I read.)

What is non-blocking code? What's blocking code, for that matter? Would the answers to these questions help you write a better web server? If so, how could you do it? Let's find out!

Writing asynchronous programs requires that you think differently about programming. While this new way of thinking can be hard to wrap your head around, it's also an interesting exercise. That's because the real world is almost entirely asynchronous, and so is how you interact with it.

Imagine this: you're a parent trying to do several things at once. You have to balance the checkbook, do the laundry, and keep an eye on the kids. Somehow, you're able to do all of these things at the same time without even thinking about it! Let's break it down:

- Balancing the checkbook is a **synchronous** task. One step follows another until it's done. You're doing all the work yourself.
- However, you can break away from the checkbook to do laundry. You unload the dryer, move clothes from the washer to the dryer, and start another load in the washer.
- Working with the washer and dryer is a synchronous task, but the bulk of the work happens *after* the washer and dryer are started. Once you've got them going, you can walk away and get back to the checkbook task. At this point, the washer and dryer tasks have become **asynchronous**. The washer and dryer will run independently until the buzzer goes off (notifying you that the task needs attention).
- Watching your kids is another asynchronous task. Once they are set up and playing, they can do so independently for the most part. This changes when someone needs attention, like when someone gets hungry or hurt. When one of your kids yells in alarm, you react. The kids are a long-running task with high priority. Watching them supersedes any other tasks you might be doing, like the checkbook or laundry.

These examples can help to illustrate the concepts of blocking and non-blocking code. Let's think about this in programming terms. In this example, you're like the CPU. While you're moving the laundry around, you (the CPU) are busy and blocked from doing other work, like balancing the checkbook. But that's okay because the task is relatively quick.

On the other hand, starting the washer and dryer does not block you from performing other tasks. It's an asynchronous function because you don't have to wait for it to finish. Once it's started, you can go back to something else. This is called a context switch: the context of what you're doing has changed, and the machine's buzzer will notify you sometime in the future when the laundry task is complete.

As a human, this is how you work all the time. You naturally juggle multiple things at once, often without thinking about it. As a developer, the trick is how to translate this kind of behavior into code that does the same kind of thing.

Programming Parents: Not as Easy as It Looks!

If you recognize yourself (or your parents) in the example above, then that's great! You've got a leg up in understanding asynchronous programming. Again, you're able to switch contexts between competing tasks fairly easily, picking up some tasks and resuming others. Now you're going to try and program this behavior into virtual parents!

Thought Experiment #1: The Synchronous Parent

How would you create a parent program to do the above tasks in a completely synchronous manner? Since watching the kids is a high-priority task, perhaps your program would do just that. The parent watches over the kids while waiting for something to happen that might need their attention. However, nothing else (like the checkbook or laundry) would get done in this scenario.

Now, you can re-prioritize the tasks any way you want, but only one of them would happen at any given time. This is the result of a synchronous, step-by-step approach. Like the synchronous web server described above, this would work, but it might not be the best way to live. The parent wouldn't be able to complete any other tasks until the kids fell asleep. All other tasks would happen afterward, well into the night. (A couple of weeks of this and many real parents might jump out the window!)

Thought Experiment #2: The Polling Parent

If you used [polling](#), then you could change things up so that multiple tasks are completed. In this approach, the parent would periodically break away from the current task and check to see if any other tasks need attention.

Let's make the polling interval something like fifteen minutes. Now, every fifteen minutes your parent checks to see if the washer, dryer or kids need any attention. If not, then the parent can go back to work on the checkbook. However, if any of those tasks *do* need attention, then the parent will take care of it before going back to the checkbook. This cycle continues on until the next timeout out of the polling loop.

This approach works as well since multiple tasks are getting attention. However, there are a couple of problems:

1. **The parent may spend a lot of time checking on things that don't need attention:** The washer and dryer haven't yet finished, and the kids don't need any attention unless something unexpected happens.
2. **The parent may miss completed tasks that do need attention:** For instance, if the washer finished its cycle at the beginning of the polling interval, then it wouldn't get any attention for up to fifteen minutes! What's more, watching the kids is supposedly the highest priority task. They couldn't tolerate fifteen minutes with no attention when something might be going drastically wrong.

You could address these issues by shortening the polling interval, but now your parent (the CPU) would be spending more time context switching between tasks. This is when you start to hit a point of diminishing returns. (Once again, a couple of weeks living like this and, well... See the previous comment about windows and jumping.)

Thought Experiment #3: The Threading Parent

"If I could only clone myself..." If you're a parent, then you've probably had similar thoughts! Since you're programming virtual parents, you can essentially do this by using threading. This is a mechanism that allows multiple sections of one program to run at the same time. Each section of code that runs independently is known as a thread, and all threads share the same memory space.

If you think of each task as a part of one program, then you can separate them and run them as threads. In other words, you can "clone" the parent, creating one instance for each task: watching the kids, monitoring the washer,

monitoring the dryer, and balancing the checkbook. All of these “clones” are running independently.

This sounds like a pretty nice solution, but there are some issues here as well. One is that you’ll have to explicitly tell each parent instance what to do in your program. This can lead to some problems since all instances share everything in the program space.

For example, say that Parent A is monitoring the dryer. Parent A sees that the clothes are dry, so they take control of the dryer and begin unloading the clothes. At the same time, Parent B sees that the washer is done, so they take control of the washer and begin removing clothes. However, Parent B also needs to take control of the dryer so they can put the wet clothes inside. This can’t happen, because Parent A currently has control of the dryer.

After a short while, Parent A has finished unloading clothes. Now they want to take control of the washer and start moving clothes into the empty dryer. This can’t happen, either, because Parent B currently has control of the washer!

These two parents are now deadlocked. Both have control of their own resource *and* want control of the other resource. They’ll wait forever for the other parent instance to release control. As the programmer, you’d have to write code to work this situation out.

Note: Threaded programs allow you to create multiple, parallel paths of execution that all share the same memory space. This is both an advantage and a disadvantage. Any memory shared between threads is subject to one or more threads trying to use the same shared memory at the same time. This can lead to data corruption, data read in an invalid state, and data that’s just messy in general.

In threaded programming, the context switch happens under system control, not the programmer. The system controls when to switch contexts and when to give threads access to shared data, thereby changing the context of how the memory is being used. All of these kinds of problems are manageable in threaded code, but it’s difficult to get right, and hard to debug when it’s wrong.

Here’s another issue that might arise from threading. Suppose that a child gets hurt and needs to be taken to urgent care. Parent C has been assigned the task of watching over the kids, so they take the child right away. At the urgent care, Parent C needs to write a fairly large check to cover the cost of seeing the doctor.

Meanwhile, Parent D is at home working on the checkbook. They’re unaware of this large check being written, so they’re very surprised when the family checking account is suddenly overdrawn!

Remember, these two parent instances are working within the same program. The family checking account is a shared resource, so you’d have to work out a way for the child-watching parent to inform the checkbook-balancing parent. Otherwise, you’d need to provide some kind of locking mechanism so that the checkbook resource can only be used by one parent at a time, with updates.

 [Remove ads](#)

Using Python Async Features in Practice

Now you’re going to take some of the approaches outlined in the thought experiments above and turn them into functioning Python programs.

All of the examples in this article have been tested with Python 3.7.2. The requirements.txt file indicates which modules you'll need to install to run all the examples. If you haven't yet downloaded the file, you can do so now:

Download Code: [Click here to download the code you'll use](#) to learn about async features in Python in this tutorial.

You also might want to set up a [Python virtual environment](#) to run the code so you don't interfere with your system Python.

Synchronous Programming

This first example shows a somewhat contrived way of having a task retrieve work from a queue and process that work. A queue in Python is a nice [FIFO](#) (first in first out) data structure. It provides methods to put things in a queue and take them out again in the order they were inserted.

In this case, the work is to get a number from the queue and have a loop count up to that number. It prints to the console when the loop begins, and again to output the total. This program demonstrates one way for multiple synchronous tasks to process the work in a queue.

The program named example_1.py in the repository is listed in full below:

```
Python
1 import queue
2
3 def task(name, work_queue):
4     if work_queue.empty():
5         print(f"Task {name} nothing to do")
6     else:
7         while not work_queue.empty():
8             count = work_queue.get()
9             total = 0
10            print(f"Task {name} running")
11            for x in range(count):
12                total += 1
13            print(f"Task {name} total: {total}")
14
15 def main():
16     """
17     This is the main entry point for the program
18     """
19     # Create the queue of work
20     work_queue = queue.Queue()
21
22     # Put some work in the queue
23     for work in [15, 10, 5, 2]:
```

```

24     work_queue.put(work)
25
26     # Create some synchronous tasks
27     tasks = [(task, "One", work_queue), (task, "Two", work_queue)]
28
29     # Run the tasks
30     for t, n, q in tasks:
31         t(n, q)
32
33 if __name__ == "__main__":
34     main()

```

Let's take a look at what each line does:

- **Line 1** imports the queue module. This is where the program stores work to be done by the tasks.
- **Lines 3 to 13** define task(). This function pulls work out of work_queue and processes the work until there isn't any more to do.
- **Line 15** defines main() to run the program tasks.
- **Line 20** creates the work_queue. All tasks use this shared resource to retrieve work.
- **Lines 23 to 24** put work in work_queue. In this case, it's just a random count of values for the tasks to process.
- **Line 27** creates a list of task tuples, with the parameter values those tasks will be passed.
- **Lines 30 to 31** iterate over the list of task tuples, calling each one and passing the previously defined parameter values.
- **Line 34** calls main() to run the program.

The task in this program is just a function accepting a string and a queue as parameters. When executed, it looks for anything in the queue to process. If there is work to do, then it pulls values off the queue, starts a [for loop](#) to count up to that value, and outputs the total at the end. It continues getting work off the queue until there is nothing left and it exits.

When this program is run, it produces the output you see below:

Shell

```

Task One running
Task One total: 15
Task One running
Task One total: 10
Task One running
Task One total: 5
Task One running
Task One total: 2
Task Two nothing to do

```

This shows that Task One does all the work. The [while loop](#) that Task One hits within task() consumes all the work on the queue and processes it. When that loop exits, Task Two gets a chance to run. However, it finds that the queue is empty, so Task Two prints a statement that says it has nothing to do and then exits. There's nothing in the code to allow both Task One and Task Two to switch contexts and work together.

Simple Cooperative Concurrency

The next version of the program allows the two tasks to work together. Adding a `yield` statement means the loop will yield control at the specified point while still maintaining its context. This way, the yielding task can be restarted later.

The `yield` statement turns `task()` into a [generator](#). A generator function is called just like any other function in Python, but when the `yield` statement is executed, control is returned to the caller of the function. This is

essentially a context switch, as control moves from the generator function to the caller.

The interesting part is that control can be given *back* to the generator function by calling `next()` on the generator. This is a context switch back to the generator function, which picks up execution with all function variables that were defined before the `yield` still intact.

The `while` loop in `main()` takes advantage of this when it calls `next(t)`. This statement restarts the task at the point where it previously yielded. All of this means that you're in control when the context switch happens: when the `yield` statement is executed in `task()`.

This is a form of cooperative multitasking. The program is yielding control of its current context so that something else can run. In this case, it allows the `while` loop in `main()` to run two instances of `task()` as a generator function. Each instance consumes work from the same queue. This is sort of clever, but it's also a lot of work to get the same results as the first program. The program `example_2.py` demonstrates this simple concurrency and is listed below:

Python

```
1 import queue
2
3 def task(name, queue):
4     while not queue.empty():
5         count = queue.get()
6         total = 0
7         print(f"Task {name} running")
8         for x in range(count):
9             total += 1
10            yield
11        print(f"Task {name} total: {total}")
12
13 def main():
14     """
15     This is the main entry point for the program
16     """
17     # Create the queue of work
18     work_queue = queue.Queue()
19
20     # Put some work in the queue
21     for work in [15, 10, 5, 2]:
22         work_queue.put(work)
23
24     # Create some tasks
25     tasks = [task("One", work_queue), task("Two", work_queue)]
26
27     # Run the tasks
28     done = False
29     while not done:
30         for t in tasks:
31             try:
32                 next(t)
```

```

33         except StopIteration:
34             tasks.remove(t)
35             if len(tasks) == 0:
36                 done = True
37
38     if __name__ == "__main__":
39         main()

```

Here's what's happening in the code above:

- **Lines 3 to 11** define task() as before, but the addition of yield on Line 10 turns the function into a generator. This where the context switch is made and control is handed back to the while loop in main().
- **Line 25** creates the task list, but in a slightly different manner than you saw in the previous example code. In this case, each task is called with its parameters as its entered in the tasks list variable. This is necessary to get the task() generator function running the first time.
- **Lines 31 to 36** are the modifications to the while loop in main() that allow task() to run cooperatively. This is where control returns to each instance of task() when it yields, allowing the loop to continue and run another task.
- **Line 32** gives control back to task(), and continues its execution after the point where yield was called.
- **Line 36** sets the done variable. The while loop ends when all tasks have been completed and removed from tasks.

This is the output produced when you run this program:

Shell

```

Task One running
Task Two running
Task Two total: 10
Task Two running
Task One total: 15
Task One running
Task Two total: 5
Task One total: 2

```

You can see that both Task One and Task Two are running and consuming work from the queue. This is what's intended, as both tasks are processing work, and each is responsible for two items in the queue. This is interesting, but again, it takes quite a bit of work to achieve these results.

The trick here is using the yield statement, which turns task() into a generator and performs a context switch. The program uses this context switch to give control to the while loop in main(), allowing two instances of a task to run cooperatively.

Notice how Task Two outputs its total first. This might lead you to think that the tasks are running asynchronously. However, this is still a synchronous program. It's structured so the two tasks can trade contexts back and forth. The reason why Task Two outputs its total first is that it's only counting to 10, while Task One is counting to 15. Task Two simply arrives at its total first, so it gets to print its output to the console before Task One.

Note: All of the example code that follows from this point use a module called [codetiming](#) to time and output how long sections of code took to execute. There is a great article [here](#) on RealPython that goes into depth about the codetiming module and how to use it.

This module is part of the Python Package Index and is built by [Geir Arne Hjelle](#), who is part of the *Real Python* team. Geir Arne has been a great help to me reviewing and suggesting things for this article. If you are writing code that needs to include timing functionality, Geir Arne's codetiming module is well worth looking at.

To make the codetiming module available for the examples that follow you'll need to install it. This can be

To make the codetiming module available for the examples that follow you'll need to install it. This can be done with pip with this command: `pip install codetiming`, or with this command: `pip install -r requirements.txt`. The requirements.txt file is part of the example code repository.

[Remove ads](#)

Cooperative Concurrency With Blocking Calls

The next version of the program is the same as the last, except for the addition of a `time.sleep(delay)` in the body of your task loop. This adds a delay based on the value retrieved from the work queue to every iteration of the task loop. The delay simulates the effect of a blocking call occurring in your task.

A blocking call is code that stops the CPU from doing anything else for some period of time. In the thought experiments above, if a parent wasn't able to break away from balancing the checkbook until it was complete, that would be a blocking call.

`time.sleep(delay)` does the same thing in this example, because the CPU can't do anything else but wait for the delay to expire.

Python

```
1 import time
2 import queue
3 from codetiming import Timer
4
5 def task(name, queue):
6     timer = Timer(text=f"Task {name} elapsed time: {:.1f}")
7     while not queue.empty():
8         delay = queue.get()
9         print(f"Task {name} running")
10        timer.start()
11        time.sleep(delay)
12        timer.stop()
13        yield
14
15 def main():
16     """
17     This is the main entry point for the program
18     """
19     # Create the queue of work
20     work_queue = queue.Queue()
21
22     # Put some work in the queue
23     for work in [15, 10, 5, 2]:
24         work_queue.put(work)
25
26     tasks = [task("One", work_queue), task("Two", work_queue)]
27
28     # Run the tasks
29     done = False
30     with Timer(text="\nTotal elapsed time: {:.1f}"):
31         while not done:
32             for t in tasks:
33                 try:
34                     next(t)
35                 except StopIteration:
36                     tasks.remove(t)
37                     if len(tasks) == 0:
38                         done = True
39
40 if __name__ == "__main__":
41     main()
```

Here's what's different in the code above:

- **Line 1** imports the [time module](#) to give the program access to `time.sleep()`.
- **Line 3** imports the the `Timer` code from the `cotiming` module.
- **Line 6** creates the `Timer` instance used to measure the time taken for each iteration of the task loop.
- **Line 10** starts the `timer` instance
- **Line 11** changes `task()` to include a `time.sleep(delay)` to mimic an IO delay. This replaces the `for` loop that did the counting in `example_1.py`.
- **Line 12** stops the `timer` instance and outputs the elapsed time since `timer.start()` was called.
- **Line 30** creates a `Timer` context manager that will output the elapsed time the entire while loop took to execute.

When you run this program, you'll see the following output:

Shell

```
Task One running
Task One elapsed time: 15.0
Task Two running
Task Two elapsed time: 10.0
Task One running
Task One elapsed time: 5.0
Task Two running
Task Two elapsed time: 2.0

Total elapsed time: 32.0
```

As before, both `Task One` and `Task Two` are running, consuming work from the queue and processing it. However, even with the addition of the delay, you can see that cooperative concurrency hasn't gotten you anything. The delay stops the processing of the entire program, and the CPU just waits for the IO delay to be over.

This is exactly what's meant by blocking code in Python `async` documentation. You'll notice that the time it takes to run the entire program is just the cumulative time of all the delays. Running tasks this way is not a win.

Cooperative Concurrency With Non-Blocking Calls

The next version of the program has been modified quite a bit. It makes use of Python `async` features using [`asyncio/await`](#) provided in Python 3.

The `time` and `queue` modules have been replaced with the `asyncio` package. This gives your program access to asynchronous friendly (non-blocking) sleep and queue functionality. The change to `task()` defines it as asynchronous with the addition of the `async` prefix on line 4. This indicates to Python that the function will be asynchronous.

The other big change is removing the `time.sleep(delay)` and `yield` statements, and replacing them with `await asyncio.sleep(delay)`. This creates a non-blocking delay that will perform a context switch back to the caller `main()`.

The `while` loop inside `main()` no longer exists. Instead of `task_array`, there's a call to `await asyncio.gather(...)`. This tells `asyncio` two things:

1. Create two tasks based on `task()` and start running them.
2. Wait for both of these to be completed before moving forward.

The last line of the program `asyncio.run(main())` runs `main()`. This creates what's known as an [event loop](#), it's this loop that will run `main()`, which in turn will run the two instances of `task()`.

The event loop is at the heart of the Python async system. It runs all the code, including `main()`. When task code is executing, the CPU is busy doing work. When the `await` keyword is reached, a context switch occurs, and control passes back to the event loop. The event loop looks at all the tasks waiting for an event (in this case, an `asyncio.sleep(delay)`) and passes control to a task with an event that's ready.

`await asyncio.sleep(delay)` is non-blocking in regards to the CPU. Instead of waiting for the delay to timeout, the CPU registers a sleep event on the event loop task queue and performs a context switch by passing control to the event loop. The event loop continuously looks for completed events and passes control back to the task waiting for that event. In this way, the CPU can stay busy if work is available, while the event loop monitors the events that will happen in the future.

Note: An asynchronous program runs in a single thread of execution. The context switch from one section of code to another that would affect data is completely in your control. This means you can atomize and complete all shared memory data access before making a context switch. This simplifies the shared memory problem inherent in threaded code.

The `example_4.py` code is listed below:

Python

```
1 import asyncio
2 from codetiming import Timer
3
4 async def task(name, work_queue):
5     timer = Timer(text=f"Task {name} elapsed time: {:.1f}"))
6     while not work_queue.empty():
7         delay = await work_queue.get()
8         print(f"Task {name} running")
9         timer.start()
10        await asyncio.sleep(delay)
11        timer.stop()
12
13 async def main():
14     """
15     This is the main entry point for the program
16     """
17     # Create the queue of work
18     work_queue = asyncio.Queue()
19
20     # Put some work in the queue
21     for work in [15, 10, 5, 2]:
22         await work_queue.put(work)
23
24     # Run the tasks
25     with Timer(text="\nTotal elapsed time: {:.1f}"):
26         await asyncio.gather(
27             asyncio.create_task(task("One", work_queue)),
28             asyncio.create_task(task("Two", work_queue)),
29         )
30
31 if __name__ == "__main__":
32     asyncio.run(main())
```

Here's what's different between this program and `example_3.py`:

- **Line 1** imports `asyncio` to gain access to Python `async` functionality. This replaces the `time` import.
- **Line 2** imports the `Timer` code from the `codetiming` module.
- **Line 4** shows the addition of the `async` keyword in front of the `task()` definition. This informs the program that task can run asynchronously.
- **Line 5** creates the `Timer` instance used to measure the time taken for each iteration of the task loop.
- **Line 9** starts the `timer` instance
- **Line 10** replaces `time.sleep(delay)` with the non-blocking `asyncio.sleep(delay)`, which also yields control (or switches contexts) back to the main event loop.
- **Line 11** stops the `timer` instance and outputs the elapsed time since `timer.start()` was called.

- **Line 18** creates the non-blocking asynchronous work_queue.
- **Lines 21 to 22** put work into work_queue in an asynchronous manner using the await keyword.
- **Line 25** creates a Timer context manager that will output the elapsed time the entire while loop took to execute.
- **Lines 26 to 29** create the two tasks and gather them together, so the program will wait for both tasks to complete.
- **Line 32** starts the program running asynchronously. It also starts the internal event loop.

When you look at the output of this program, notice how both Task One and Task Two start at the same time, then wait at the mock IO call:

Shell

```
Task One running
Task Two running
Task Two total elapsed time: 10.0
Task Two running
Task One total elapsed time: 15.0
Task One running
Task Two total elapsed time: 5.0
Task One total elapsed time: 2.0

Total elapsed time: 17.0
```

This indicates that `await asyncio.sleep(delay)` is non-blocking, and that other work is being done.

At the end of the program, you'll notice the total elapsed time is essentially half the time it took for `example_3.py` to run. That's the advantage of a program that uses Python async features! Each task was able to run `await asyncio.sleep(delay)` at the same time. The total execution time of the program is now less than the sum of its parts. You've broken away from the synchronous model!

[Remove ads](#)

Synchronous (Blocking) HTTP Calls

The next version of the program is kind of a step forward as well as a step back. The program is doing some actual work with real IO by making HTTP requests to a list of URLs and getting the page contents. However, it's doing so in a blocking (synchronous) manner.

The program has been modified to import [the wonderful requests module](#) to make the actual HTTP requests. Also, the queue now contains a list of URLs, rather than numbers. In addition, `task()` no longer increments a counter. Instead, `requests` gets the contents of a URL retrieved from the queue, and prints how long it took to do so.

The `example_5.py` code is listed below:

Python

```
1 import queue
2 import requests
3 from codetiming import Timer
4
5 def task(name, work_queue):
6     timer = Timer(text=f"Task {name} elapsed time: {:.1f}")
7     with requests.Session() as session:
8         while not work_queue.empty():
9             url = work_queue.get()
10            print(f"Task {name} getting URL: {url}")
11            timer.start()
12            session.get(url)
13            timer.stop()
14            yield
15
16 def main():
17     """
18     This is the main entry point for the program
19     """
20
21     # Create the queue of work
22     work_queue = queue.Queue()
23
24     # Put some work in the queue
25     for url in [
26         "http://google.com",
27         "http://yahoo.com",
28         "http://linkedin.com",
29         "http://apple.com",
30         "http://microsoft.com",
31         "http://facebook.com",
32         "http://twitter.com",
33     ]:
34         work_queue.put(url)
35
36     tasks = [task("One", work_queue), task("Two", work_queue)]
37
38     # Run the tasks
39     done = False
40     with Timer(text="\nTotal elapsed time: {:.1f}"):
41         while not done:
42             for t in tasks:
43                 try:
44                     next(t)
45                 except StopIteration:
46                     tasks.remove(t)
47                     if len(tasks) == 0:
48                         done = True
49
50 if __name__ == "__main__":
51     main()
```

Here's what's happening in this program:

- **Line 2** imports `requests`, which provides a convenient way to make HTTP calls.
- **Line 3** imports the `Timer` code from the `codetiming` module.
- **Line 6** creates the `Timer` instance used to measure the time taken for each iteration of the task loop.

- **Line 11** starts the `timer` instance
- **Line 12** introduces a delay, similar to `example_3.py`. However, this time it calls `session.get(url)`, which returns the contents of the URL retrieved from `work_queue`.
- **Line 13** stops the `timer` instance and outputs the elapsed time since `timer.start()` was called.
- **Lines 23 to 32** put the list of URLs into `work_queue`.
- **Line 39** creates a `Timer` context manager that will output the elapsed time the entire while loop took to execute.

When you run this program, you'll see the following output:

Shell

```
Task One getting URL: http://google.com
Task One total elapsed time: 0.3
Task Two getting URL: http://yahoo.com
Task Two total elapsed time: 0.8
Task One getting URL: http://linkedin.com
Task One total elapsed time: 0.4
Task Two getting URL: http://apple.com
Task Two total elapsed time: 0.3
Task One getting URL: http://microsoft.com
Task One total elapsed time: 0.5
Task Two getting URL: http://facebook.com
Task Two total elapsed time: 0.5
Task One getting URL: http://twitter.com
Task One total elapsed time: 0.4

Total elapsed time: 3.2
```

Just like in earlier versions of the program, `yield` turns `task()` into a generator. It also performs a context switch that lets the other task instance run.

Each task gets a URL from the work queue, retrieves the contents of the page, and reports how long it took to get that content.

As before, `yield` allows both your tasks to run cooperatively. However, since this program is running synchronously, each `session.get()` call blocks the CPU until the page is retrieved. **Note the total time it took to run the entire program at the end.** This will be meaningful for the next example.

Asynchronous (Non-Blocking) HTTP Calls

This version of the program modifies the previous one to use Python async features. It also imports the [aiohttp](#) module, which is a library to make HTTP requests in an asynchronous fashion using `asyncio`.

The tasks here have been modified to remove the `yield` call since the code to make the HTTP GET call is no longer blocking. It also performs a context switch back to the event loop.

The `example_6.py` program is listed below:

Python

```
1 import asyncio
2 import aiohttp
3 from codetiming import Timer
4
5 async def task(name, work_queue):
6     timer = Timer(text=f"Task {name} elapsed time: {:.1f}")
7     async with aiohttp.ClientSession() as session:
8         while not work_queue.empty():
9             url = await work_queue.get()
10            print(f"Task {name} getting URL: {url}")
11            timer.start()
12            async with session.get(url) as response:
13                await response.text()
14            timer.stop()
15
16 async def main():
17     """
18     This is the main entry point for the program
19     """
20     # Create the queue of work
21     work_queue = asyncio.Queue()
22
23     # Put some work in the queue
24     for url in [
25         "http://google.com",
26         "http://yahoo.com",
27         "http://linkedin.com",
28         "http://apple.com",
29         "http://microsoft.com",
30         "http://facebook.com",
31         "http://twitter.com",
32     ]:
33         await work_queue.put(url)
34
35     # Run the tasks
36     with Timer(text="\nTotal elapsed time: {:.1f}"):
37         await asyncio.gather(
38             asyncio.create_task(task("One", work_queue)),
39             asyncio.create_task(task("Two", work_queue)),
40         )
41
42 if __name__ == "__main__":
43     asyncio.run(main())
```

Here's what's happening in this program:

- **Line 2** imports the aiohttp library, which provides an asynchronous way to make HTTP calls.
- **Line 3** imports the the Timer code from the codetiming module.
- **Line 5** marks task() as an asynchronous function.
- **Line 6** creates the Timer instance used to measure the time taken for each iteration of the task loop.
- **Line 7** creates an aiohttp session context manager.
- **Line 8** creates an aiohttp response context manager. It also makes an HTTP GET call to the URL taken from work_queue.
- **Line 11** starts the timer instance
- **Line 12** uses the session to get the text retrieved from the URL asynchronously.
- **Line 13** stops the timer instance and outputs the elapsed time since timer.start() was called.
- **Line 39** creates a Timer context manager that will output the elapsed time the entire while loop took to execute.

When you run this program, you'll see the following output:

Shell

```
Task One getting URL: http://google.com
Task Two getting URL: http://yahoo.com
Task One total elapsed time: 0.3
Task One getting URL: http://linkedin.com
Task One total elapsed time: 0.3
Task One getting URL: http://apple.com
Task One total elapsed time: 0.3
Task One getting URL: http://microsoft.com
Task Two total elapsed time: 0.9
Task Two getting URL: http://facebook.com
Task Two total elapsed time: 0.4
Task Two getting URL: http://twitter.com
Task One total elapsed time: 0.5
Task Two total elapsed time: 0.3

Total elapsed time: 1.7
```

Take a look at the total elapsed time, as well as the individual times to get the contents of each URL. You'll see that the duration is about half the cumulative time of all the HTTP GET calls. This is because the HTTP GET calls are running asynchronously. In other words, you're effectively taking better advantage of the CPU by allowing it to make multiple requests at once.

Because the CPU is so fast, this example could likely create as many tasks as there are URLs. In this case, the program's run time would be that of the single slowest URL retrieval.

[Remove ads](#)

Conclusion

This article has given you the tools you need to start making asynchronous programming techniques a part of your repertoire. Using Python async features gives you programmatic control of when context switches take place. This means that many of the tougher issues you might see in threaded programming are easier to deal with.

Asynchronous programming is a powerful tool, but it isn't useful for every kind of program. If you're writing a program that calculates pi to the millionth decimal place, for instance, then asynchronous code won't help you. That kind of program is CPU bound, without much IO. However, if you're trying to implement a server or a program that performs IO (like file or network access), then using Python async features could make a huge difference.

To sum it up, you've learned:

- What **synchronous programs** are
- How **asynchronous programs** are different, but also powerful and manageable
- Why you might want to write asynchronous programs
- How to use the built-in `async` features in Python

You can get the code for all of the example programs used in this tutorial:

Download Code: [Click here to download the code you'll use](#) to learn about `async` features in Python in this tutorial.

Now that you're equipped with these powerful skills, you can take your programs to the next level!

About Doug Farrell

Doug is a Python developer with more than 25 years of experience. He writes about Python on his personal website and works as a Senior Web Engineer with Shutterfly.

[» More about Doug](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Brad](#)

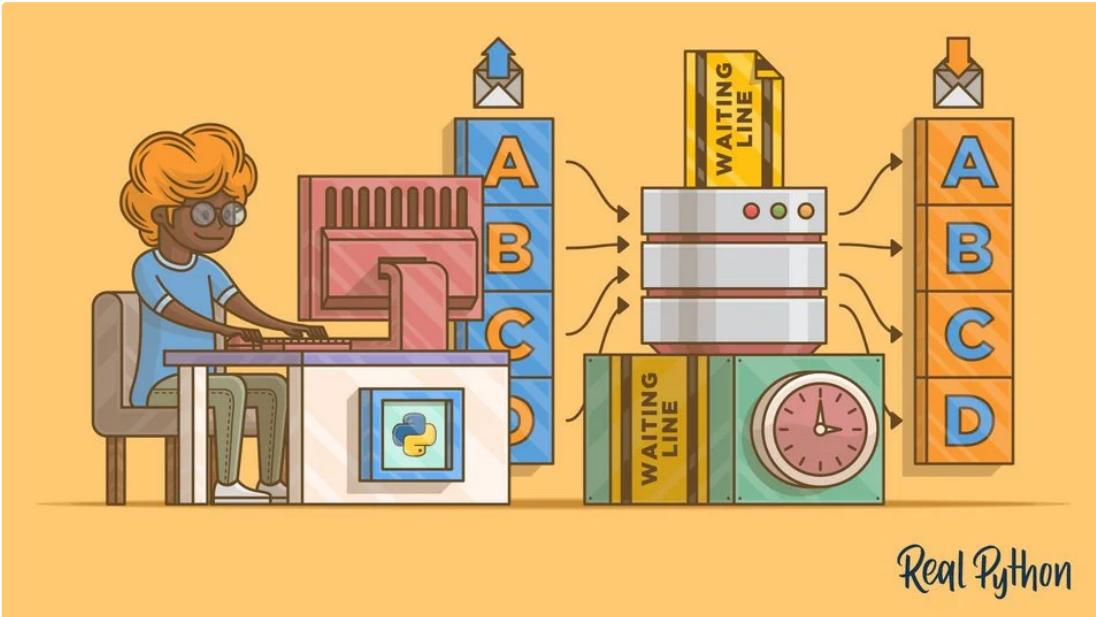
[Geir Arne](#)

[Jaya](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Real Python

Async IO in Python: A Complete Walkthrough

by Brad Solomon 38 Comments Intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Setting Up Your Environment](#)
- [The 10,000-Foot View of Async IO](#)
 - [Where Does Async IO Fit In?](#)
 - [Async IO Explained](#)
 - [Async IO Is Not Easy](#)
- [The asyncio Package and async/await](#)
 - [The async/await Syntax and Native Coroutines](#)
 - [The Rules of Async IO](#)
- [Async IO Design Patterns](#)
 - [Chaining Coroutines](#)
 - [Using a Queue](#)
- [Async IO's Roots in Generators](#)
 - [Other Features: async for and Async Generators + Comprehensions](#)
 - [The Event Loop and asyncio.run\(\)](#)
- [A Full Program: Asynchronous Requests](#)
- [Async IO in Context](#)
 - [When and Why Is Async IO the Right Choice?](#)
 - [Async IO It Is, but Which One?](#)
- [Odds and Ends](#)
 - [Other Top-Level asyncio Functions](#)
 - [The Precedence of await](#)
- [Conclusion](#)
- [Resources](#)
 - [Python Version Specifics](#)
 - [Articles](#)
 - [Related PEPs](#)
 - [Libraries That Work With async/await](#)

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Hands-on Python 3 Concurrency With the asyncio Module](#)

Async IO is a concurrent programming design that has received dedicated support in Python, evolving rapidly from Python 3.4 through 3.7, and [probably beyond](#).

You may be thinking with dread, “Concurrency, parallelism, threading, multiprocessing. That’s a lot to grasp already. Where does async IO fit in?”

This tutorial is built to help you answer that question, giving you a firmer grasp of Python’s approach to async IO.

Here's what you'll cover:

- **Asynchronous IO (async IO)**: a language-agnostic paradigm (model) that has implementations across a host of programming languages
- **async/await**: two new [Python keywords](#) that are used to define coroutines
- **asyncio**: the Python package that provides a foundation and API for running and managing coroutines

Couroutines (specialized generator functions) are the heart of async IO in Python, and we’ll dive into them later on.

Note: In this article, I use the term **async IO** to denote the language-agnostic design of asynchronous IO, while **asyncio** refers to the Python package.

Before you get started, you’ll need to make sure you’re set up to use `asyncio` and other libraries found in this tutorial.

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Setting Up Your Environment

You’ll need Python 3.7 or above to follow this article in its entirety, as well as the `aiohttp` and `aiofiles` packages:

Shell

```
$ python3.7 -m venv ./py37async
$ source ./py37async/bin/activate # Windows: .\py37async\Scripts\activate.bat
$ pip install --upgrade pip aiohttp aiofiles # Optional: aiodns
```

For help with installing Python 3.7 and setting up a virtual environment, check out [Python 3 Installation & Setup Guide](#) or [Virtual Environments Primer](#).

With that, let’s jump in.

The 10,000-Foot View of Async IO

Async IO is a bit lesser known than its tried-and-true cousins, multiprocessing and threading. This section will give you a fuller picture of what async IO is and how it fits into its surrounding landscape.

Where Does Async IO Fit In?

Concurrency and parallelism are expansive subjects that are not easy to wade into. While this article focuses on async IO and its implementation in Python, it’s worth taking a minute to compare async IO to its counterparts in order to have context about how async IO fits into the larger, sometimes dizzying puzzle.

Parallelism consists of performing multiple operations at the same time. **Multiprocessing** is a means to effect parallelism, and it entails spreading tasks over a computer’s central processing units (CPUs, or cores).

Multiprocessing is well-suited for CPU-bound tasks: tightly bound for loops and mathematical computations usually fall into this category.

Concurrency is a slightly broader term than parallelism. It suggests that multiple tasks have the ability to run in an overlapping manner. (There's a saying that concurrency does not imply parallelism.)

Threading is a concurrent execution model whereby multiple [threads](#) take turns executing tasks. One process can contain multiple threads. Python has a complicated relationship with threading thanks to its [GIL](#), but that's beyond the scope of this article.

What's important to know about threading is that it's better for IO-bound tasks. While a CPU-bound task is characterized by the computer's cores continually working hard from start to finish, an IO-bound job is dominated by a lot of waiting on input/output to complete.

To recap the above, concurrency encompasses both multiprocessing (ideal for CPU-bound tasks) and threading (suited for IO-bound tasks). Multiprocessing is a form of parallelism, with parallelism being a specific type (subset) of concurrency. The Python standard library has offered longstanding support for both of these through its [multiprocessing](#), [threading](#), and [concurrent.futures](#) packages.

Now it's time to bring a new member to the mix. Over the last few years, a separate design has been more comprehensively built into CPython: asynchronous IO, enabled through the standard library's [asyncio](#) package and the new [async](#) and [await](#) language keywords. To be clear, [async](#) IO is not a newly invented concept, and it has existed or is being built into other languages and runtime environments, such as [Go](#), [C#](#), or [Scala](#).

The [asyncio](#) package is billed by the Python documentation as [a library to write concurrent code](#). However, [async](#) IO is not threading, nor is it multiprocessing. It is not built on top of either of these.

In fact, [async](#) IO is a single-threaded, single-process design: it uses **cooperative multitasking**, a term that you'll flesh out by the end of this tutorial. It has been said in other words that [async](#) IO gives a feeling of concurrency despite using a single thread in a single process. Coroutines (a central feature of [async](#) IO) can be scheduled concurrently, but they are not inherently concurrent.

To reiterate, [async](#) IO is a style of concurrent programming, but it is not parallelism. It's more closely aligned with threading than with multiprocessing but is very much distinct from both of these and is a standalone member in concurrency's bag of tricks.

That leaves one more term. What does it mean for something to be **asynchronous**? This isn't a rigorous definition, but for our purposes here, I can think of two properties:

- Asynchronous routines are able to "pause" while waiting on their ultimate result and let other routines run in the meantime.
- Asynchronous code, through the mechanism above, facilitates concurrent execution. To put it differently, asynchronous code gives the look and feel of concurrency.

Here's a diagram to put it all together. The white terms represent concepts, and the green terms represent ways in which they are implemented or effected:

I'll stop there on the comparisons between concurrent programming models. This tutorial is focused on the subcomponent that is async IO, how to use it, and the APIs that have sprung up around it. For a thorough exploration of threading versus multiprocessing versus async IO, pause here and check out Jim Anderson's [overview of concurrency in Python](#). Jim is way funnier than me and has sat in more meetings than me, to boot.

Async IO Explained

Async IO may at first seem counterintuitive and paradoxical. How does something that facilitates concurrent code use a single thread and a single CPU core? I've never been very good at conjuring up examples, so I'd like to paraphrase one from Miguel Grinberg's 2017 PyCon talk, which explains everything quite beautifully:

Chess master Judit Polgár hosts a chess exhibition in which she plays multiple amateur players. She has two ways of conducting the exhibition: synchronously and asynchronously.

Assumptions:

- 24 opponents
- Judit makes each chess move in 5 seconds
- Opponents each take 55 seconds to make a move
- Games average 30 pair-moves (60 moves total)

Synchronous version: Judit plays one game at a time, never two at the same time, until the game is complete. Each game takes $(55 + 5) * 30 == 1800$ seconds, or 30 minutes. The entire exhibition takes $24 * 30 == 720$ minutes, or **12 hours**.

Asynchronous version: Judit moves from table to table, making one move at each table. She leaves the table and lets the opponent make their next move during the wait time. One move on all 24 games takes Judit $24 * 5 == 120$ seconds, or 2 minutes. The entire exhibition is now cut down to $120 * 30 == 3600$ seconds, or just **1 hour**. [\(Source\)](#)

There is only one Judit Polgár, who has only two hands and makes only one move at a time by herself. But playing asynchronously cuts the exhibition time down from 12 hours to one. So, cooperative multitasking is a fancy way of saying that a program's event loop (more on that later) communicates with multiple tasks to let each take turns running at the optimal time.

Async IO takes long waiting periods in which functions would otherwise be blocking and allows other functions to run during that downtime. (A function that blocks effectively forbids others from running from the time that it starts until the time that it returns.)

Async IO Is Not Easy

I've heard it said, "Use async IO when you can; use threading when you must." The truth is that building durable multithreaded code can be hard and error-prone. Async IO avoids some of the potential speedbumps that you might otherwise encounter with a threaded design.

But that's not to say that async IO in Python is easy. Be warned: when you venture a bit below the surface level, async programming can be difficult too! Python's async model is built around concepts such as callbacks, events, transports, protocols, and futures—just the terminology can be intimidating. The fact that its API has been changing continually makes it no easier.

Luckily, `asyncio` has matured to a point where most of its features are no longer provisional, while its documentation has received a huge overhaul and some quality resources on the subject are starting to emerge as well.

The asyncio Package and `async/await`

Now that you have some background on async IO as a design, let's explore Python's implementation. Python's `asyncio` package (introduced in Python 3.4) and its two keywords, `async` and `await`, serve different purposes but come together to help you declare, build, execute, and manage asynchronous code.

The `async/await` Syntax and Native Coroutines

A Word of Caution: Be careful what you read out there on the Internet. Python's async IO API has evolved rapidly from Python 3.4 to Python 3.7. Some old patterns are no longer used, and some things that were at first disallowed are now allowed through new introductions. For all I know, this tutorial will join the club of the outdated soon too.

At the heart of async IO are coroutines. A coroutine is a specialized version of a Python generator function. Let's start with a baseline definition and then build off of it as you progress here: a coroutine is a function that can suspend its execution before reaching `return`, and it can indirectly pass control to another coroutine for some time.

Later, you'll dive a lot deeper into how exactly the traditional generator is repurposed into a coroutine. For now, the easiest way to pick up how coroutines work is to start making some.

Let's take the immersive approach and write some async IO code. This short program is the Hello World of async IO but goes a long way towards illustrating its core functionality:

Python

```
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"__file__ executed in {elapsed:0.2f} seconds.")
```

When you execute this file, take note of what looks different than if you were to define the functions with just `def` and `time.sleep()`:

Shell

```
$ python3 countasync.py
One
One
One
Two
Two
Two
countasync.py executed in 1.01 seconds.
```

The order of this output is the heart of async IO. Talking to each of the calls to `count()` is a single event loop, or coordinator. When each task reaches `await asyncio.sleep(1)`, the function yells up to the event loop and gives control back to it, saying, "I'm going to be sleeping for 1 second. Go ahead and let something else meaningful be done in the meantime."

Contrast this to the synchronous version:

Python

```
#!/usr/bin/env python3
# countsync.py

import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

When executed, there is a slight but critical change in order and execution time:

Shell

```
$ python3 countsync.py
One
Two
One
Two
One
Two
countsync.py executed in 3.01 seconds.
```

While using `time.sleep()` and `asyncio.sleep()` may seem banal, they are used as stand-ins for any time-intensive processes that involve wait time. (The most mundane thing you can wait on is a `sleep()` call that does basically nothing.) That is, `time.sleep()` can represent any time-consuming blocking function call, while `asyncio.sleep()` is used to stand in for a non-blocking call (but one that also takes some time to complete).

As you'll see in the next section, the benefit of awaiting something, including `asyncio.sleep()`, is that the surrounding function can temporarily cede control to another function that's more readily able to do something immediately. In contrast, `time.sleep()` or any other blocking call is incompatible with asynchronous Python code, because it will stop everything in its tracks for the duration of the sleep time.

The Rules of Async IO

At this point, a more formal definition of `async`, `await`, and the coroutine functions that they create are in order. This section is a little dense, but getting a hold of `async/await` is instrumental, so come back to this if you need to:

- The syntax `async def` introduces either a **native coroutine** or an **asynchronous generator**. The expressions `async with` and `async for` are also valid, and you'll see them later on.
- The keyword `await` passes function control back to the event loop. (It suspends the execution of the surrounding coroutine.) If Python encounters an `await f()` expression in the scope of `g()`, this is how `await` tells the event loop, “Suspend execution of `g()` until whatever I'm waiting on—the result of `f()`—is returned. In the meantime, go let something else run.”

In code, that second bullet point looks roughly like this:

Python

```
async def g():
    # Pause here and come back to g() when f() is ready
    r = await f()
    return r
```

There's also a strict set of rules around when and how you can and cannot use `async/await`. These can be handy whether you are still picking up the syntax or already have exposure to using `async/await`.

- A function that you introduce with `async def` is a coroutine. It may use `await`, `return`, or `yield`, but all of these are optional. Declaring `async def noop(): pass` is valid:
 - Using `await` and/or `return` creates a coroutine function. To call a coroutine function, you must `await` it to get its results.
 - It is less common (and only recently legal in Python) to use `yield` in an `async def` block. This creates an [asynchronous generator](#), which you iterate over with `async for`. Forget about `async` generators for the time being and focus on getting down the syntax for coroutine functions, which use `await` and/or `return`.
 - Anything defined with `async def` may not use `yield from`, which will raise a [SyntaxError](#).
- Just like it's a `SyntaxError` to use `yield` outside of a `def` function, it is a `SyntaxError` to use `await` outside of an `async def` coroutine. You can only use `await` in the body of coroutines.

Here are some terse examples meant to summarize the above few rules:

Python

```
async def f(x):
    y = await z(x) # OK - `await` and `return` allowed in coroutines
    return y

async def g(x):
    yield x # OK - this is an async generator

async def m(x):
    yield from gen(x) # No - SyntaxError

def m(x):
    y = await z(x) # Still no - SyntaxError (no `async def` here)
    return y
```

Finally, when you use `await f()`, it's required that `f()` be an object that is [awaitable](#). Well, that's not very helpful, is it? For now, just know that an awaitable object is either (1) another coroutine or (2) an object defining an `__await__()` dunder method that returns an iterator. If you're writing a program, for the large majority of purposes, you should only need to worry about case #1.

That brings us to one more technical distinction that you may see pop up: an older way of marking a function as a coroutine is to decorate a normal `def` function with `@asyncio.coroutine`. The result is a **generator-based coroutine**. This construction has been outdated since the `async/await` syntax was put in place in Python 3.5. These two coroutines are essentially equivalent (both are awaitable), but the first is **generator-based**, while the second is a **native coroutine**:

Python

```
import asyncio

@asyncio.coroutine
def py34_coro():
    """Generator-based coroutine, older syntax"""
    yield from stuff()

async def py35_coro():
    """Native coroutine, modern syntax"""
    await stuff()
```

If you're writing any code yourself, prefer native coroutines for the sake of being explicit rather than implicit. Generator-based coroutines will be [removed](#) in Python 3.10.

Towards the latter half of this tutorial, we'll touch on generator-based coroutines for explanation's sake only. The reason that `async/await` were introduced is to make coroutines a standalone feature of Python that can be easily differentiated from a normal generator function, thus reducing ambiguity.

Don't get bogged down in generator-based coroutines, which have been [deliberately outdated](#) by `async/await`. They have their own small set of rules (for instance, `await` cannot be used in a generator-based coroutine) that are largely irrelevant if you stick to the `async/await` syntax.

Without further ado, let's take on a few more involved examples.

Here's one example of how async IO cuts down on wait time: given a coroutine `makerandom()` that keeps producing random integers in the range [0, 10], until one of them exceeds a threshold, you want to let multiple calls of this coroutine not need to wait for each other to complete in succession. You can largely follow the patterns from the two scripts above, with slight changes:

Python

```
#!/usr/bin/env python3
# rand.py

import asyncio
import random

# ANSI colors
c = (
    "\u033[0m",    # End of color
    "\u033[36m",   # Cyan
    "\u033[91m",   # Red
    "\u033[35m",   # Magenta
)

async def makerandom(idx: int, threshold: int = 6) -> int:
    print(c[idx + 1] + f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(c[idx + 1] + f"makerandom({idx}) == {i} too low; retrying.")
        await asyncio.sleep(idx + 1)
        i = random.randint(0, 10)
    print(c[idx + 1] + f"--> Finished: makerandom({idx}) == {i}" + c[0])
    return i

async def main():
    res = await asyncio.gather(*[makerandom(i, 10 - i - 1) for i in range(3)])
    return res

if __name__ == "__main__":
    random.seed(444)
    r1, r2, r3 = asyncio.run(main())
    print()
```

```
    print(f"r1: {r1}, r2: {r2}, r3: {r3}")
```

The colorized output says a lot more than I can and gives you a sense for how this script is carried out:

rand.py execution

This program uses one main coroutine, `makerandom()`, and runs it concurrently across 3 different inputs. Most programs will contain small, modular coroutines and one wrapper function that serves to chain each of the smaller coroutines together. `main()` is then used to gather tasks (futures) by mapping the central coroutine across some iterable or pool.

In this miniature example, the pool is `range(3)`. In a fuller example presented later, it is a set of URLs that need to be requested, parsed, and processed concurrently, and `main()` encapsulates that entire routine for each URL.

While “making random integers” (which is CPU-bound more than anything) is maybe not the greatest choice as a candidate for `asyncio`, it’s the presence of `asyncio.sleep()` in the example that is designed to mimic an IO-bound process where there is uncertain wait time involved. For example, the `asyncio.sleep()` call might represent sending and receiving not-so-random integers between two clients in a message application.

Async IO Design Patterns

Async IO comes with its own set of possible script designs, which you’ll get introduced to in this section.

Chaining Coroutines

A key feature of coroutines is that they can be chained together. (Remember, a coroutine object is awaitable, so another coroutine can await it.) This allows you to break programs into smaller, manageable, recyclable coroutines:

Python

```
#!/usr/bin/env python3
# chained.py

import asyncio
import random
import time

async def part1(n: int) -> str:
    i = random.randint(0, 10)
    print(f"part1({n}) sleeping for {i} seconds.")
    await asyncio.sleep(i)
    result = f"result{n}-1"
    print(f"Returning part1({n}) == {result}.")
    return result

async def part2(n: int, arg: str) -> str:
    i = random.randint(0, 10)
    print(f"part2{n, arg} sleeping for {i} seconds.")
    await asyncio.sleep(i)
```

```

        await asyncio.sleep(1,
    result = f"result{n}-2 derived from {arg}"
    print(f"Returning part2{n, arg} == {result}.")
    return result

async def chain(n: int) -> None:
    start = time.perf_counter()
    p1 = await part1(n)
    p2 = await part2(n, p1)
    end = time.perf_counter() - start
    print(f"-->Chained result{n} => {p2} (took {end:.2f} seconds.)")

async def main(*args):
    await asyncio.gather(*(chain(n) for n in args))

if __name__ == "__main__":
    import sys
    random.seed(444)
    args = [1, 2, 3] if len(sys.argv) == 1 else map(int, sys.argv[1:])
    start = time.perf_counter()
    asyncio.run(main(*args))
    end = time.perf_counter() - start
    print(f"Program finished in {end:.2f} seconds.")

```

Pay careful attention to the output, where `part1()` sleeps for a variable amount of time, and `part2()` begins working with the results as they become available:

Shell

```

$ python3 chained.py 9 6 3
part1(9) sleeping for 4 seconds.
part1(6) sleeping for 4 seconds.
part1(3) sleeping for 0 seconds.
Returning part1(3) == result3-1.
part2(3, 'result3-1') sleeping for 4 seconds.
Returning part1(9) == result9-1.
part2(9, 'result9-1') sleeping for 7 seconds.
Returning part1(6) == result6-1.
part2(6, 'result6-1') sleeping for 4 seconds.
Returning part2(3, 'result3-1') == result3-2 derived from result3-1.
-->Chained result3 => result3-2 derived from result3-1 (took 4.00 seconds).
Returning part2(6, 'result6-1') == result6-2 derived from result6-1.
-->Chained result6 => result6-2 derived from result6-1 (took 8.01 seconds).
Returning part2(9, 'result9-1') == result9-2 derived from result9-1.
-->Chained result9 => result9-2 derived from result9-1 (took 11.01 seconds).
Program finished in 11.01 seconds.

```

In this setup, the runtime of `main()` will be equal to the maximum runtime of the tasks that it gathers together and schedules.

Using a Queue

The `asyncio` package provides [queue classes](#) that are designed to be similar to classes of the `queue` module. In our examples so far, we haven't really had a need for a queue structure. In `chained.py`, each task (future) is composed of a set of coroutines that explicitly await each other and pass through a single input per chain.

There is an alternative structure that can also work with async IO: a number of producers, which are not associated with each other, add items to a queue. Each producer may add multiple items to the queue at staggered, random, unannounced times. A group of consumers pull items from the queue as they show up, greedily and without waiting for any other signal.

In this design, there is no chaining of any individual consumer to a producer. The consumers don't know the number of producers, or even the cumulative number of items that will be added to the queue, in advance.

It takes an individual producer or consumer a variable amount of time to put and extract items from the queue,

respectively. The queue serves as a throughput that can communicate with the producers and consumers without them talking to each other directly.

Note: While queues are often used in threaded programs because of the thread-safety of `queue.Queue()`, you shouldn't need to concern yourself with thread safety when it comes to async IO. (The exception is when you're combining the two, but that isn't done in this tutorial.)

One use-case for queues (as is the case here) is for the queue to act as a transmitter for producers and consumers that aren't otherwise directly chained or associated with each other.

The synchronous version of this program would look pretty dismal: a group of blocking producers serially add items to the queue, one producer at a time. Only after all producers are done can the queue be processed, by one consumer at a time processing item-by-item. There is a ton of latency in this design. Items may sit idly in the queue rather than be picked up and processed immediately.

An asynchronous version, `asyncq.py`, is below. The challenging part of this workflow is that there needs to be a signal to the consumers that production is done. Otherwise, `await q.get()` will hang indefinitely, because the queue will have been fully processed, but consumers won't have any idea that production is complete.

(Big thanks for some help from a StackOverflow [user](#) for helping to straighten out `main()`: the key is to `await q.join()`, which blocks until all items in the queue have been received and processed, and then to cancel the consumer tasks, which would otherwise hang up and wait endlessly for additional queue items to appear.)

Here is the full script:

Python

```
#!/usr/bin/env python3
# asyncq.py

import asyncio
import itertools as it
import os
import random
import time

async def makeitem(size: int = 5) -> str:
    return os.urandom(size).hex()

async def randsleep(a: int = 1, b: int = 5, caller=None) -> None:
    i = random.randint(a, b)
    if caller:
        print(f"{caller} sleeping for {i} seconds.")
    await asyncio.sleep(i)

async def produce(name: int, q: asyncio.Queue) -> None:
    n = random.randint(0, 10)
    for _ in it.repeat(None, n): # Synchronous loop for each single producer
        await randsleep(caller=f"Producer {name}")
        i = await makeitem()
        t = time.perf_counter()
        await q.put((i, t))
        print(f"Producer {name} added <{i}> to queue.")

async def consume(name: int, q: asyncio.Queue) -> None:
    while True:
        await randsleep(caller=f"Consumer {name}")
        i, t = await q.get()
        now = time.perf_counter()
        print(f"Consumer {name} got element <{i}>\n"
              f"  in {now-t:0.5f} seconds.")
        q.task_done()

async def main(nprod: int, ncon: int):
    q = asyncio.Queue()
    producers = [asyncio.create_task(produce(n, q)) for n in range(nprod)]
    consumers = [asyncio.create_task(consume(n, q)) for n in range(ncon)]
    await asyncio.gather(*producers)
    await q.join() # Implicitly awaits consumers, too
    for c in consumers:
        c.cancel()
```

```

if __name__ == "__main__":
    import argparse
    random.seed(444)
    parser = argparse.ArgumentParser()
    parser.add_argument("-p", "--nprod", type=int, default=5)
    parser.add_argument("-c", "--ncon", type=int, default=10)
    ns = parser.parse_args()
    start = time.perf_counter()
    asyncio.run(main(**ns.__dict__))
    elapsed = time.perf_counter() - start
    print(f"Program completed in {elapsed:.0f} seconds.")

```

The first few coroutines are helper functions that return a random string, a fractional-second performance counter, and a random integer. A producer puts anywhere from 1 to 5 items into the queue. Each item is a tuple of (i, t) where i is a random string and t is the time at which the producer attempts to put the tuple into the queue.

When a consumer pulls an item out, it simply calculates the elapsed time that the item sat in the queue using the timestamp that the item was put in with.

Keep in mind that `asyncio.sleep()` is used to mimic some other, more complex coroutine that would eat up time and block all other execution if it were a regular blocking function.

Here is a test run with two producers and five consumers:

Shell

```

$ python3 asyncq.py -p 2 -c 5
Producer 0 sleeping for 3 seconds.
Producer 1 sleeping for 3 seconds.
Consumer 0 sleeping for 4 seconds.
Consumer 1 sleeping for 3 seconds.
Consumer 2 sleeping for 3 seconds.
Consumer 3 sleeping for 5 seconds.
Consumer 4 sleeping for 4 seconds.
Producer 0 added <377b1e8f82> to queue.
Producer 0 sleeping for 5 seconds.
Producer 1 added <413b8802f8> to queue.
Consumer 1 got element <377b1e8f82> in 0.00013 seconds.
Consumer 1 sleeping for 3 seconds.
Consumer 2 got element <413b8802f8> in 0.00009 seconds.
Consumer 2 sleeping for 4 seconds.
Producer 0 added <06c055b3ab> to queue.
Producer 0 sleeping for 1 seconds.
Consumer 0 got element <06c055b3ab> in 0.00021 seconds.
Consumer 0 sleeping for 4 seconds.
Producer 0 added <17a8613276> to queue.
Consumer 4 got element <17a8613276> in 0.00022 seconds.
Consumer 4 sleeping for 5 seconds.
Program completed in 9.00954 seconds.

```

In this case, the items process in fractions of a second. A delay can be due to two reasons:

- Standard, largely unavoidable overhead
- Situations where all consumers are sleeping when an item appears in the queue

With regards to the second reason, luckily, it is perfectly normal to scale to hundreds or thousands of consumers. You should have no problem with `python3 asyncq.py -p 5 -c 100`. The point here is that, theoretically, you could have different users on different systems controlling the management of producers and consumers, with the queue serving as the central throughput.

So far, you've been thrown right into the fire and seen three related examples of `asyncio` calling coroutines defined with `async` and `await`. If you're not completely following or just want to get deeper into the mechanics of how modern coroutines came to be in Python, you'll start from square one with the next section.

Async IO's Roots in Generators

Earlier, you saw an example of the old-style generator-based coroutines, which have been outdated by more explicit native coroutines. The example is worth re-showing with a small tweak:

Python

```
import asyncio

@asyncio.coroutine
def py34_coro():
    """Generator-based coroutine"""
    # No need to build these yourself, but be aware of what they are
    s = yield from stuff()
    return s

async def py35_coro():
    """Native coroutine, modern syntax"""
    s = await stuff()
    return s

async def stuff():
    return 0x10, 0x20, 0x30
```

As an experiment, what happens if you call `py34_coro()` or `py35_coro()` on its own, without `await`, or without any calls to `asyncio.run()` or other `asyncio` “porcelain” functions? Calling a coroutine in isolation returns a coroutine object:

Python

```
>>> py35_coro()
<coroutine object py35_coro at 0x10126dcc8>
```

This isn’t very interesting on its surface. The result of calling a coroutine on its own is an awaitable **coroutine object**.

Time for a quiz: what other feature of Python looks like this? (What feature of Python doesn’t actually “do much” when it’s called on its own?)

Hopefully you’re thinking of **generators** as an answer to this question, because coroutines are enhanced generators under the hood. The behavior is similar in this regard:

Python

```
>>> def gen():
...     yield 0x10, 0x20, 0x30
...
>>> g = gen()
>>> g # Nothing much happens - need to iterate with `.__next__()`
<generator object gen at 0x1012705e8>
>>> next(g)
(16, 32, 48)
```

Generator functions are, as it so happens, the foundation of async IO (regardless of whether you declare coroutines with `async def` rather than the older `@asyncio.coroutine` wrapper). Technically, `await` is more closely analogous to `yield from` than it is to `yield`. (But remember that `yield from x()` is just syntactic sugar to replace `for i in x(): yield i`.)

One critical feature of generators as it pertains to async IO is that they can effectively be stopped and restarted at will. For example, you can break out of iterating over a generator object and then resume iteration on the remaining values later. When a generator function reaches `yield`, it yields that value, but then it sits idle until it is told to `yield` its subsequent value.

This can be fleshed out through an example:

Python

```
>>> from itertools import cycle
>>> def endless():
...     """Yields 9, 8, 7, 6, 9, 8, 7, 6, ... forever"""
...     yield from cycle((9, 8, 7, 6))

>>> e = endless()
>>> total = 0
>>> for i in e:
...     total += i
...     if total > 100:
...         break
```

```

...
    if total < 30:
...
        print(i, end=" ")
        total += i
...
    else:
...
        print()
...
        # Pause execution. We can resume later.
...
        break
9 8 7 6 9 8 7 6 9 8 7 6 9 8

>>> # Resume
>>> next(e), next(e), next(e)
(6, 9, 8)

```

The `await` keyword behaves similarly, marking a break point at which the coroutine suspends itself and lets other coroutines work. “Suspended,” in this case, means a coroutine that has temporarily ceded control but not totally exited or finished. Keep in mind that `yield`, and by extension `yield from` and `await`, mark a break point in a generator’s execution.

This is the fundamental difference between functions and generators. A function is all-or-nothing. Once it starts, it won’t stop until it hits a `return`, then pushes that value to the caller (the function that calls it). A generator, on the other hand, pauses each time it hits a `yield` and goes no further. Not only can it push this value to calling stack, but it can keep a hold of its local variables when you resume it by calling `next()` on it.

There’s a second and lesser-known feature of generators that also matters. You can send a value into a generator as well through its `.send()` method. This allows generators (and coroutines) to call (`await`) each other without blocking. I won’t get any further into the nuts and bolts of this feature, because it matters mainly for the implementation of coroutines behind the scenes, but you shouldn’t ever really need to use it directly yourself.

If you’re interested in exploring more, you can start at [PEP 342](#), where coroutines were formally introduced. Brett Cannon’s [How the Heck Does Async-Await Work in Python](#) is also a good read, as is the [PYMOTW writeup on asyncio](#). Lastly, there’s David Beazley’s [Curious Course on Coroutines and Concurrency](#), which dives deep into the mechanism by which coroutines run.

Let’s try to condense all of the above articles into a few sentences: there is a particularly unconventional mechanism by which these coroutines actually get run. Their result is an attribute of the exception object that gets thrown when their `.send()` method is called. There’s some more wonky detail to all of this, but it probably won’t help you use this part of the language in practice, so let’s move on for now.

To tie things together, here are some key points on the topic of coroutines as generators:

- Coroutines are [repurposed generators](#) that take advantage of the peculiarities of generator methods.
- Old generator-based coroutines use `yield from` to wait for a coroutine result. Modern Python syntax in native coroutines simply replaces `yield from` with `await` as the means of waiting on a coroutine result. The `await` is analogous to `yield from`, and it often helps to think of it as such.
- The use of `await` is a signal that marks a break point. It lets a coroutine temporarily suspend execution and permits the program to come back to it later.

Other Features: `async for` and Async Generators + Comprehensions

Along with plain `async/await`, Python also enables `async for` to iterate over an **asynchronous iterator**. The purpose of an asynchronous iterator is for it to be able to call asynchronous code at each stage when it is iterated over.

A natural extension of this concept is an **asynchronous generator**. Recall that you can use `await`, `return`, or `yield` in a native coroutine. Using `yield` within a coroutine became possible in Python 3.6 (via PEP 525), which introduced asynchronous generators with the purpose of allowing `await` and `yield` to be used in the same coroutine function body:

Python	>>>
<pre>>>> async def mygen(u: int = 10): ... """Yield powers of 2.""" </pre>	

```
...     i = 0
...     while i < u:
...         yield 2 ** i
...         i += 1
...     await asyncio.sleep(0.1)
```

Last but not least, Python enables **asynchronous comprehension** with `async for`. Like its synchronous cousin, this is largely syntactic sugar:

```
Python >>>
>>> async def main():
...     # This does *not* introduce concurrent execution
...     # It is meant to show syntax only
...     g = [i async for i in mygen()]
...     f = [j async for j in mygen() if not (j // 3 % 5)]
...     return g, f
...
>>> g, f = asyncio.run(main())
>>> g
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> f
[1, 2, 16, 32, 256, 512]
```

This is a crucial distinction: **neither asynchronous generators nor comprehensions make the iteration concurrent**. All that they do is provide the look-and-feel of their synchronous counterparts, but with the ability for the loop in question to give up control to the event loop for some other coroutine to run.

In other words, asynchronous iterators and asynchronous generators are not designed to concurrently map some function over a sequence or iterator. They're merely designed to let the enclosing coroutine allow other tasks to take their turn. The `async for` and `async with` statements are only needed to the extent that using plain `for` or `with` would “break” the nature of `await` in the coroutine. This distinction between asynchronicity and concurrency is a key one to grasp.

The Event Loop and `asyncio.run()`

You can think of an event loop as something like a `while True` loop that monitors coroutines, taking feedback on what's idle, and looking around for things that can be executed in the meantime. It is able to wake up an idle coroutine when whatever that coroutine is waiting on becomes available.

Thus far, the entire management of the event loop has been implicitly handled by one function call:

```
Python
asyncio.run(main()) # Python 3.7+
```

`asyncio.run()`, introduced in Python 3.7, is responsible for getting the event loop, running tasks until they are marked as complete, and then closing the event loop.

There's a more long-winded way of managing the `asyncio` event loop, with `get_event_loop()`. The typical pattern looks like this:

```
Python
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

You'll probably see `loop.get_event_loop()` floating around in older examples, but unless you have a specific need to fine-tune control over the event loop management, `asyncio.run()` should be sufficient for most programs.

If you do need to interact with the event loop within a Python program, `loop` is a good-old-fashioned Python object that supports introspection with `loop.is_running()` and `loop.is_closed()`. You can manipulate it if you need to

get more fine-tuned control, such as in [scheduling a callback](#) by passing the loop as an argument.

What is more crucial is understanding a bit beneath the surface about the mechanics of the event loop. Here are a few points worth stressing about the event loop.

#1: Coroutines don't do much on their own until they are tied to the event loop.

You saw this point before in the explanation on generators, but it's worth restating. If you have a main coroutine that awaits others, simply calling it in isolation has little effect:

Python

```
>>> import asyncio  
  
>>> async def main():  
...     print("Hello ...")  
...     await asyncio.sleep(1)  
...     print("World!")  
  
>>> routine = main()  
>>> routine  
<coroutine object main at 0x1027a6150>
```

>>>

Remember to use `asyncio.run()` to actually force execution by scheduling the `main()` coroutine (future object) for execution on the event loop:

Python

```
>>> asyncio.run(routine)  
Hello ...  
World!
```

>>>

(Other coroutines can be executed with `await`. It is typical to wrap just `main()` in `asyncio.run()`, and chained coroutines with `await` will be called from there.)

#2: By default, an async IO event loop runs in a single thread and on a single CPU core. Usually, running one single-threaded event loop in one CPU core is more than sufficient. It is also possible to run event loops across multiple cores. Check out this [talk by John Reese](#) for more, and be warned that your laptop may spontaneously combust.

#3. Event loops are pluggable. That is, you could, if you really wanted, write your own event loop implementation and have it run tasks just the same. This is wonderfully demonstrated in the [uvloop](#) package, which is an implementation of the event loop in Cython.

That is what is meant by the term “pluggable event loop”: you can use any working implementation of an event loop, unrelated to the structure of the coroutines themselves. The `asyncio` package itself ships with [two different event loop implementations](#), with the default being based on the `selectors` module. (The second implementation is built for Windows only.)

A Full Program: Asynchronous Requests

You've made it this far, and now it's time for the fun and painless part. In this section, you'll build a web-scraping URL collector, `areq.py`, using `aiohttp`, a blazingly fast async HTTP client/server framework. (We just need the client part.) Such a tool could be used to map connections between a cluster of sites, with the links forming a [directed graph](#).

Note: You may be wondering why Python's `requests` package isn't compatible with async IO. `requests` is built on top of `urllib3`, which in turn uses Python's `http` and `socket` modules.

By default, socket operations are blocking. This means that Python won't like `await requests.get(url)` because `.get()` is not awaitable. In contrast, almost everything in `aiohttp` is an awaitable coroutine, such as `session.request()` and `response.text()`. It's a great package otherwise, but you're doing yourself a disservice by using `requests` in asynchronous code.

The high-level program structure will look like this:

1. Read a sequence of URLs from a local file, `urls.txt`.
2. Send GET requests for the URLs and decode the resulting content. If this fails, stop there for a URL.
3. Search for the URLs within `href` tags in the HTML of the responses.
4. Write the results to `foundurls.txt`.
5. Do all of the above as asynchronously and concurrently as possible. (Use `aiohttp` for the requests, and `aiofiles` for the file-appends. These are two primary examples of IO that are well-suited for the async IO model.)

Here are the contents of `urls.txt`. It's not huge, and contains mostly highly trafficked sites:

Shell

```
$ cat urls.txt
https://regex101.com/
https://docs.python.org/3>this-url-will-404.html
https://www.nytimes.com/guides/
https://www.mediamatters.org/
https://1.1.1.1/
https://www.politico.com/tipsheets/morning-money
https://www.bloomberg.com/markets/economics
https://www.ietf.org/rfc/rfc2616.txt
```

The second URL in the list should return a 404 response, which you'll need to handle gracefully. If you're running an expanded version of this program, you'll probably need to deal with much hairier problems than this, such as server disconnections and endless redirects.

The requests themselves should be made using a single session, to take advantage of reusage of the session's internal connection pool.

Let's take a look at the full program. We'll walk through things step-by-step after:

Python

```
#!/usr/bin/env python3
# areq.py

"""Asynchronously get links embedded in multiple pages' HTML."""

import asyncio
import logging
import re
import sys
from typing import IO
import urllib.error
import urllib.parse

import aiofiles
import aiohttp
from aiohttp import ClientSession

logging.basicConfig(
    format="%(asctime)s %(levelname)s:%(name)s: %(message)s",
    level=logging.DEBUG,
    datefmt="%H:%M:%S",
    stream=sys.stderr,
)
logger = logging.getLogger("areq")
logging.getLogger("chardet.charsetprober").disabled = True

HREF_RE = re.compile(r'href="(.*?)"')

async def fetch_html(url: str, session: ClientSession, **kwargs) -> str:
    """GET request wrapper to fetch page HTML.

    kwargs are passed to `session.request()`.

    """
    resp = await session.request(method="GET", url=url, **kwargs)
    resp.raise_for_status()
    logger.info("Got response [%s] for URL: %s", resp.status, url)
```

```

        html = await resp.text()
        return html

    async def parse(url: str, session: ClientSession, **kwargs) -> set:
        """Find HREFs in the HTML of `url`."""
        found = set()
        try:
            html = await fetch_html(url=url, session=session, **kwargs)
        except (
            aiohttp.ClientError,
            aiohttp.http_exceptions.HttpProcessingError,
        ) as e:
            logger.error(
                "aiohttp exception for %s [%s]: %s",
                url,
                getattr(e, "status", None),
                getattr(e, "message", None),
            )
        return found
    except Exception as e:
        logger.exception(
            "Non-aiohttp exception occurred: %s", getattr(e, "__dict__", {})
        )
    return found
else:
    for link in HREF_RE.findall(html):
        try:
            abslink = urllib.parse.urljoin(url, link)
        except (urllib.error.URLError, ValueError):
            logger.exception("Error parsing URL: %s", link)
            pass
        else:
            found.add(abslink)
    logger.info("Found %d links for %s", len(found), url)
    return found

async def write_one(file: IO, url: str, **kwargs) -> None:
    """Write the found HREFs from `url` to `file`."""
    res = await parse(url=url, **kwargs)
    if not res:
        return None
    async with aiofiles.open(file, "a") as f:
        for p in res:
            await f.write(f"{url}\t{p}\n")
    logger.info("Wrote results for source URL: %s", url)

async def bulk_crawl_and_write(file: IO, urls: set, **kwargs) -> None:
    """Crawl & write concurrently to `file` for multiple `urls`."""
    async with ClientSession() as session:
        tasks = []
        for url in urls:
            tasks.append(
                write_one(file=file, url=url, session=session, **kwargs)
            )
        await asyncio.gather(*tasks)

if __name__ == "__main__":
    import pathlib
    import sys

    assert sys.version_info >= (3, 7), "Script requires Python 3.7+."
    here = pathlib.Path(__file__).parent

    with open(here.joinpath("urls.txt")) as infile:
        urls = set(map(str.strip, infile))

    outpath = here.joinpath("foundurls.txt")
    with open(outpath, "w") as outfile:
        outfile.write("source_url\tparsed_url\n")

    asyncio.run(bulk_crawl_and_write(file=outpath, urls=urls))

```

This script is longer than our initial toy programs, so let's break it down.

The constant `HREF_RE` is a [regular expression](#) to extract what we're ultimately searching for, `href` tags within HTML:

Python

>>>

```
>>> HREF_RE.search('Go to <a href="https://realpython.com/">Real Python</a>')
<re.Match object; span=(15, 45), match='href="https://realpython.com/"'>
```

The coroutine `fetch_html()` is a wrapper around a GET request to make the request and decode the resulting page HTML. It makes the request, awaits the response, and raises right away in the case of a non-200 status:

Python

```
resp = await session.request(method="GET", url=url, **kwargs)
resp.raise_for_status()
```

If the status is okay, `fetch_html()` returns the page HTML (a str). Notably, there is no exception handling done in this function. The logic is to propagate that exception to the caller and let it be handled there:

Python

```
html = await resp.text()
```

We await `session.request()` and `resp.text()` because they're awaitable coroutines. The request/response cycle would otherwise be the long-tailed, time-hogging portion of the application, but with async IO, `fetch_html()` lets the event loop work on other readily available jobs such as parsing and writing URLs that have already been fetched.

Next in the chain of coroutines comes `parse()`, which waits on `fetch_html()` for a given URL, and then extracts all of the href tags from that page's HTML, making sure that each is valid and formatting it as an absolute path.

Admittedly, the second portion of `parse()` is blocking, but it consists of a quick regex match and ensuring that the links discovered are made into absolute paths.

In this specific case, this synchronous code should be quick and inconspicuous. But just remember that any line within a given coroutine will block other coroutines unless that line uses `yield`, `await`, or `return`. If the parsing was a more intensive process, you might want to consider running this portion in its own process with `loop.run_in_executor()`.

Next, the coroutine `write()` takes a file object and a single URL, and waits on `parse()` to return a set of the parsed URLs, writing each to the file asynchronously along with its source URL through use of `aiofiles`, a package for async file IO.

Lastly, `bulk_crawl_and_write()` serves as the main entry point into the script's chain of coroutines. It uses a single session, and a task is created for each URL that is ultimately read from `urls.txt`.

Here are a few additional points that deserve mention:

- The default `ClientSession` has an [adapter](#) with a maximum of 100 open connections. To change that, pass an instance of `asyncio.connector.TCPConnector` to `ClientSession`. You can also specify limits on a per-host basis.
- You can specify max [timeouts](#) for both the session as a whole and for individual requests.
- This script also uses `async with`, which works with an [asynchronous context manager](#). I haven't devoted a whole section to this concept because the transition from synchronous to asynchronous context managers is fairly straightforward. The latter has to define `__aenter__()` and `__aexit__()` rather than `__exit__()` and `__enter__()`. As you might expect, `async with` can only be used inside a coroutine function declared with `async def`.

If you'd like to explore a bit more, the [companion files](#) for this tutorial up at GitHub have comments and docstrings attached as well.

Here's the execution in all of its glory, as `areq.py` gets, parses, and saves results for 9 URLs in under a second:

Shell

```
$ python3 areq.py
21:33:22 DEBUG:asyncio: Using selector: KqueueSelector
```

```
21:33:22 INFO:areq: Got response [200] for URL: https://www.mediamatters.org/
21:33:22 INFO:areq: Found 115 links for https://www.mediamatters.org/
21:33:22 INFO:areq: Got response [200] for URL: https://www.nytimes.com/guides/
21:33:22 INFO:areq: Got response [200] for URL: https://www.politico.com/tipsheets/morning-money
21:33:22 INFO:areq: Got response [200] for URL: https://www.ietf.org/rfc/rfc2616.txt
21:33:22 ERROR:areq: aiohttp exception for https://docs.python.org/3/this-url-will-404.html [404]: N
21:33:22 INFO:areq: Found 120 links for https://www.nytimes.com/guides/
21:33:22 INFO:areq: Found 143 links for https://www.politico.com/tipsheets/morning-money
21:33:22 INFO:areq: Wrote results for source URL: https://www.mediamatters.org/
21:33:22 INFO:areq: Found 0 links for https://www.ietf.org/rfc/rfc2616.txt
21:33:22 INFO:areq: Got response [200] for URL: https://1.1.1.1/
21:33:22 INFO:areq: Wrote results for source URL: https://www.nytimes.com/guides/
21:33:22 INFO:areq: Wrote results for source URL: https://www.politico.com/tipsheets/morning-money
21:33:22 INFO:areq: Got response [200] for URL: https://www.bloomberg.com/markets/economics
21:33:22 INFO:areq: Found 3 links for https://www.bloomberg.com/markets/economics
21:33:22 INFO:areq: Wrote results for source URL: https://www.bloomberg.com/markets/economics
21:33:23 INFO:areq: Found 36 links for https://1.1.1.1/
21:33:23 INFO:areq: Got response [200] for URL: https://regex101.com/
21:33:23 INFO:areq: Found 23 links for https://regex101.com/
21:33:23 INFO:areq: Wrote results for source URL: https://regex101.com/
21:33:23 INFO:areq: Wrote results for source URL: https://1.1.1.1/
```

That's not too shabby! As a sanity check, you can check the line-count on the output. In my case, it's 626, though keep in mind this may fluctuate:

Shell

```
$ wc -l foundurls.txt
626 foundurls.txt

$ head -n 3 foundurls.txt
source_url    parsed_url
https://www.bloomberg.com/markets/economics https://www.bloomberg.com/feedback
https://www.bloomberg.com/markets/economics https://www.bloomberg.com/notices/tos
```

Next Steps: If you'd like to up the ante, make this webcrawler recursive. You can use [aio-redis](#) to keep track of which URLs have been crawled within the tree to avoid requesting them twice, and connect links with Python's `networkx` library.

Remember to be nice. Sending 1000 concurrent requests to a small, unsuspecting website is bad, bad, bad. There are ways to limit how many concurrent requests you're making in one batch, such as in using the `semaphore` objects of `asyncio` or using a pattern [like this one](#). If you don't heed this warning, you may get a massive batch of `TimeoutError` exceptions and only end up hurting your own program.

Async IO in Context

Now that you've seen a healthy dose of code, let's step back for a minute and consider when async IO is an ideal option and how you can make the comparison to arrive at that conclusion or otherwise choose a different model of concurrency.

When and Why Is Async IO the Right Choice?

This tutorial is no place for an extended treatise on async IO versus threading versus multiprocessing. However, it's useful to have an idea of when async IO is probably the best candidate of the three.

The battle over async IO versus multiprocessing is not really a battle at all. In fact, they can be [used in concert](#). If you have multiple, fairly uniform CPU-bound tasks (a great example is a `grid search` in libraries such as `scikit-learn` or `keras`), multiprocessing should be an obvious choice.

Simply putting `async` before every function is a bad idea if all of the functions use blocking calls. (This can actually slow down your code.) But as mentioned previously, there are places where async IO and multiprocessing can [live in harmony](#).

The contest between async IO and threading is a little bit more direct. I mentioned in the introduction that "threading is hard." The full story is that, even in cases where threading seems easy to implement, it can still lead to infamous impossible-to-trace bugs due to race conditions and memory usage, among other things.

Threading also tends to scale less elegantly than async IO, because threads are a system resource with a finite availability. Creating thousands of threads will fail on many machines, and I don't recommend trying it in the first place. Creating thousands of async IO tasks is completely feasible.

Async IO shines when you have multiple IO-bound tasks where the tasks would otherwise be dominated by blocking IO-bound wait time, such as:

- Network IO, whether your program is the server or the client side
- Serverless designs, such as a peer-to-peer, multi-user network like a group chatroom
- Read/write operations where you want to mimic a “fire-and-forget” style but worry less about holding a lock on whatever you’re reading and writing to

The biggest reason not to use it is that `await` only supports a specific set of objects that define a specific set of methods. If you want to do async read operations with a certain DBMS, you’ll need to find not just a Python wrapper for that DBMS, but one that supports the `async/await` syntax. Coroutines that contain synchronous calls block other coroutines and tasks from running.

For a shortlist of libraries that work with `async/await`, see the [list](#) at the end of this tutorial.

Async IO It Is, but Which One?

This tutorial focuses on async IO, the `async/await` syntax, and using `asyncio` for event-loop management and specifying tasks. `asyncio` certainly isn’t the only async IO library out there. This observation from Nathaniel J. Smith says a lot:

[In] a few years, `asyncio` might find itself relegated to becoming one of those stdlib libraries that savvy developers avoid, like `urllib2`.

...

What I’m arguing, in effect, is that `asyncio` is a victim of its own success: when it was designed, it used the best approach possible; but since then, work inspired by `asyncio` – like the addition of `async/await` – has shifted the landscape so that we can do even better, and now `asyncio` is hamstrung by its earlier commitments.

[\(Source\)](#)

To that end, a few big-name alternatives that do what `asyncio` does, albeit with different APIs and different approaches, are `curio` and `trio`. Personally, I think that if you’re building a moderately sized, straightforward program, just using `asyncio` is plenty sufficient and understandable, and lets you avoid adding yet another large dependency outside of Python’s standard library.

But by all means, check out `curio` and `trio`, and you might find that they get the same thing done in a way that’s more intuitive for you as the user. Many of the package-agnostic concepts presented here should permeate to alternative async IO packages as well.

Odds and Ends

In these next few sections, you’ll cover some miscellaneous parts of `asyncio` and `async/await` that haven’t fit neatly into the tutorial thus far, but are still important for building and understanding a full program.

Other Top-Level `asyncio` Functions

In addition to `asyncio.run()`, you’ve seen a few other package-level functions such as `asyncio.create_task()` and `asyncio.gather()`.

You can use `create_task()` to schedule the execution of a coroutine object, followed by `asyncio.run()`:

```
Python >>>
>>> import asyncio
>>> async def coro(seq) -> list:
...     """IO' wait time is proportional to the max element."""
...     await asyncio.sleep(max(seq))
```

```

...
    return list(reversed(seq))
...
>>> async def main():
...     # This is a bit redundant in the case of one task
...     # We could use `await coro([3, 2, 1])` on its own
...     t = asyncio.create_task(coro([3, 2, 1])) # Python 3.7+
...     await t
...     print(f't: type {type(t)}')
...     print(f't done: {t.done()}')
...
>>> t = asyncio.run(main())
t: type <class '_asyncio.Task'>
t done: True

```

There's a subtlety to this pattern: if you don't `await t` within `main()`, it may finish before `main()` itself signals that it is complete. Because `asyncio.run(main())` [calls `loop.run_until_complete\(main\(\)\)`](#), the event loop is only concerned (without `await t` present) that `main()` is done, not that the tasks that get created within `main()` are done. Without `await t`, the loop's other tasks [will be cancelled](#), possibly before they are completed. If you need to get a list of currently pending tasks, you can use `asyncio.Task.all_tasks()`.

Note: `asyncio.create_task()` was introduced in Python 3.7. In Python 3.6 or lower, use `asyncio.ensure_future()` in place of `create_task()`.

Separately, there's `asyncio.gather()`. While it doesn't do anything tremendously special, `gather()` is meant to neatly put a collection of coroutines (futures) into a single future. As a result, it returns a single future object, and, if you `await asyncio.gather()` and specify multiple tasks or coroutines, you're waiting for all of them to be completed. (This somewhat parallels `queue.join()` from our earlier example.) The result of `gather()` will be a list of the results across the inputs:

```

Python >>>

>>> import time
>>> async def main():
...     t = asyncio.create_task(coro([3, 2, 1]))
...     t2 = asyncio.create_task(coro([10, 5, 0])) # Python 3.7+
...     print('Start:', time.strftime('%X'))
...     a = await asyncio.gather(t, t2)
...     print('End:', time.strftime('%X')) # Should be 10 seconds
...     print(f'Both tasks done: {all((t.done(), t2.done()))}')
...     return a
...
>>> a = asyncio.run(main())
Start: 16:20:11
End: 16:20:21
Both tasks done: True
>>> a
[[1, 2, 3], [0, 5, 10]]

```

You probably noticed that `gather()` waits on the entire result set of the Futures or coroutines that you pass it. Alternatively, you can loop over `asyncio.as_completed()` to get tasks as they are completed, in the order of

completion. The function returns an iterator that yields tasks as they finish. Below, the result of `coro([3, 2, 1])` will be available before `coro([10, 5, 0])` is complete, which is not the case with `gather()`:

```
Python >>> >>>
>>> async def main():
...     t = asyncio.create_task(coro([3, 2, 1]))
...     t2 = asyncio.create_task(coro([10, 5, 0]))
...     print('Start:', time.strftime('%X'))
...     for res in asyncio.as_completed((t, t2)):
...         compl = await res
...         print(f'res: {compl} completed at {time.strftime("%X")}')
...     print('End:', time.strftime('%X'))
...     print(f'Both tasks done: {all((t.done(), t2.done()))}')
...
>>> a = asyncio.run(main())
Start: 09:49:07
res: [1, 2, 3] completed at 09:49:10
res: [0, 5, 10] completed at 09:49:17
End: 09:49:17
Both tasks done: True
```

Lastly, you may also see `asyncio.ensure_future()`. You should rarely need it, because it's a lower-level plumbing API and largely replaced by `create_task()`, which was introduced later.

The Precedence of `await`

While they behave somewhat similarly, the `await` keyword has significantly higher precedence than `yield`. This means that, because it is more tightly bound, there are a number of instances where you'd need parentheses in a `yield from` statement that are not required in an analogous `await` statement. For more information, see [examples of await expressions](#) from PEP 492.

Conclusion

You're now equipped to use `async/await` and the libraries built off of it. Here's a recap of what you've covered:

- Asynchronous IO as a language-agnostic model and a way to effect concurrency by letting coroutines indirectly communicate with each other
- The specifics of Python's new `async` and `await` keywords, used to mark and define coroutines
- `asyncio`, the Python package that provides the API to run and manage coroutines

Resources

Python Version Specifics

Async IO in Python has evolved swiftly, and it can be hard to keep track of what came when. Here's a list of Python minor-version changes and introductions related to `asyncio`:

- 3.3: The `yield from` expression allows for generator delegation.
- 3.4: `asyncio` was introduced in the Python standard library with provisional API status.
- 3.5: `async` and `await` became a part of the Python grammar, used to signify and wait on coroutines. They were not yet reserved keywords. (You could still define functions or variables named `async` and `await`.)
- 3.6: Asynchronous generators and asynchronous comprehensions were introduced. The API of `asyncio` was declared stable rather than provisional.
- 3.7: `async` and `await` became reserved keywords. (They cannot be used as identifiers.) They are intended to replace the `asyncio.coroutine()` decorator. `asyncio.run()` was introduced to the `asyncio` package, among a [bunch of other features](#).

If you want to be safe (and be able to use `asyncio.run()`), go with Python 3.7 or above to get the full set of features.

Articles

Here's a curated list of additional resources:

- Real Python: [Speed up your Python Program with Concurrency](#)
- Real Python: [What is the Python Global Interpreter Lock?](#)
- CPython: The `asyncio` package [source](#)
- Python docs: [Data model > Coroutines](#)
- TalkPython: [Async Techniques and Examples in Python](#)
- Brett Cannon: [How the Heck Does Async-Await Work in Python 3.5?](#)
- PYMOTW: [asyncio](#)
- A. Jesse Jiryu Davis and Guido van Rossum: [A Web Crawler With asyncio Coroutines](#)
- Andy Pearce: [The State of Python Coroutines: `yield from`](#)
- Nathaniel J. Smith: [Some Thoughts on Asynchronous API Design in a Post-`async/await` World](#)
- Armin Ronacher: [I don't understand Python's Asyncio](#)
- Andy Balaam: [series on asyncio](#) (4 posts)
- Stack Overflow: [Python `asyncio.semaphore` in `async-await` function](#)
- Yeray Diaz:
 - [AsyncIO for the Working Python Developer](#)
 - [Asyncio Coroutine Patterns: Beyond `await`](#)

A few Python *What's New* sections explain the motivation behind language changes in more detail:

- [What's New in Python 3.3](#) (`yield from` and PEP 380)
- [What's New in Python 3.6](#) (PEP 525 & 530)

From David Beazley:

- [Generator: Tricks for Systems Programmers](#)
- [A Curious Course on Coroutines and Concurrency](#)
- [Generators: The Final Frontier](#)

YouTube talks:

- [John Reese - Thinking Outside the GIL with AsyncIO and Multiprocessing - PyCon 2018](#)
- [Keynote David Beazley - Topics of Interest \(Python Asyncio\)](#)
- [David Beazley - Python Concurrency From the Ground Up: LIVE! - PyCon 2015](#)
- [Raymond Hettinger, Keynote on Concurrency, PyBay 2017](#)
- [Thinking about Concurrency, Raymond Hettinger, Python core developer](#)
- [Miguel Grinberg Asynchronous Python for the Complete Beginner PyCon 2017](#)
- [Yury Selivanov asyncio and `asyncio` in Python 3.6 and beyond PyCon 2017](#)
- [Fear and Awaiting in Async: A Savage Journey to the Heart of the Coroutine Dream](#)
- [What Is Async, How Does It Work, and When Should I Use It? \(PyCon APAC 2014\)](#)

Related PEPs

PEP	Date Created
PEP 342 – Coroutines via Enhanced Generators	2005-05
PEP 380 – Syntax for Delegating to a Subgenerator	2009-02
PEP 3153 – Asynchronous IO support	2011-05
PEP 3156 – Asynchronous IO Support Rebooted: the “<code>asyncio</code>” Module	2012-12
PEP 492 – Coroutines with <code>async</code> and <code>await</code> syntax	2015-04

Libraries That Work With `async/await`

From [aio-libs](#):

- [aiohttp](#): Asynchronous HTTP client/server framework
- [aioredis](#): Async IO Redis support
- [aiopg](#): Async IO PostgreSQL support
- [aiomcache](#): Async IO memcached client
- [aiokafka](#): Async IO Kafka client
- [aiozmq](#): Async IO ZeroMQ support
- [aiojobs](#): Jobs scheduler for managing background tasks
- [async_lru](#): Simple LRU cache for async IO

From [magicstack](#):

- [uvloop](#): Ultra fast async IO event loop
- [asyncpg](#): (Also very fast) async IO PostgreSQL support

From other hosts:

- [trio](#): Friendlier asyncio intended to showcase a radically simpler design
- [aiofiles](#): Async file IO
- [asks](#): Async requests-like http library
- [asyncio-redis](#): Async IO Redis support
- [aioprocessing](#): Integrates multiprocessing module with asyncio
- [umongo](#): Async IO MongoDB client
- [unsync](#): Unsynchronize asyncio
- [aiostream](#): Like itertools, but async

 [Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Hands-on Python 3 Concurrency With the `asyncio` Module](#)

About Brad Solomon

Brad is a software engineer and a member of the Real Python Tutorial Team.

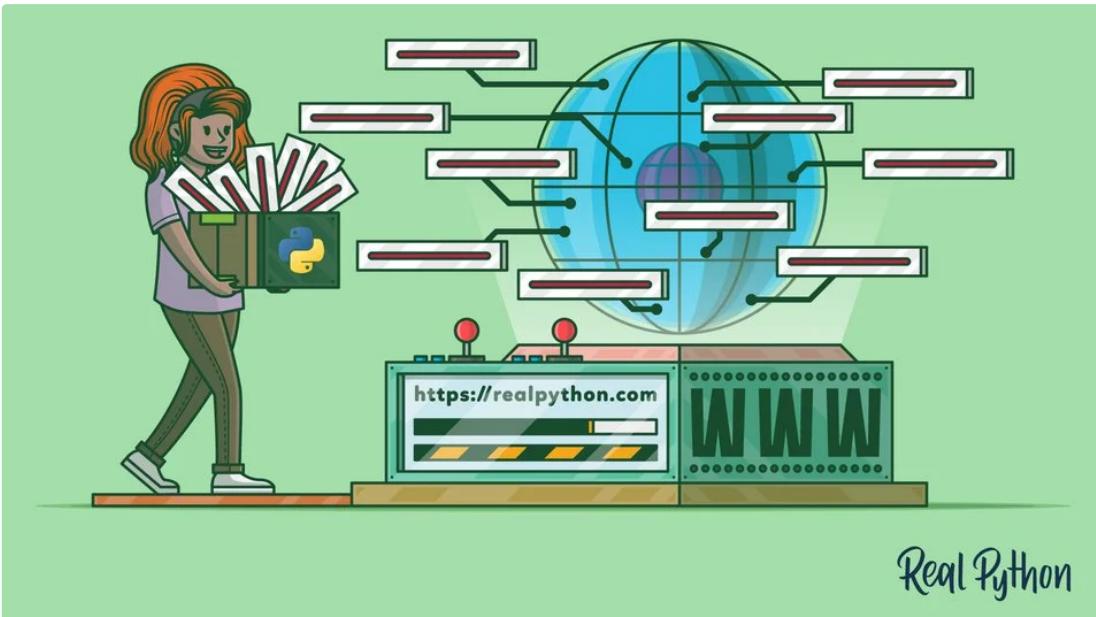
[» More about Brad](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)

Recommended Video Course: [Hands-on Python 3 Concurrency With the asyncio Module](#)



Real Python

A Practical Introduction to Web Scraping in Python

by [David Amos](#) ⌂ Aug 17, 2020 🗣 49 Comments 📁 [intermediate](#) [web-scraping](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Scrape and Parse Text From Websites](#)
 - [Your First Web Scraper](#)
 - [Extract Text From HTML With String Methods](#)
 - [A Primer on Regular Expressions](#)
 - [Extract Text From HTML With Regular Expressions](#)
 - [Check Your Understanding](#)
- [Use an HTML Parser for Web Scraping in Python](#)
 - [Install BeautifulSoup](#)
 - [Create a BeautifulSoup Object](#)
 - [Use a BeautifulSoup Object](#)
 - [Check Your Understanding](#)
- [Interact With HTML Forms](#)
 - [Install MechanicalSoup](#)
 - [Create a Browser Object](#)
 - [Submit a Form With MechanicalSoup](#)
 - [Check Your Understanding](#)
- [Interact With Websites in Real Time](#)
- [Conclusion](#)
- [Additional Resources](#)

Leverage Python's Third-Party Package Ecosystem and Get More Done, Faster

[realpython.com](#)



Web scraping is the process of collecting and parsing raw data from the Web, and the Python community has come up with some pretty powerful web scraping tools.

The Internet hosts perhaps the greatest source of information—and misinformation—on the planet. Many disciplines, such as data science, business intelligence, and investigative reporting, can benefit enormously from

collecting and analyzing data from websites.

In this tutorial, you'll learn how to:

- Parse website data using **string methods** and **regular expressions**
- Parse website data using an **HTML parser**
- Interact with **forms** and other website components

Note: This tutorial is adapted from the chapter “Interacting With the Web” in *Python Basics: A Practical Introduction to Python 3*.

The book uses Python’s built-in **IDLE** editor to create and edit Python files and interact with the Python shell, so you will see occasional references to IDLE throughout this tutorial. However, you should have no problems running the example code from the editor and environment of your choice.

Free Bonus: [Click here to get our free Python Cheat Sheet](#) that shows you the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Scrape and Parse Text From Websites

Collecting data from websites using an automated process is known as web scraping. Some websites explicitly forbid users from scraping their data with automated tools like the ones you’ll create in this tutorial. Websites do this for two possible reasons:

1. The site has a good reason to protect its data. For instance, Google Maps doesn’t let you request too many results too quickly.
2. Making many repeated requests to a website’s server may use up bandwidth, slowing down the website for other users and potentially overloading the server such that the website stops responding entirely.

Important: Before using your Python skills for web scraping, you should always check your target website’s acceptable use policy to see if accessing the website with automated tools is a violation of its terms of use. Legally, web scraping against the wishes of a website is very much a gray area.

Please be aware that the following techniques [may be illegal](#) when used on websites that prohibit web scraping.

Let’s start by grabbing all the HTML code from a single web page. You’ll use a page on *Real Python* that’s been set up for use with this tutorial.

Your First Web Scraper

One useful package for web scraping that you can find in Python’s [standard library](#) is `urllib`, which contains tools for working with URLs. In particular, the `urllib.request` module contains a function called `urlopen()` that can be used to open a URL within a program.

In IDLE’s interactive window, type the following to import `urlopen()`:

```
Python >>>
>>> from urllib.request import urlopen
```

The web page that we’ll open is at the following URL:

```
Python >>>
>>> url = "http://olympus.realpython.org/profiles/aphrodite"
```

To open the web page, pass `url` to `urlopen()`:

Python

>>>

```
>>> page = urlopen(url)
```

urlopen() returns an `HTTPResponse` object:

Python

>>>

```
>>> page
<http.client.HTTPResponse object at 0x105fef820>
```

To extract the HTML from the page, first use the `HTTPResponse` object's `.read()` method, which returns a sequence of bytes. Then use `.decode()` to decode the bytes to a string using [UTF-8](#):

Python

>>>

```
>>> html_bytes = page.read()
>>> html = html_bytes.decode("utf-8")
```

Now you can print the HTML to see the contents of the web page:

Python

>>>

```
>>> print(html)
<html>
<head>
<title>Profile: Aphrodite</title>
</head>
<body bgcolor="yellow">
<center>
<br><br>

<h2>Name: Aphrodite</h2>
<br><br>
Favorite animal: Dove
<br><br>
Favorite color: Red
<br><br>
Hometown: Mount Olympus
</center>
</body>
</html>
```

Once you have the HTML as text, you can extract information from it in a couple of different ways.

Extract Text From HTML With String Methods

One way to extract information from a web page's HTML is to use [string methods](#). For instance, you can use `.find()` to search through the text of the HTML for the `<title>` tags and extract the title of the web page.

Let's extract the title of the web page you requested in the previous example. If you know the index of the first character of the title and the first character of the closing `</title>` tag, then you can use a [string slice](#) to extract the title.

Since `.find()` returns the index of the first occurrence of a substring, you can get the index of the opening `<title>` tag by passing the string "`<title>`" to `.find()`:

Python

>>>

```
>>> title_index = html.find("<title>")
>>> title_index
14
```

You don't want the index of the `<title>` tag, though. You want the index of the title itself. To get the index of the first letter in the title, you can add the length of the string "`<title>`" to `title_index`:

Python

>>>

```
>>> start_index = title_index + len("<title>")
>>> start_index
21
```

Now get the index of the closing `</title>` tag by passing the string `"</title>"` to `.find()`:

```
Python >>>
>>> end_index = html.find("</title>")
>>> end_index
39
```

Finally, you can extract the title by slicing the `html` string:

```
Python >>>
>>> title = html[start_index:end_index]
>>> title
'Profile: Aphrodite'
```

Real-world HTML can be much more complicated and far less predictable than the HTML on the Aphrodite profile page. Here's [another profile page](#) with some messier HTML that you can scrape:

```
Python >>>
>>> url = "http://olympus.realpython.org/profiles/poseidon"
```

Try extracting the title from this new URL using the same method as the previous example:

```
Python >>>
>>> url = "http://olympus.realpython.org/profiles/poseidon"
>>> page = urlopen(url)
>>> html = page.read().decode("utf-8")
>>> start_index = html.find("<title>") + len("<title>")
>>> end_index = html.find("</title>")
>>> title = html[start_index:end_index]
>>> title
'\n<head>\n<title>Profile: Poseidon'
```

Whoops! There's a bit of HTML mixed in with the title. Why's that?

The HTML for the `/profiles/poseidon` page looks similar to the `/profiles/aphrodite` page, but there's a small difference. The opening `<title>` tag has an extra space before the closing angle bracket (`>`), rendering it as `<title>`.

`html.find("<title>")` returns `-1` because the exact substring `"<title>"` doesn't exist. When `-1` is added to `len("<title>")`, which is `7`, the `start_index` variable is assigned the value `6`.

The character at index `6` of the string `html` is a newline character (`\n`) right before the opening angle bracket (`<`) of the `<head>` tag. This means that `html[start_index:end_index]` returns all the HTML starting with that newline and ending just before the `</title>` tag.

These sorts of problems can occur in countless unpredictable ways. You need a more reliable way to extract text from HTML.

A Primer on Regular Expressions

Regular expressions—or **regexes** for short—are patterns that can be used to search for text within a string. Python supports regular expressions through the standard library's `re` module.

Note: Regular expressions aren't particular to Python. They're a general programming concept and can be used with any programming language.

To work with regular expressions, the first thing you need to do is import the `re` module:

Python

>>>

```
>>> import re
```

Regular expressions use special characters called **metacharacters** to denote different patterns. For instance, the asterisk character (*) stands for zero or more of whatever comes just before the asterisk.

In the following example, you use `.findall()` to find any text within a string that matches a given regular expression:

Python

>>>

```
>>> re.findall("ab*c", "ac")  
['ac']
```

The first argument of `re.findall()` is the regular expression that you want to match, and the second argument is the string to test. In the above example, you search for the pattern "ab*c" in the string "ac".

The regular expression "ab*c" matches any part of the string that begins with an "a", ends with a "c", and has zero or more instances of "b" between the two. `re.findall()` returns a list of all matches. The string "ac" matches this pattern, so it's returned in the list.

Here's the same pattern applied to different strings:

Python

>>>

```
>>> re.findall("ab*c", "abcd")  
['abc']  
  
>>> re.findall("ab*c", "acc")  
['ac']  
  
>>> re.findall("ab*c", "abcac")  
['abc', 'ac']  
  
>>> re.findall("ab*c", "abdc")  
[]
```

Notice that if no match is found, then `.findall()` returns an empty list.

Pattern matching is case sensitive. If you want to match this pattern regardless of the case, then you can pass a third argument with the value `re.IGNORECASE`:

Python

>>>

```
>>> re.findall("ab*c", "ABC")  
[]  
  
>>> re.findall("ab*c", "ABC", re.IGNORECASE)  
['ABC']
```

You can use a period (.) to stand for any single character in a regular expression. For instance, you could find all the strings that contain the letters "a" and "c" separated by a single character as follows:

Python

>>>

```
>>> re.findall("a.c", "abc")  
['abc']  
  
>>> re.findall("a.c", "abbc")  
[]  
  
>>> re.findall("a.c", "ac")  
[]  
  
>>> re.findall("a.c", "acc")  
['acc']
```

The pattern `.*` inside a regular expression stands for any character repeated any number of times. For instance, "`a.*c`" can be used to find every substring that starts with "a" and ends with "c", regardless of which letter—or

letters—are in between:

```
Python >>>
>>> re.findall("a.*c", "abc")
['abc']

>>> re.findall("a.*c", "abbc")
['abbc']

>>> re.findall("a.*c", "ac")
['ac']

>>> re.findall("a.*c", "acc")
['acc']
```

Often, you use `re.search()` to search for a particular pattern inside a string. This function is somewhat more complicated than `re.findall()` because it returns an object called a `MatchObject` that stores different groups of data. This is because there might be matches inside other matches, and `re.search()` returns every possible result.

The details of the `MatchObject` are irrelevant here. For now, just know that calling `.group()` on a `MatchObject` will return the first and most inclusive result, which in most cases is just what you want:

```
Python >>>
>>> match_results = re.search("ab*c", "ABC", re.IGNORECASE)
>>> match_results.group()
'ABC'
```

There's one more function in the `re` module that's useful for parsing out text. `re.sub()`, which is short for *substitute*, allows you to replace text in a string that matches a regular expression with new text. It behaves sort of like the `.replace()` string method.

The arguments passed to `re.sub()` are the regular expression, followed by the replacement text, followed by the string. Here's an example:

```
Python >>>
>>> string = "Everything is <replaced> if it's in <tags>."
>>> string = re.sub("<.*>", "ELEPHANTS", string)
>>> string
'Everything is ELEPHANTS.'
```

Perhaps that wasn't quite what you expected to happen.

`re.sub()` uses the regular expression "`<.*>`" to find and replace everything between the first `<` and last `>`, which spans from the beginning of `<replaced>` to the end of `<tags>`. This is because Python's regular expressions are **greedy**, meaning they try to find the longest possible match when characters like `*` are used.

Alternatively, you can use the non-greedy matching pattern `*?`, which works the same way as `*` except that it matches the shortest possible string of text:

```
Python >>>
>>> string = "Everything is <replaced> if it's in <tags>."
>>> string = re.sub("<.*?>", "ELEPHANTS", string)
>>> string
"Everything is ELEPHANTS if it's in ELEPHANTS."
```

This time, `re.sub()` finds two matches, `<replaced>` and `<tags>`, and substitutes the string "ELEPHANTS" for both matches.

Extract Text From HTML With Regular Expressions

Armed with all this knowledge, let's now try to parse out the title from a [new profile page](#), which includes this rather carelessly written line of HTML:

HTML

```
<TITLE>Profile: Dionysus</TITLE> / >
```

The `.find()` method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, you can handle this code quickly and efficiently:

Python

```
import re
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")

pattern = "<title.*?>.*?</title.*?>"
match_results = re.search(pattern, html, re.IGNORECASE)
title = match_results.group()
title = re.sub("<.*?>", "", title) # Remove HTML tags

print(title)
```

Let's take a closer look at the first regular expression in the pattern string by breaking it down into three parts:

1. `<title.*?>` matches the opening `<TITLE>` tag in `html`. The `<title` part of the pattern matches with `<TITLE` because `re.search()` is called with `re.IGNORECASE`, and `.*?>` matches any text after `<TITLE` up to the first instance of `>`.
2. `.*?` non-greedily matches all text after the opening `<TITLE>`, stopping at the first match for `</title.*?>`.
3. `</title.*?>` differs from the first pattern only in its use of the `/` character, so it matches the closing `</title>` tag in `html`.

The second regular expression, the string "`<.*?>`", also uses the non-greedy `.*?` to match all the HTML tags in the `title` string. By replacing any matches with `" "`, `re.sub()` removes all the tags and returns only the text.

Note: Web scraping in Python or any other language can be tedious. No two websites are organized the same way, and HTML is often messy. Moreover, websites change over time. Web scrapers that work today are not guaranteed to work next year—or next week, for that matter!

Regular expressions are a powerful tool when used correctly. This introduction barely scratches the surface. For more about regular expressions and how to use them, check out the two-part series [Regular Expressions: Regexes in Python](#).

Check Your Understanding

Expand the block below to check your understanding.

You can expand the block below to see a solution.

Solution: Scrape Data From a Website

Show/Hide

When you're ready, you can move on to the next section.

Use an HTML Parser for Web Scraping in Python

Although regular expressions are great for pattern matching in general, sometimes it's easier to use an HTML parser that's explicitly designed for parsing out HTML pages. There are many Python tools written for this purpose, but the [Beautiful Soup](#) library is a good one to start with.

Install BeautifulSoup

To install BeautifulSoup, you can run the following in your terminal:

Shell

```
$ python3 -m pip install beautifulsoup4
```

Run `pip show` to see the details of the package you just installed:

Shell

```
$ python3 -m pip show beautifulsoup4
Name: beautifulsoup4
Version: 4.9.1
Summary: Screen-scraping library
Home-page: http://www.crummy.com/software/BeautifulSoup/bs4/
Author: Leonard Richardson
Author-email: leonardr@segfault.org
License: MIT
Location: c:\realpython\venv\lib\site-packages
Requires:
Required-by:
```

In particular, notice that the latest version at the time of writing was 4.9.1.

Create a BeautifulSoup Object

Type the following program into a new editor window:

Python

```
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")
soup = BeautifulSoup(html, "html.parser")
```

This program does three things:

1. Opens the URL `http://olympus.realpython.org/profiles/dionysus` using `urlopen()` from the `urllib.request` module
2. Reads the HTML from the page as a string and assigns it to the `html` variable
3. Creates a `BeautifulSoup` object and assigns it to the `soup` variable

The `BeautifulSoup` object assigned to `soup` is created with two arguments. The first argument is the HTML to be parsed, and the second argument, the string `"html.parser"`, tells the object which parser to use behind the scenes. `"html.parser"` represents Python's built-in HTML parser.

Use a BeautifulSoup Object

Save and run the above program. When it's finished running, you can use the `soup` variable in the interactive window to parse the content of `html` in various ways.

For example, `BeautifulSoup` objects have a `.get_text()` method that can be used to extract all the text from the

document and automatically remove any HTML tags.

Type the following code into IDLE's interactive window:

```
Python >>> Python
>>> print(soup.get_text())

Profile: Dionysus

Name: Dionysus
Hometown: Mount Olympus
Favorite animal: Leopard
Favorite Color: Wine
```

There are a lot of blank lines in this output. These are the result of newline characters in the HTML document's text. You can remove them with the string `.replace()` method if you need to.

Often, you need to get only specific text from an HTML document. Using Beautiful Soup first to extract the text and then using the `.find()` string method is *sometimes* easier than working with regular expressions.

However, sometimes the HTML tags themselves are the elements that point out the data you want to retrieve. For instance, perhaps you want to retrieve the URLs for all the images on the page. These links are contained in the `src` attribute of `` HTML tags.

In this case, you can use `find_all()` to return a list of all instances of that particular tag:

```
Python >>> Python
>>> soup.find_all("img")
[, ]
```

This returns a list of all `` tags in the HTML document. The objects in the list look like they might be strings representing the tags, but they're actually instances of the `Tag` object provided by Beautiful Soup. `Tag` objects provide a simple interface for working with the information they contain.

Let's explore this a little by first unpacking the `Tag` objects from the list:

```
Python >>> Python
>>> image1, image2 = soup.find_all("img")
```

Each `Tag` object has a `.name` property that returns a string containing the HTML tag type:

```
Python >>> Python
>>> image1.name
'img'
```

You can access the HTML attributes of the `Tag` object by putting their name between square brackets, just as if the attributes were keys in a dictionary.

For example, the `` tag has a single attribute, `src`, with the value `"/static/dionysus.jpg"`. Likewise, an HTML tag such as the link `` has two attributes, `href` and `target`.

To get the source of the images in the Dionysus profile page, you access the `src` attribute using the dictionary notation mentioned above:

```
Python >>> Python
```

```
>>> image1["src"]
'/static/dionysus.jpg'

>>> image2["src"]
'/static/grapes.png'
```

Certain tags in HTML documents can be accessed by properties of the Tag object. For example, to get the `<title>` tag in a document, you can use the `.title` property:

```
Python >>>
>>> soup.title
<title>Profile: Dionysus</title>
```

If you look at the source of the Dionysus profile by navigating to the [profile page](#), right-clicking on the page, and selecting *View page source*, then you'll notice that the `<title>` tag as written in the document looks like this:

```
HTML
<title >Profile: Dionysus</title/>
```

Beautiful Soup automatically cleans up the tags for you by removing the extra space in the opening tag and the extraneous forward slash (/) in the closing tag.

You can also retrieve just the string between the title tags with the `.string` property of the Tag object:

```
Python >>>
>>> soup.title.string
'Profile: Dionysus'
```

One of the more useful features of Beautiful Soup is the ability to search for specific kinds of tags whose attributes match certain values. For example, if you want to find all the `` tags that have a `src` attribute equal to the value `/static/dionysus.jpg`, then you can provide the following additional argument to `.find_all()`:

```
Python >>>
>>> soup.find_all("img", src="/static/dionysus.jpg")
[]
```

This example is somewhat arbitrary, and the usefulness of this technique may not be apparent from the example. If you spend some time browsing various websites and viewing their page sources, then you'll notice that many websites have extremely complicated HTML structures.

When scraping data from websites with Python, you're often interested in particular parts of the page. By spending some time looking through the HTML document, you can identify tags with unique attributes that you can use to extract the data you need.

Then, instead of relying on complicated regular expressions or using `.find()` to search through the document, you can directly access the particular tag you're interested in and extract the data you need.

In some cases, you may find that Beautiful Soup doesn't offer the functionality you need. The [xml](#) library is somewhat trickier to get started with but offers far more flexibility than Beautiful Soup for parsing HTML documents. You may want to check it out once you're comfortable using Beautiful Soup.

Note: HTML parsers like Beautiful Soup can save you a lot of time and effort when it comes to locating specific data in web pages. However, sometimes HTML is so poorly written and disorganized that even a sophisticated parser like Beautiful Soup can't interpret the HTML tags properly.

In this case, you're often left with using `.find()` and regular expression techniques to try to parse out the information you need.

BeautifulSoup is great for scraping data from a website's HTML, but it doesn't provide any way to work with HTML forms. For example, if you need to search a website for some query and then scrape the results, then BeautifulSoup alone won't get you very far.

Check Your Understanding

Expand the block below to check your understanding.

Exercise: Parse HTML With BeautifulSoup

Show/Hide

You can expand the block below to see a solution:

Solution: Parse HTML With BeautifulSoup

Show/Hide

When you're ready, you can move on to the next section.

Interact With HTML Forms

The `urllib` module you've been working with so far in this tutorial is well suited for requesting the contents of a web page. Sometimes, though, you need to interact with a web page to obtain the content you need. For example, you might need to submit a form or click a button to display hidden content.

Note: This tutorial is adapted from the chapter "Interacting With the Web" in [Python Basics: A Practical Introduction to Python 3](#). If you enjoy what you're reading, then be sure to check out [the rest of the book](#).

The Python standard library doesn't provide a built-in means for working with web pages interactively, but many third-party packages are available from PyPI. Among these, [MechanicalSoup](#) is a popular and relatively straightforward package to use.

In essence, MechanicalSoup installs what's known as a **headless browser**, which is a web browser with no graphical user interface. This browser is controlled programmatically via a Python program.

Install MechanicalSoup

You can install MechanicalSoup with [pip](#) in your terminal:

Shell

```
$ python3 -m pip install MechanicalSoup
```

You can now view some details about the package with `pip show`:

Shell

```
$ python3 -m pip show mechanichalsoup
Name: MechanicalSoup
Version: 0.12.0
Summary: A Python library for automating interaction with websites
Home-page: https://mechanicalsoup.readthedocs.io/
Author: UNKNOWN
Author-email: UNKNOWN
License: MIT
Location: c:\realpython\venv\lib\site-packages
Requires: requests, beautifulsoup4, six, lxml
Required-by:
```

In particular, notice that the latest version at the time of writing was 0.12.0. You'll need to close and restart your IDLE session for MechanicalSoup to load and be recognized after it's been installed.

Create a Browser Object

Type the following into IDLE's interactive window:

```
Python >>>
>>> import mechanicalsoup
>>> browser = mechanicalsoup.Browser()
```

Browser objects represent the headless web browser. You can use them to request a page from the Internet by passing a URL to their `.get()` method:

```
Python >>>
>>> url = "http://olympus.realpython.org/login"
>>> page = browser.get(url)
```

page is a Response object that stores the response from requesting the URL from the browser:

```
Python >>>
>>> page
<Response [200]>
```

The number 200 represents the [status code](#) returned by the request. A status code of 200 means that the request was successful. An unsuccessful request might show a status code of 404 if the URL doesn't exist or 500 if there's a server error when making the request.

MechanicalSoup uses Beautiful Soup to parse the HTML from the request. page has a `.soup` attribute that represents a BeautifulSoup object:

```
Python >>>
>>> type(page.soup)
<class 'bs4.BeautifulSoup'>
```

You can view the HTML by inspecting the `.soup` attribute:

```
Python >>>
>>> page.soup
<html>
<head>
<title>Log In</title>
</head>
<body bgcolor="yellow">
<center>
<br/><br/>
<h2>Please log in to access Mount Olympus:</h2>
<br/><br/>
<form action="/login" method="post" name="login">
  Username: <input name="user" type="text"/><br/>
  Password: <input name="pwd" type="password"/><br/><br/>
  <input type="submit" value="Submit"/>
</form>
</center>
</body>
</html>
```

Notice this page has a `<form>` on it with `<input>` elements for a username and a password.

Submit a Form With MechanicalSoup

Open the [/login](#) page from the previous example in a browser and look at it yourself before moving on. Try typing in a random username and password combination. If you guess incorrectly, then the message "Wrong username or

password!" is displayed at the bottom of the page.

However, if you provide the correct login credentials (username zeus and password ThunderDude), then you're redirected to the [/profiles](#) page.

In the next example, you'll see how to use MechanicalSoup to fill out and submit this form using Python!

The important section of HTML code is the login form—that is, everything inside the `<form>` tags. The `<form>` on this page has the name attribute set to `login`. This form contains two `<input>` elements, one named `user` and the other named `pwd`. The third `<input>` element is the Submit button.

Now that you know the underlying structure of the login form, as well as the credentials needed to log in, let's take a look at a program that fills the form out and submits it.

In a new editor window, type in the following program:

Python

```
import mechanize

# 1
browser = mechanize.Browser()
url = "http://olympus.realpython.org/login"
login_page = browser.get(url)
login_html = login_page.read()

# 2
form = login_html.select("form")[0]
form["input"][0]["value"] = "zeus"
form["input"][1]["value"] = "ThunderDude"

# 3
profiles_page = browser.submit(form, login_page.url)
```

Save the file and press `F5` to run it. You can confirm that you successfully logged in by typing the following into the interactive window:

Python

>>>

```
>>> profiles_page.url
'http://olympus.realpython.org/profiles'
```

Let's break down the above example:

1. You create a `Browser` instance and use it to request the URL `http://olympus.realpython.org/login`. You assign the HTML content of the page to the `login_html` variable using the `.read()` property.
2. `login_html.select("form")` returns a list of all `<form>` elements on the page. Since the page has only one `<form>` element, you can access the form by retrieving the element at index `0` of the list. The next two lines select the username and password inputs and set their value to `"zeus"` and `"ThunderDude"`, respectively.
3. You submit the form with `browser.submit()`. Notice that you pass two arguments to this method, the `form` object and the URL of the `login_page`, which you access via `login_page.url`.

In the interactive window, you confirm that the submission successfully redirected to the `/profiles` page. If something had gone wrong, then the value of `profiles_page.url` would still be `"http://olympus.realpython.org/login"`.

Note: Hackers can use automated programs like the one above to **brute force** logins by rapidly trying many different usernames and passwords until they find a working combination.

Besides this being highly illegal, almost all websites these days lock you out and report your IP address if they see you making too many failed requests, so don't try it!

Now that we have the `profiles_page` variable set, let's see how to programmatically obtain the URL for each link on

the /profiles page.

To do this, you use `.select()` again, this time passing the string "a" to select all the `<a>` anchor elements on the page:

Python

>>>

```
>>> links = profiles_page.soup.select("a")
```

Now you can iterate over each link and print the `href` attribute:

Python

>>>

```
>>> for link in links:
...     address = link["href"]
...     text = link.text
...     print(f"{text}: {address}")
...
Aphrodite: /profiles/aphrodite
Poseidon: /profiles/poseidon
Dionysus: /profiles/dionysus
```

The URLs contained in each `href` attribute are relative URLs, which aren't very helpful if you want to navigate to them later using MechanicalSoup. If you happen to know the full URL, then you can assign the portion needed to construct a full URL.

In this case, the base URL is just `http://olympus.realpython.org`. Then you can concatenate the base URL with the relative URLs found in the `src` attribute:

Python

>>>

```
>>> base_url = "http://olympus.realpython.org"
>>> for link in links:
...     address = base_url + link["href"]
...     text = link.text
...     print(f"{text}: {address}")
...
Aphrodite: http://olympus.realpython.org/profiles/aphrodite
Poseidon: http://olympus.realpython.org/profiles/poseidon
Dionysus: http://olympus.realpython.org/profiles/dionysus
```

You can do a lot with just `.get()`, `.select()`, and `.submit()`. That said, MechanicalSoup is capable of much more. To learn more about MechanicalSoup, check out the [official docs](#).

Check Your Understanding

Expand the block below to check your understanding

Exercise: Submit a Form With MechanicalSoup

Show/Hide

You can expand the block below to see a solution.

Solution: Submit a Form With MechanicalSoup

Show/Hide

When you're ready, you can move on to the next section.

Interact With Websites in Real Time

Sometimes you want to be able to fetch real-time data from a website that offers continually updated information.

In the dark days before you learned Python programming, you had to sit in front of a browser, clicking the Refresh

button to reload the page each time you wanted to check if updated content was available. But now you can automate this process using the `.get()` method of the MechanicalSoup Browser object.

Open your browser of choice and navigate to the URL <http://olympus.realpython.org/dice>. This [/dice](#) page simulates a roll of a six-sided die, updating the result each time you refresh the browser. Below, you'll write a program that repeatedly scrapes the page for a new result.

The first thing you need to do is determine which element on the page contains the result of the die roll. Do this now by right-clicking anywhere on the page and selecting *View page source*. A little more than halfway down the HTML code is an `<h2>` tag that looks like this:

HTML

```
<h2 id="result">4</h2>
```

The text of the `<h2>` tag might be different for you, but this is the page element you need for scraping the result.

Note: For this example, you can easily check that there's only one element on the page with `id="result"`.

Although the `id` attribute is supposed to be unique, in practice you should always check that the element you're interested in is uniquely identified.

Let's start by writing a simple program that opens the [/dice](#) page, scrapes the result, and prints it to the console:

Python

```
import mechanize

browser = mechanize.Browser()
page = browser.get("http://olympus.realpython.org/dice")
tag = page.select("#result")[0]
result = tag.text

print(f"The result of your dice roll is: {result}")
```

This example uses the BeautifulSoup object's `.select()` method to find the element with `id=result`. The string `"#result"` that you pass to `.select()` uses the [CSS ID selector](#) # to indicate that `result` is an `id` value.

To periodically get a new result, you'll need to create a loop that loads the page at each step. So everything below the line `browser = mechanize.Browser()` in the above code needs to go in the body of the loop.

For this example, let's get four rolls of the dice at ten-second intervals. To do that, the last line of your code needs to tell Python to pause running for ten seconds. You can do this with [sleep\(\)](#) from Python's [time module](#). `sleep()` takes a single argument that represents the amount of time to sleep in seconds.

Here's an example that illustrates how `sleep()` works:

Python

```
import time

print("I'm about to wait for five seconds...")
time.sleep(5)
print("Done waiting!")
```

When you run this code, you'll see that the "Done waiting!" message isn't displayed until 5 seconds have passed from when the first `print()` function was executed.

For the die roll example, you'll need to pass the number `10` to `sleep()`. Here's the updated program:

Python

```
import time
import mechanize

browser = mechanize.Browser()

for i in range(4):
    page = browser.get("http://olympus.realpython.org/dice")
    tag = page.select("#result")[-1]
```

```
result = tag.text
print(f"The result of your dice roll is: {result}")
time.sleep(10)
```

When you run the program, you'll immediately see the first result printed to the console. After ten seconds, the second result is displayed, then the third, and finally the fourth. What happens after the fourth result is printed? The program continues running for another ten seconds before it finally stops!

Well, of course it does—that's what you told it to do! But it's kind of a waste of time. You can stop it from doing this by using an [if statement](#) to run `time.sleep()` for only the first three requests:

Python

```
import time
import mechanize

browser = mechanize.Browser()

for i in range(4):
    page = browser.get("http://olympus.realpython.org/dice")
    tag = page.select("#result")[0]
    result = tag.text
    print(f"The result of your dice roll is: {result}")

    # Wait 10 seconds if this isn't the last request
    if i < 3:
        time.sleep(10)
```

With techniques like this, you can scrape data from websites that periodically update their data. However, you should be aware that requesting a page multiple times in rapid succession can be seen as suspicious, or even malicious, use of a website.

Important: Most websites publish a Terms of Use document. You can often find a link to it in the website's footer.

Always read this document before attempting to scrape data from a website. If you can't find the Terms of Use, try to contact the website owner and ask them if they have any policies regarding request volume.

Failure to comply with the Terms of Use could result in your IP being blocked, so be careful and be respectful!

It's even possible to crash a server with an excessive number of requests, so you can imagine that many websites are concerned about the volume of requests to their server! Always check the Terms of Use and be respectful when sending multiple requests to a website.

Conclusion

Although it's possible to parse data from the Web using tools in Python's standard library, there are many tools on PyPI that can help simplify the process.

In this tutorial, you learned how to:

- Request a web page using Python's built-in `urllib` module
- Parse HTML using **Beautiful Soup**
- Interact with web forms using **MechanicalSoup**
- Repeatedly request data from a website to **check for updates**

Writing automated web scraping programs is fun, and the Internet has no shortage of content that can lead to all sorts of exciting projects.

Just remember, not everyone wants you pulling data from their web servers. Always check a website's Terms of Use before you start scraping, and be respectful about how you time your web requests so that you don't flood a server with traffic.

Additional Resources

For more information on web scraping with Python, check out the following resources:

- [Beautiful Soup: Build a Web Scraper With Python](#)
- [API Integration in Python](#)

Note: If you enjoyed what you learned in this sample from [Python Basics: A Practical Introduction to Python 3](#), then be sure to check out [the rest of the book](#).

About David Amos

David is a mathematician by training, a data scientist/Python developer by profession, and a coffee junkie by choice.

[» More about David](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

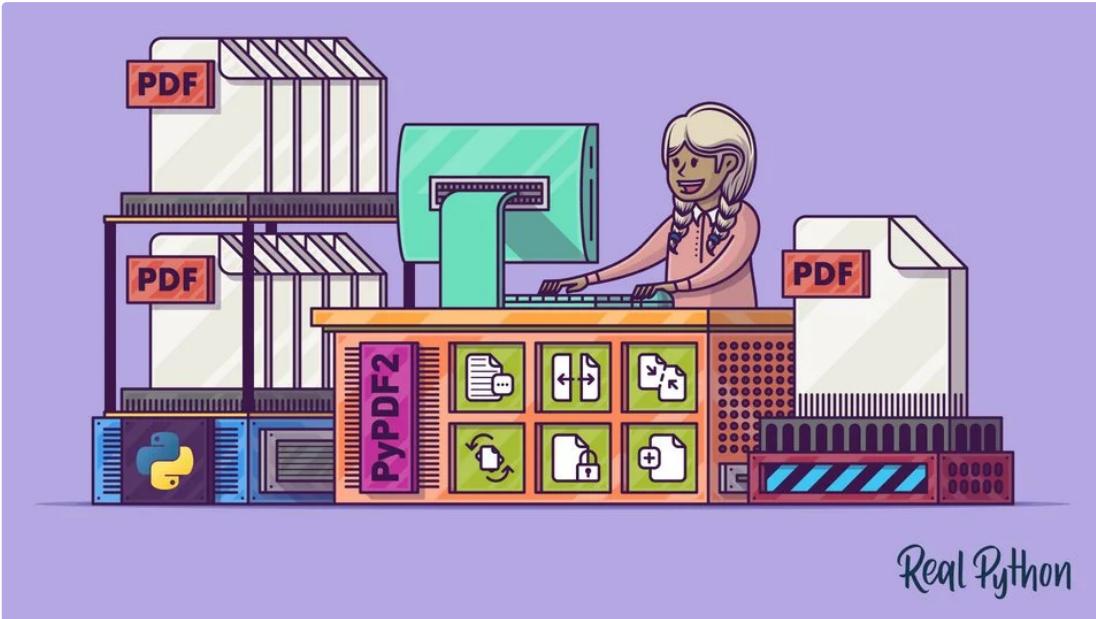
[Aldren](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [web-scraping](#)



Create and Modify PDF Files in Python

by [David Amos](#) ⌂ May 25, 2020 🗣 9 Comments 🏷 [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Extracting Text From a PDF](#)
 - [Opening a PDF File](#)
 - [Extracting Text From a Page](#)
 - [Putting It All Together](#)
 - [Check Your Understanding](#)
- [Extracting Pages From a PDF](#)
 - [Using the PdfFileWriter Class](#)
 - [Extracting a Single Page From a PDF](#)
 - [Extracting Multiple Pages From a PDF](#)
 - [Check Your Understanding](#)
- [Concatenating and Merging PDFs](#)
 - [Using the PdfFileMerger Class](#)
 - [Concatenating PDFs With .append\(\)](#)
 - [Merging PDFs With .merge\(\)](#)
 - [Check Your Understanding](#)
- [Rotating and Cropping PDF Pages](#)
 - [Rotating Pages](#)
 - [Cropping Pages](#)
 - [Check Your Understanding](#)
- [Encrypting and Decrypting PDFs](#)
 - [Encrypting PDFs](#)
 - [Decrypting PDFs](#)
 - [Check Your Understanding](#)
- [Creating a PDF File From Scratch](#)
 - [Installing reportlab](#)
 - [Using the Canvas Class](#)
 - [Setting the Page Size](#)
 - [Setting Font Properties](#)
 - [Check Your Understanding](#)
- [Conclusion: Create and Modify PDF Files in Python](#)



Enhance Python with Redis

[Explore Redis Labs](#)



redislabs
HOME OF REDIS

It's really useful to know how to create and modify PDF files in Python. The **PDF**, or **Portable Document Format**, is one of the most common formats for sharing documents over the Internet. [PDFs](#) can contain text, images, tables, forms, and rich media like videos and animations, all in a single file.

This abundance of content types can make working with PDFs difficult. There are a lot of different kinds of data to decode when opening a PDF file! Fortunately, the Python ecosystem has some great packages for reading, manipulating, and creating PDF files.

In this tutorial, you'll learn how to:

- **Read** text from a PDF
- **Split** a PDF into multiple files
- **Concatenate** and **merge** PDF files
- **Rotate** and **crop** pages in a PDF file
- **Encrypt** and **decrypt** PDF files with passwords
- **Create** a PDF file from scratch

Note: This tutorial is adapted from the chapter “Creating and Modifying PDF Files” in [Python Basics: A Practical Introduction to Python 3](#).

The book uses Python’s built-in [IDLE](#) editor to create and edit Python files and interact with the Python shell, so you will see occasional references to IDLE throughout this tutorial. However, you should have no problems running the example code from the editor and environment of your choice.

Along the way, you’ll have several opportunities to deepen your understanding by following along with the examples. You can download the materials used in the examples by clicking on the link below:

Download the sample materials: [Click here to get the materials you'll use](#) to learn about creating and modifying PDF files in this tutorial.

[Remove ads](#)

Extracting Text From a PDF

In this section, you’ll learn how to read a PDF file and extract the text using the [PyPDF2](#) package. Before you can do that, though, you need to [install it with pip](#):

Shell

```
$ python3 -m pip install PyPDF2
```

Verify the installation by running the following command in your terminal:

Shell

```
$ python3 -m pip show PyPDF2
Name: PyPDF2
Version: 1.26.0
Summary: PDF toolkit
Home-page: http://mstamy2.github.com/PyPDF2
Author: Mathieu Fenniak
Author-email: biziqe@mathieu.fenniak.net
License: UNKNOWN
Location: c:\\users\\david\\python38-32\\lib\\site-packages
Requires:
Required-by:
```

Pay particular attention to the version information. At the time of writing, the latest version of PyPDF2 was 1.26.0. If you have IDLE open, then you'll need to restart it before you can use the PyPDF2 package.

Opening a PDF File

Let's get started by opening a PDF and reading some information about it. You'll use the `Pride_and_Prejudice.pdf` file located in the `practice_files/` folder in the companion repository.

Open IDLE's interactive window and [import](#) the `PdfFileReader` class from the `PyPDF2` package:

```
Python >>>
>>> from PyPDF2 import PdfFileReader
```

To create a new instance of the `PdfFileReader` class, you'll need the [path](#) to the PDF file that you want to open. Let's get that now using the `pathlib` module:

```
Python >>>
>>> from pathlib import Path
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "Pride_and_Prejudice.pdf"
... )
```

The `pdf_path` variable now contains the path to a PDF version of Jane Austen's *Pride and Prejudice*.

Note: You may need to change `pdf_path` so that it corresponds to the location of the `creating-and-modifying-pdfs/` folder on your computer.

Now create the `PdfFileReader` instance:

```
Python >>>
>>> pdf = PdfFileReader(str(pdf_path))
```

You convert `pdf_path` to a [string](#) because `PdfFileReader` doesn't know how to read from a `pathlib.Path` object.

Recall from [chapter 12](#), “File Input and Output,” that all open files should be closed before a program terminates. The `PdfFileReader` object does all of this for you, so you don't need to worry about opening or closing the PDF file!

Now that you've created a `PdfFileReader` instance, you can use it to gather information about the PDF. For example, `.getNumPages()` returns the number of pages contained in the PDF file:

```
Python >>>
>>> pdf.getNumPages()
234
```

Notice that `.getNumPages()` is written in mixedCase, not lower_case_with_underscores as recommended in [PEP 8](#). Remember, PEP 8 is a set of guidelines, not rules. As far as Python is concerned, mixedCase is perfectly acceptable.

Note: PyPDF2 was adapted from the pyPdf package. pyPdf was written in 2005, only four years after PEP 8 was published.

At that time, many Python programmers were migrating from languages in which mixedCase was more common.

You can also access some document information using the `.documentInfo` attribute:

Python

>>>

```
>>> pdf.documentInfo
{'/Title': 'Pride and Prejudice, by Jane Austen', '/Author': 'Chuck',
'/Creator': 'Microsoft® Office Word 2007',
'/CreationDate': 'D:20110812174208', '/ModDate': 'D:20110812174208',
'/Producer': 'Microsoft® Office Word 2007'}
```

The object returned by `.documentInfo` looks like a [dictionary](#), but it's not really the same thing. You can access each item in `.documentInfo` as an [attribute](#).

For example, to get the title, use the `.title` attribute:

Python

>>>

```
>>> pdf.documentInfo.title
'Pride and Prejudice, by Jane Austen'
```

The `.documentInfo` object contains the PDF **metadata**, which is set when a PDF is created.

The `PdfFileReader` class provides all the necessary methods and attributes that you need to access data in a PDF file. Let's explore what you can do with a PDF file and how you can do it!

[Remove ads](#)

Extracting Text From a Page

PDF pages are represented in PyPDF2 with the `PageObject` class. You use `PageObject` instances to interact with pages in a PDF file. You don't need to create your own `PageObject` instances directly. Instead, you can access them through the `PdfFileReader` object's `.getPage()` method.

There are two steps to extracting text from a single PDF page:

1. Get a `PageObject` with `PdfFileReader.getPage()`.
2. Extract the text as a string with the `PageObject` instance's `.extractText()` method.

`Pride_and_Prejudice.pdf` has 234 pages. Each page has an index between 0 and 233. You can get the `PageObject` representing a specific page by passing the page's index to `PdfFileReader.getPage()`:

Python

>>>

```
>>> first_page = pdf.getPage(0)
```

`.getPage()` returns a `PageObject`:

Python

>>>

```
>>> type(first_page)
<class 'PyPDF2.pdf.PageObject'>
```

You can extract the page's text with `PageObject.extractText()`:

Python

>>>

```
>>> first_page.extractText()
'\\n \\n\\n The Project Gutenberg EBook of Pride and Prejudice, by Jane
Austen\\n \\n\\n This eBook is for the use of anyone anywhere at no cost
and with\\n \\n almost no restrictions whatsoever. You may copy it,
give it away or\\n \\n reuse it under the terms of the Project
Gutenberg License included\\n \\n with this eBook or online at
www.gutenberg.org\\n \\n \\n \\nTitle: Pride and Prejudice\\n \\n
\\nAuthor: Jane Austen\\n \\n \\nRelease Date: August 26, 2008
[EBook #1342]\\n \\n [Last updated: August 11, 2011]\\n \\n \\nLanguage:
Eng\\nlish\\n \\n \\nCharacter set encoding: ASCII\\n \\n \\n***\nSTART OF THIS PROJECT GUTENBERG EBOOK PRIDE AND PREJUDICE ***\\n \\n
\\n \\n \\n \\nProduced by Anonymous Volunteers, and David Widger\\n
\\n \\n \\n \\n \\n \\n \\nPRIDE AND PREJUDICE \\n \\n \\n \\nBy Jane
Austen \\n \\n \\n \\n \\nContents\\n \\n'
```

Note that the output displayed here has been formatted to fit better on this page. The output you see on your computer may be formatted differently.

Every PdfFileReader object has a .pages attribute that you can use to iterate over all of the pages in the PDF in order.

For example, the following `for` loop prints the text from every page in the *Pride and Prejudice* PDF:

Python

22

```
>>> for page in pdf.pages:  
...     print(page.extractText())  
...
```

Let's combine everything you've learned and write a program that extracts all of the text from the *Pride and Prejudice.pdf* file and saves it to a *.txt* file.

Putting It All Together

Open a new editor window in IDLE and type in the following code:

Python

```
from pathlib import Path
from PyPDF2 import PdfFileReader

# Change the path below to the correct path for your computer
pdf_path = (
    Path.home()
    / "creating-and-modifying-pdfs"
    / "practice-files"
    / "Pride_and_Prejudice.pdf"
)

# 1
pdf_reader = PdfFileReader(str(pdf_path))
output_file_path = Path.home() / "Pride_and_Prejudice.txt"

# 2
with output_file_path.open(mode="w") as output_file:
    # 3
    title = pdf_reader.getDocumentInfo().title
```

```
title = pdf_reader.getDocumentInfo().title
num_pages = pdf_reader.getNumPages()
output_file.write(f"\n{title}\nNumber of pages: {num_pages}\n\n")

# 4
for page in pdf_reader.pages:
    text = page.extractText()
    output_file.write(text)
```

Let's break that down:

1. First, you assign a new PdfFileReader instance to the pdf_reader variable. You also create a new Path object that points to the file `Pride_and_Prejudice.txt` in your home directory and assign it to the output_file_path variable.
2. Next, you open output_file_path in write mode and assign the file object returned by `.open()` to the variable output_file. The with statement, which you learned about in [chapter 12](#), “File Input and Output,” ensures that the file is closed when the with block exits.
3. Then, inside the with block, you write the PDF title and number of pages to the text file using `output_file.write()`.
4. Finally, you use a for loop to iterate over all the pages in the PDF. At each step in the loop, the next PageObject is assigned to the page variable. The text from each page is extracted with `page.extractText()` and is written to the output_file.

When you save and run the program, it will create a new file in your home directory called `Pride_and_Prejudice.txt` containing the full text of the `Pride_and_Prejudice.pdf` document. Open it up and check it out!

[Remove ads](#)

Check Your Understanding

Expand the block below to check your understanding:

You can expand the block below to see a solution:

Solution: Print Text From a PDF

Show/Hide

When you're ready, you can move on to the next section.

Extracting Pages From a PDF

In the previous section, you learned how to extract all of the text from a PDF file and save it to a .txt file. Now you'll learn how to extract a page or range of pages from an existing PDF and save them to a new PDF.

You can use the PdfFileWriter to create a new PDF file. Let's explore this class and learn the steps needed to create a PDF using PyPDF2.

Using the PdfFileWriter Class

The `PdfFileWriter` class creates new PDF files. In IDLE's interactive window, import the `PdfFileWriter` class and create a new instance called `pdf_writer`:

Python

>>>

```
>>> from PyPDF2 import PdfFileWriter  
>>> pdf_writer = PdfFileWriter()
```

`PdfFileWriter` objects are like blank PDF files. You need to add some pages to them before you can save them to a file.

Go ahead and add a blank page to `pdf_writer`:

Python

>>>

```
>>> page = pdf_writer.addBlankPage(width=72, height=72)
```

The `width` and `height` parameters are required and determine the dimensions of the page in units called **points**. One point equals 1/72 of an inch, so the above code adds a one-inch-square blank page to `pdf_writer`.

`.addBlankPage()` returns a new `PageObject` instance representing the page that you added to the `PdfFileWriter`:

Python

>>>

```
>>> type(page)  
<class 'PyPDF2.pdf.PageObject'>
```

In this example, you've assigned the `PageObject` instance returned by `.addBlankPage()` to the `page` variable, but in practice you don't usually need to do this. That is, you usually call `.addBlankPage()` without assigning the return value to anything:

Python

>>>

```
>>> pdf_writer.addBlankPage(width=72, height=72)
```

To write the contents of `pdf_writer` to a PDF file, pass a file object in binary write mode to `pdf_writer.write()`:

Python

>>>

```
>>> from pathlib import Path  
>>> with Path("blank.pdf").open(mode="wb") as output_file:  
...     pdf_writer.write(output_file)  
...
```

This creates a new file in your current working directory called `blank.pdf`. If you open the file with a PDF reader, such as Adobe Acrobat, then you'll see a document with a single blank one-inch-square page.

Technical detail: Notice that you save the PDF file by passing the file object to the `PdfFileWriter` object's `.write()` method, *not* to the file object's `.write()` method.

In particular, the following code will not work:

Python

>>>

```
>>> with Path("blank.pdf").open(mode="wb") as output_file:  
...     output_file.write(pdf_writer)
```

This approach seems backwards to many new programmers, so make sure you avoid this mistake!

`PdfFileWriter` objects can write to new PDF files, but they can't create new content from scratch other than blank pages.

This might seem like a big problem, but in many situations, you don't need to create new content. Often, you'll work with pages extracted from PDF files that you've opened with a `PdfFileReader` instance.

Note: You'll learn how to create PDF files from scratch below, in the section "Creating a PDF File From Scratch."

In the example you saw above, there were three steps to create a new PDF file using PyPDF2:

1. Create a `PdfFileWriter` instance.
2. Add one or more pages to the `PdfFileWriter` instance.
3. Write to a file using `PdfFileWriter.write()`.

You'll see this pattern over and over as you learn various ways to add pages to a `PdfFileWriter` instance.

[Remove ads](#)

Extracting a Single Page From a PDF

Let's revisit the *Pride and Prejudice* PDF that you worked with in the previous section. You'll open the PDF, extract the first page, and create a new PDF file containing just the single extracted page.

Open IDLE's interactive window and import `PdfFileReader` and `PdfFileWriter` from PyPDF2 as well as the `Path` class from the `pathlib` module:

```
Python >>>
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now open the `Pride_and_Prejudice.pdf` file with a `PdfFileReader` instance:

```
Python >>>
>>> # Change the path to work on your computer if necessary
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "Pride_and_Prejudice.pdf"
... )
>>> input_pdf = PdfFileReader(str(pdf_path))
```

Pass the index `0` to `.getPage()` to get a `PageObject` representing the first page of the PDF:

```
Python >>>
>>> first_page = input_pdf.getPage(0)
```

Now create a new `PdfFileWriter` instance and add `first_page` to it with `.addPage()`:

```
Python >>>
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.addPage(first_page)
```

The `.addPage()` method adds a page to the set of pages in the `pdf_writer` object, just like `.addBlankPage()`. The difference is that it requires an existing `PageObject`.

Now write the contents of `pdf_writer` to a new file:

Python

>>>

```
>>> with Path("first_page.pdf").open(mode="wb") as output_file:  
...     pdf_writer.write(output_file)  
...
```

You now have a new PDF file saved in your current working directory called `first_page.pdf`, which contains the cover page of the `Pride_and_Prejudice.pdf` file. Pretty neat!

Extracting Multiple Pages From a PDF

Let's extract the first chapter from `Pride_and_Prejudice.pdf` and save it to a new PDF.

If you open `Pride_and_Prejudice.pdf` with a PDF viewer, then you can see that the first chapter is on the second, third, and fourth pages of the PDF. Since pages are indexed starting with 0, you'll need to extract the pages at the indices 1, 2, and 3.

You can set everything up by importing the classes you need and opening the PDF file:

Python

>>>

```
>>> from PyPDF2 import PdfFileReader, PdfFileWriter  
>>> from pathlib import Path  
>>> pdf_path = (  
...     Path.home()  
...     / "creating-and-modifying-pdfs"  
...     / "practice_files"  
...     / "Pride_and_Prejudice.pdf"  
... )  
>>> input_pdf = PdfFileReader(str(pdf_path))
```

Your goal is to extract the pages at indices 1, 2, and 3, add these to a new `PdfFileWriter` instance, and then write them to a new PDF file.

One way to do this is to loop over the range of numbers starting at 1 and ending at 3, extracting the page at each step of the loop and adding it to the `PdfFileWriter` instance:

Python

>>>

```
>>> pdf_writer = PdfFileWriter()  
>>> for n in range(1, 4):  
...     page = input_pdf.getPage(n)  
...     pdf_writer.addPage(page)  
...
```

The loop iterates over the numbers 1, 2, and 3 since `range(1, 4)` doesn't include the right-hand endpoint. At each step in the loop, the page at the current index is extracted with `.getPage()` and added to the `pdf_writer` using `.addPage()`.

Now `pdf_writer` has three pages, which you can check with `.getNumPages()`:

Python

>>>

```
>>> pdf_writer.getNumPages()
```

Finally, you can write the extracted pages to a new PDF file:

```
Python >>>
>>> with Path("chapter1.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
```

Now you can open the chapter1.pdf file in your current working directory to read just the first chapter of *Pride and Prejudice*.

Another way to extract multiple pages from a PDF is to take advantage of the fact that PdfFileReader.pages supports [slice notation](#). Let's redo the previous example using .pages instead of looping over a [range object](#).

Start by initializing a new PdfFileWriter object:

```
Python >>>
>>> pdf_writer = PdfFileWriter()
```

Now loop over a slice of .pages from indices starting at 1 and ending at 4:

```
Python >>>
>>> for page in input_pdf.pages[1:4]:
...     pdf_writer.addPage(page)
...
```

Remember that the values in a slice range from the item at the first index in the slice up to, but not including, the item at the second index in the slice. So .pages[1:4] returns an iterable containing the pages with indices 1, 2, and 3.

Finally, write the contents of pdf_writer to the output file:

```
Python >>>
>>> with Path("chapter1_slice.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
```

Now open the chapter1_slice.pdf file in your current working directory and compare it to the chapter1.pdf file you made by looping over the range object. They contain the same pages!

Sometimes you need to extract every page from a PDF. You can use the methods illustrated above to do this, but PyPDF2 provides a shortcut. PdfFileWriter instances have an .appendPagesFromReader() method that you can use to append pages from a PdfFileReader instance.

To use .appendPagesFromReader(), pass a PdfFileReader instance to the method's reader parameter. For example, the following code copies every page from the *Pride and Prejudice* PDF to a PdfFileWriter instance:

```
Python >>>
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.appendPagesFromReader(pdf_reader)
```

pdf_writer now contains every page in pdf_reader!

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Extract The Last Page of a PDF

Show/Hide

You can expand the block below to see a solution:

Solution: Extract The Last Page of a PDF

Show/Hide

When you're ready, you can move on to the next section.

Concatenating and Merging PDFs

Two common tasks when working with PDF files are concatenating and merging several PDFs into a single file.

When you **concatenate** two or more PDFs, you join the files one after another into a single document. For example, a company may concatenate several daily reports into one monthly report at the end of a month.

Merging two PDFs also joins the PDFs into a single file. But instead of joining the second PDF to the end of the first, merging allows you to insert it after a specific page in the first PDF. Then it pushes all of the first PDF's pages after the insertion point to the end of the second PDF.

In this section, you'll learn how to concatenate and merge PDFs using the PyPDF2 package's PdfFileMerger.

Using the PdfFileMerger Class

The PdfFileMerger class is a lot like the PdfFileWriter class that you learned about in the previous section. You can use both classes to write PDF files. In both cases, you add pages to instances of the class and then write them to a file.

The main difference between the two is that PdfFileWriter can only append, or concatenate, pages to the end of the list of pages already contained in the writer, whereas PdfFileMerger can insert, or merge, pages at any location.

Go ahead and create your first PdfFileMerger instance. In IDLE's interactive window, type the following code to import the PdfFileMerger class and create a new instance:

Python

>>>

```
>>> from PyPDF2 import PdfFileMerger  
>>> pdf_merger = PdfFileMerger()
```

PdfFileMerger objects are empty when they're first instantiated. You'll need to add some pages to your object before you can do anything with it.

There are a couple of ways to add pages to the pdf_merger object, and which one you use depends on what you need to accomplish:

- `.append()` concatenates every page in an existing PDF document to the end of the pages currently in the PdfFileMerger.
- `.merge()` inserts all of the pages in an existing PDF document after a specific page in the PdfFileMerger.

You'll look at both methods in this section, starting with `.append()`.

[Remove ads](#)

Concatenating PDFs With .append()

The practice_files/ folder has a subdirectory called expense_reports that contains three expense reports for an employee named Peter Python.

Peter needs to concatenate these three PDFs and submit them to his employer as a single PDF file so that he can get reimbursed for some work-related expenses.

You can start by using the pathlib module to get a list of Path objects for each of the three expense reports in the expense_reports/ folder:

Python

```
>>> from pathlib import Path
>>> reports_dir = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "expense_reports"
... )
```

>>>

After you import the Path class, you need to build the path to the expense_reports/ directory. Note that you may need to alter the code above to get the correct path on your computer.

Once you have the path to the expense_reports/ directory assigned to the reports_dir variable, you can use .glob() to get an iterable of paths to PDF files in the directory.

Take a look at what's in the directory:

Python

```
>>> for path in reports_dir.glob("*.pdf"):
...     print(path.name)
...
Expense report 1.pdf
Expense report 3.pdf
Expense report 2.pdf
```

>>>

The names of the three files are listed, but they aren't in order. Furthermore, the order of the files you see in the output on your computer may not match the output shown here.

In general, the order of paths returned by .glob() is not guaranteed, so you'll need to order them yourself. You can do this by creating a list containing the three file paths and then calling [.sort\(\)](#) on that list:

Python

```
>>> expense_reports = list(reports_dir.glob("*.pdf"))
>>> expense_reports.sort()
```

>>>

Remember that .sort() sorts a list in place, so you don't need to assign the return value to a variable. The expense_reports list will be sorted alphabetically by filename after .list() is called.

To confirm that the sorting worked, loop over expense_reports again and print out the filenames:

Python

```
>>> for path in expense_reports:
...     print(path.name)
```

>>>

```
...  
Expense report 1.pdf  
Expense report 2.pdf  
Expense report 3.pdf
```

That looks good!

Now you can concatenate the three PDFs. To do that, you'll use `PdfFileMerger.append()`, which requires a single string argument representing the path to a PDF file. When you call `.append()`, all of the pages in the PDF file are appended to the set of pages in the `PdfFileMerger` object.

Let's see this in action. First, import the `PdfFileMerger` class and create a new instance:

```
Python >>>  
>>> from PyPDF2 import PdfFileMerger  
>>> pdf_merger = PdfFileMerger()
```

Now loop over the paths in the sorted `expense_reports` list and append them to `pdf_merger`:

```
Python >>>  
>>> for path in expense_reports:  
...     pdf_merger.append(str(path))  
...
```

Notice that each `Path` object in `expense_reports` is converted to a string with `str()` before being passed to `pdf_merger.append()`.

With all of the PDF files in the `expense_reports/` directory concatenated in the `pdf_merger` object, the last thing you need to do is write everything to an output PDF file. `PdfFileMerger` instances have a `.write()` method that works just like the `PdfFileWriter.write()`.

Open a new file in binary write mode, then pass the file object to the `pdf_merger.write()` method:

```
Python >>>  
>>> with Path("expense_reports.pdf").open(mode="wb") as output_file:  
...     pdf_merger.write(output_file)  
...
```

You now have a PDF file in your current working directory called `expense_reports.pdf`. Open it up with a PDF reader and you'll find all three expense reports together in the same PDF file.

[Remove ads](#)

Merging PDFs With `.merge()`

To merge two or more PDFs, use `PdfFileMerger.merge()`. This method is similar to `.append()`, except that you must specify where in the output PDF to insert all of the content from the PDF you are merging.

Take a look at an example. Goggle, Inc. prepared a quarterly report but forgot to include a table of contents. Peter Python noticed the mistake and quickly created a PDF with the missing table of contents. Now he needs to merge that PDF into the original report.

Both the report PDF and the table of contents PDF can be found in the `quarterly_report/` subfolder of the `practice_files` folder. The report is in a file called `report.pdf`, and the table of contents is in a file called `toc.pdf`.

In IDLE's interactive window, import the `PdfFileMerger` class and create the `Path` objects for the `report.pdf` and `toc.pdf` files:

```
Python >>>  
>>> from pathlib import Path  
>>> from PyPDF2 import PdfFileMerger
```

```
>>> report_dir = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "quarterly_report"
... )
>>> report_path = report_dir / "report.pdf"
>>> toc_path = report_dir / "toc.pdf"
```

The first thing you'll do is append the report PDF to a new PdfFileMerger instance using .append():

```
Python >>>
>>> pdf_merger = PdfFileMerger()
>>> pdf_merger.append(str(report_path))
```

Now that pdf_merger has some pages in it, you can merge the table of contents PDF into it at the correct location. If you open the report.pdf file with a PDF reader, then you'll see that the first page of the report is a title page. The second is an introduction, and the remaining pages contain different report sections.

You want to insert the table of contents after the title page and just before the introduction section. Since PDF page indices start with 0 in PyPDF2, you need to insert the table of contents after the page at index 0 and before the page at index 1.

To do that, call pdf_merger.merge() with two arguments:

1. The integer 1, indicating the index of the page at which the table of contents should be inserted
2. A string containing the path of the PDF file for the table of contents

Here's what that looks like:

```
Python >>>
>>> pdf_merger.merge(1, str(toc_path))
```

Every page in the table of contents PDF is inserted *before* the page at index 1. Since the table of contents PDF is only one page, it gets inserted at index 1. The page currently at index 1 then gets shifted to index 2. The page currently at index 2 gets shifted to index 3, and so on.

Now write the merged PDF to an output file:

```
Python >>>
>>> with Path("full_report.pdf").open(mode="wb") as output_file:
...     pdf_merger.write(output_file)
... 
```

You now have a full_report.pdf file in your current working directory. Open it up with a PDF reader and check that the table of contents was inserted at the correct spot.

Concatenating and merging PDFs are common operations. While the examples in this section are admittedly somewhat contrived, you can imagine how useful a program would be for merging thousands of PDFs or for automating routine tasks that would otherwise take a human lots of time to complete.

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Concatenate Two PDFs

Show/Hide

You can expand the block below to see a solution:

When you're ready, you can move on to the next section.

[Remove ads](#)

Rotating and Cropping PDF Pages

So far, you've learned how to extract text and pages from PDFs and how to and concatenate and merge two or more PDF files. These are all common operations with PDFs, but PyPDF2 has many other useful features.

Note: This tutorial is adapted from the chapter "Creating and Modifying PDF Files" in [Python Basics: A Practical Introduction to Python 3](#). If you enjoy what you're reading, then be sure to check out [the rest of the book](#).

In this section, you'll learn how to rotate and crop pages in a PDF file.

Rotating Pages

You'll start by learning how to rotate pages. For this example, you'll use the `ugly.pdf` file in the `practice_files` folder. The `ugly.pdf` file contains a lovely version of Hans Christian Andersen's *The Ugly Duckling*, except that every odd-numbered page is rotated counterclockwise by ninety degrees.

Let's fix that. In a new IDLE interactive window, start by importing the `PdfFileReader` and `PdfFileWriter` classes from PyPDF2, as well as the `Path` class from the `pathlib` module:

```
Python >>>
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now create a `Path` object for the `ugly.pdf` file:

```
Python >>>
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "ugly.pdf"
... )
```

Finally, create new `PdfFileReader` and `PdfFileWriter` instances:

```
Python >>>
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Your goal is to use `pdf_writer` to create a new PDF file in which all of the pages have the correct orientation. The even-numbered pages in the PDF are already properly oriented, but the odd-numbered pages are rotated counterclockwise by ninety degrees.

To correct the problem, you'll use `PageObject.rotateClockwise()`. This method takes an integer argument, in degrees, and rotates a page clockwise by that many degrees. For example, `.rotateClockwise(90)` rotates a PDF page clockwise by ninety degrees.

Note: In addition to `.rotateClockwise()`, the `PageObject` class also has `.rotateCounterClockwise()` for rotating pages counterclockwise.

There are several ways you can go about rotating pages in the PDF. We'll discuss two different ways of doing it. Both of them rely on `.rotateClockwise()`, but they take different approaches to determine which pages get rotated.

The first technique is to loop over the indices of the pages in the PDF and check if each index corresponds to a page that needs to be rotated. If so, then you'll call `.rotateClockwise()` to rotate the page and then add the page to `pdf_writer`.

Here's what that looks like:

```
Python >>>
>>> for n in range(pdf_reader.getNumPages()):
...     page = pdf_reader.getPage(n)
...     if n % 2 == 0:
...         page.rotateClockwise(90)
...     pdf_writer.addPage(page)
...
...
```

Notice that the page gets rotated if the index is even. That might seem strange since the odd-numbered pages in the PDF are the ones that are rotated incorrectly. However, the page numbers in the PDF start with 1, whereas the page indices start with 0. That means odd-numbered PDF pages have even indices.

If that makes your head spin, don't worry! Even after years of dealing with stuff like this, professional programmers still get tripped up by these sorts of things!

Note: When you execute the `for` loop above, you'll see a bunch of output in IDLE's interactive window. That's because `.rotateClockwise()` returns a `PageObject` instance.

You can ignore this output for now. When you execute programs from IDLE's editor window, this output won't be visible.

Now that you've rotated all the pages in the PDF, you can write the content of `pdf_writer` to a new file and check that everything worked:

```
Python >>>
>>> with Path("ugly_rotated.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
...
```

You should now have a file in your current working directory called `ugly_rotated.pdf`, with the pages from the `ugly.pdf` file all rotated correctly.

The problem with the approach you just used to rotate the pages in the `ugly.pdf` file is that it depends on knowing ahead of time which pages need to be rotated. In a real-world scenario, it isn't practical to go through an entire PDF taking note of which pages to rotate.

In fact, you can determine which pages need to be rotated without prior knowledge. Well, *sometimes* you can.

Let's see how, starting with a new `PdfFileReader` instance:

Python

>>>

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

You need to do this because you altered the pages in the old `PdfFileReader` instance by rotating them. So, by creating a new instance, you're starting fresh.

`PageObject` instances maintain a dictionary of values containing information about the page:

Python

>>>

```
>>> pdf_reader.getPage(0)
{'/Contents': [IndirectObject(11, 0), IndirectObject(12, 0),
IndirectObject(13, 0), IndirectObject(14, 0), IndirectObject(15, 0),
IndirectObject(16, 0), IndirectObject(17, 0), IndirectObject(18, 0)],
'/Rotate': -90, '/Resources': {'/ColorSpace': {'/CS1':
IndirectObject(19, 0), '/CS0': IndirectObject(19, 0)}, '/XObject':
{'/Im0': IndirectObject(21, 0)}, '/Font': {'/TT1':
IndirectObject(23, 0), '/TT0': IndirectObject(25, 0)}, '/ExtGState':
{'/GS0': IndirectObject(27, 0)}}, '/CropBox': [0, 0, 612, 792],
'/Parent': IndirectObject(1, 0), '/MediaBox': [0, 0, 612, 792],
'/Type': '/Page', '/StructParents': 0}
```

Yikes! Mixed in with all that nonsensical-looking stuff is a key called `/Rotate`, which you can see on the fourth line of output above. The value of this key is `-90`.

You can access the `/Rotate` key on a `PageObject` using subscript notation, just like you can on a Python dict object:

Python

>>>

```
>>> page = pdf_reader.getPage(0)
>>> page["/Rotate"]
-90
```

If you look at the `/Rotate` key for the second page in `pdf_reader`, you'll see that it has a value of `0`:

Python

>>>

```
>>> page = pdf_reader.getPage(1)
>>> page["/Rotate"]
0
```

What all this means is that the page at index `0` has a rotation value of `-90` degrees. In other words, it's been rotated counterclockwise by ninety degrees. The page at index `1` has a rotation value of `0`, so it has not been rotated at all.

If you rotate the first page using `.rotateClockwise()`, then the value of `/Rotate` changes from `-90` to `0`:

Python

>>>

```
>>> page = pdf_reader.getPage(0)
>>> page["/Rotate"]
-90
>>> page.rotateClockwise(90)
>>> page["/Rotate"]
0
```

Now that you know how to inspect the `/Rotate` key, you can use it to rotate the pages in the `ugly.pdf` file.

The first thing you need to do is reinitialize your `pdf_reader` and `pdf_writer` objects so that you get a fresh start:

Python

>>>

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Now write a loop that loops over the pages in the `pdf_reader.pages` iterable, checks the value of `/Rotate`, and

rotates the page if that value is -90:

```
Python >>>
>>> for page in pdf_reader.pages:
...     if page["/Rotate"] == -90:
...         page.rotateClockwise(90)
...     pdf_writer.addPage(page)
... 
```

Not only is this loop slightly shorter than the loop in the first solution, but it doesn't rely on any prior knowledge of which pages need to be rotated. You could use a loop like this to rotate pages in any PDF without ever having to open it up and look at it.

To finish out the solution, write the contents of `pdf_writer` to a new file:

```
Python >>>
>>> with Path("ugly_rotated2.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
... 
```

Now you can open the `ugly_rotated2.pdf` file in your current working directory and compare it to the `ugly_rotated.pdf` file you generated earlier. They should look identical.

Note: One word of warning about the `/Rotate` key: it's not guaranteed to exist on a page.

If the `/Rotate` key doesn't exist, then that usually means that the page has not been rotated. However, that isn't always a safe assumption.

If a `PageObject` has no `/Rotate` key, then a `KeyError` will be raised when you try to access it. You can [catch this exception with a `try...except` block](#).

The value of `/Rotate` may not always be what you expect. For example, if you scan a paper document with the page rotated ninety degrees counterclockwise, then the contents of the PDF will appear rotated. However, the `/Rotate` key may have the value `0`.

This is one of many quirks that can make working with PDF files frustrating. Sometimes you'll just need to open a PDF in a PDF reader program and manually figure things out.

 [Remove ads](#)

Cropping Pages

Another common operation with PDFs is cropping pages. You might need to do this to split a single page into multiple pages or to extract just a small portion of a page, such as a signature or a figure.

For example, the `practice_files` folder includes a file called `half_and_half.pdf`. This PDF contains a portion of Hans Christian Andersen's *The Little Mermaid*.

Each page in this PDF has two columns. Let's split each page into two pages, one for each column.

To get started, import the `PdfFileReader` and `PdfFileWriter` classes from `PyPDF2` and the `Path` class from the `pathlib` module:

Python

>>>

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now create a Path object for the half_and_half.pdf file:

Python

>>>

```
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "half_and_half.pdf"
... )
```

Next, create a new PdfFileReader object and get the first page of the PDF:

Python

>>>

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> first_page = pdf_reader.getPage(0)
```

To crop the page, you first need to know a little bit more about how pages are structured. PageObject instances like first_page have a .mediaBox attribute that represents a rectangular area defining the boundaries of the page.

You can use IDLE's interactive window to explore the .mediaBox before using it crop the page:

Python

>>>

```
>>> first_page.mediaBox
RectangleObject([0, 0, 792, 612])
```

The .mediaBox attribute returns a RectangleObject. This object is defined in the PyPDF2 package and represents a rectangular area on the page.

The list [0, 0, 792, 612] in the output defines the rectangular area. The first two numbers are the x- and y-coordinates of the lower-left corner of the rectangle. The third and fourth numbers represent the width and height of the rectangle, respectively. The units of all of the values are points, which are equal to 1/72 of an inch.

RectangleObject([0, 0, 792, 612]) represents a rectangular region with the lower-left corner at the origin, a width of 792 points, or 11 inches, and a height of 612 points, or 8.5 inches. Those are the dimensions of a standard letter-sized page in landscape orientation, which is used for the example PDF of *The Little Mermaid*. A letter-sized PDF page in portrait orientation would return the output RectangleObject([0, 0, 612, 792]).

A RectangleObject has four attributes that return the coordinates of the rectangle's corners: .lowerLeft, .lowerRight, .upperLeft, and .upperRight. Just like the width and height values, these coordinates are given in points.

You can use these four properties to get the coordinates of each corner of the RectangleObject:

Python

>>>

```
>>> first_page.mediaBox.lowerLeft
(0, 0)
>>> first_page.mediaBox.lowerRight
(792, 0)
>>> first_page.mediaBox.upperLeft
(0, 612)
>>> first_page.mediaBox.upperRight
(792, 612)
```

Each property returns a [tuple](#) containing the coordinates of the specified corner. You can access individual coordinates with square brackets just like you would any other Python tuple:

```
Python >>>
>>> first_page.mediaBox.upperRight[0]
792
>>> first_page.mediaBox.upperRight[1]
612
```

You can alter the coordinates of a `mediaBox` by assigning a new tuple to one of its properties:

```
Python >>>
>>> first_page.mediaBox.upperLeft = (0, 480)
>>> first_page.mediaBox.upperLeft
(0, 480)
```

When you change the `.upperLeft` coordinates, the `.upperRight` attribute automatically adjusts to preserve a rectangular shape:

```
Python >>>
>>> first_page.mediaBox.upperRight
(792, 480)
```

When you alter the coordinates of the `RectangleObject` returned by `.mediaBox`, you effectively crop the page. The `first_page` object now contains only the information present within the boundaries of the new `RectangleObject`.

Go ahead and write the cropped page to a new PDF file:

```
Python >>>
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.addPage(first_page)
>>> with Path("cropped_page.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
```

If you open the `cropped_page.pdf` file in your current working directory, then you'll see that the top portion of the page has been removed.

How would you crop the page so that just the text on the left side of the page is visible? You would need to cut the horizontal dimensions of the page in half. You can achieve this by altering the `.upperRight` coordinates of the `.mediaBox` object. Let's see how that works.

First, you need to get new `PdfFileReader` and `PdfFileWriter` objects since you've just altered the first page in `pdf_reader` and added it to `pdf_writer`:

```
Python >>>
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Now get the first page of the PDF:

```
Python >>>
>>> first_page = pdf_reader.getPage(0)
```

This time, let's work with a copy of the first page so that the page you just extracted stays intact. You can do that by

importing the `copy` module from Python's standard library and using `deepcopy()` to make a copy of the page:

Python

>>>

```
>>> import copy
>>> left_side = copy.deepcopy(first_page)
```

Now you can alter `left_side` without changing the properties of `first_page`. That way, you can use `first_page` later to extract the text on the right side of the page.

Now you need to do a little bit of math. You already worked out that you need to move the upper right-hand corner of the `.mediaBox` to the top center of the page. To do that, you'll create a new tuple with the first component equal to half the original value and assign it to the `.upperRight` property.

First, get the current coordinates of the upper-right corner of the `.mediaBox`.

Python

>>>

```
>>> current_coords = left_side.mediaBox.upperRight
```

Then create a new tuple whose first coordinate is half the value of the current coordinate and second coordinate is the same as the original:

Python

>>>

```
>>> new_coords = (current_coords[0] / 2, current_coords[1])
```

Finally, assign the new coordinates to the `.upperRight` property:

Python

>>>

```
>>> left_side.mediaBox.upperRight = new_coords
```

You've now cropped the original page to contain only the text on the left side! Let's extract the right side of the page next.

First get a new copy of `first_page`:

Python

>>>

```
>>> right_side = copy.deepcopy(first_page)
```

Move the `.upperLeft` corner instead of the `.upperRight` corner:

Python

>>>

```
>>> right_side.mediaBox.upperLeft = new_coords
```

This sets the upper-left corner to the same coordinates that you moved the upper-right corner to when extracting the left side of the page. So, `right_side.mediaBox` is now a rectangle whose upper-left corner is at the top center of the page and whose upper-right corner is at the top right of the page.

Finally, add the `left_side` and `right_side` pages to `pdf_writer` and write them to a new PDF file:

Python

>>>

```
>>> pdf_writer.addPage(left_side)
>>> pdf_writer.addPage(right_side)
>>> with Path("cropped_pages.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
... 
```

Now open the `cropped_pages.pdf` file with a PDF reader. You should see a file with two pages, the first containing the text from the left-hand side of the original first page, and the second containing the text from the original right-

hand side.

[Remove ads](#)

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Rotate Pages in a PDF

Show/Hide

You can expand the block below to see a solution:

Solution: Rotate Pages in a PDF

Show/Hide

Encrypting and Decrypting PDFs

Sometimes PDF files are password protected. With the PyPDF2 package, you can work with encrypted PDF files as well as add password protection to existing PDFs.

Note: This tutorial is adapted from the chapter “Creating and Modifying PDF Files” in [Python Basics: A Practical Introduction to Python 3](#). If you enjoy what you’re reading, then be sure to check out [the rest of the book](#).

Encrypting PDFs

You can add password protection to a PDF file using the `.encrypt()` method of a `PdfFileWriter()` instance. It has two main parameters:

1. `user_pwd` sets the user password. This allows for opening and reading the PDF file.
2. `owner_pwd` sets the owner password. This allows for opening the PDF without any restrictions, including editing.

Let’s use `.encrypt()` to add a password to a PDF file. First, open the `newsletter.pdf` file in the `practice_files` directory:

Python

>>>

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "newsletter.pdf"
... )
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Now create a new `PdfFileWriter` instance and add the pages from `pdf_reader` to it:

Python

>>>

```
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.appendPagesFromReader(pdf_reader)
```

Next, add the password “`SuperSecret`” with `pdf_writer.encrypt()`:

Python

>>>

```
>>> pdf_writer.encrypt(user_pwd="SuperSecret")
```

When you set only `user_pwd`, the `owner_pwd` argument defaults to the same string. So, the above line of code sets both the user and owner passwords.

Finally, write the encrypted PDF to an output file in your home directory called `newsletter_protected.pdf`:

Python

>>>

```
>>> output_path = Path.home() / "newsletter_protected.pdf"
>>> with output_path.open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
```

When you open the PDF with a PDF reader, you'll be prompted to enter a password. Enter "SuperSecret" to open the PDF.

If you need to set a separate owner password for the PDF, then pass a second string to the `owner_pwd` parameter:

Python

>>>

```
>>> user_pwd = "SuperSecret"
>>> owner_pwd = "ReallySuperSecret"
>>> pdf_writer.encrypt(user_pwd=user_pwd, owner_pwd=owner_pwd)
```

In this example, the user password is "SuperSecret" and the owner password is "ReallySuperSecret".

When you encrypt a PDF file with a password and attempt to open it, you must provide the password before you can view its contents. This protection extends to reading from the PDF in a Python program. Next, let's see how to decrypt PDF files with PyPDF2.

Decrypting PDFs

To decrypt an encrypted PDF file, use the `.decrypt()` method of a `PdfFileReader` instance.

`.decrypt()` has a single parameter called `password` that you can use to provide the password for decryption. The privileges you have when opening the PDF depend on the argument you passed to the `password` parameter.

Let's open the encrypted `newsletter_protected.pdf` file that you created in the previous section and use PyPDF2 to decrypt it.

First, create a new `PdfFileReader` instance with the path to the protected PDF:

Python

>>>

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> pdf_path = Path.home() / "newsletter_protected.pdf"
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Before you decrypt the PDF, check out what happens if you try to get the first page:

Python

>>>

```
>>> pdf_reader.getPage(0)
Traceback (most recent call last):
  File "/Users/damos/github/realpython/python-basics-exercises/venv/
    lib/python38-32/site-packages/PyPDF2/pdf.py", line 1617, in getObject
      raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
```

A `PdfReadError` exception is raised, informing you that the PDF file has not been decrypted.

Note: The above [traceback](#) has been shortened to highlight the important parts. The [traceback](#) you see on your computer will be much longer.

Go ahead and decrypt the file now...

Python

>>>

```
>>> pdf_reader.decrypt(password="SuperSecret")
1
```

.decrypt() returns an integer representing the success of the decryption:

- **0** indicates that the password is incorrect.
- **1** indicates that the user password was matched.
- **2** indicates that the owner password was matched.

Once you've decrypted the file, you can access the contents of the PDF:

Python

>>>

```
>>> pdf_reader.getPage(0)
{'/Contents': IndirectObject(7, 0), '/CropBox': [0, 0, 612, 792],
 '/MediaBox': [0, 0, 612, 792], '/Parent': IndirectObject(1, 0),
 '/Resources': IndirectObject(8, 0), '/Rotate': 0, '/Type': '/Page'}
```

Now you can extract text and crop or rotate pages to your heart's content!

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Encrypt a PDF

Show/Hide

You can expand the block below to see a solution:

Solution: Encrypt a PDF

Show/Hide

Creating a PDF File From Scratch

The PyPDF2 package is great for reading and modifying existing PDF files, but it has a major limitation: you can't use it to create a new PDF file. In this section, you'll use the [ReportLab Toolkit](#) to generate PDF files from scratch.

ReportLab is a full-featured solution for creating PDFs. There is a commercial version that costs money to use, but a limited-feature open source version is also available.

Note: This section is not meant to be an exhaustive introduction to ReportLab, but rather a sample of what is possible.

For more examples, check out the ReportLab's [code snippet page](#).

Installing reportlab

To get started, you need to install reportlab with pip:

Shell

```
$ python3 -m pip install reportlab
```

You can verify the installation with pip show:

Shell

```
$ python3 -m pip show reportlab
Name: reportlab
Version: 3.5.34
Summary: The Reportlab Toolkit
```

```
Home-page: http://www.reportlab.com/
Author: Andy Robinson, Robin Becker, the ReportLab team
       and the community
Author-email: reportlab-users@lists2.reportlab.com
License: BSD license (see license.txt for details),
         Copyright (c) 2000-2018, ReportLab Inc.
Location: c:\users\davea\venv\lib\site-packages
Requires: pillow
Required-by:
```

At the time of writing, the latest version of reportlab was 3.5.34. If you have IDLE open, then you'll need to restart it before you can use the reportlab package.

Using the Canvas Class

The main interface for creating PDFs with reportlab is the canvas class, which is located in the reportlab.pdfgen.canvas module.

Open a new IDLE interactive window and type the following to import the canvas class:

```
Python >>>
>>> from reportlab.pdfgen.canvas import Canvas
```

When you make a new canvas instance, you need to provide a string with the filename of the PDF you're creating. Go ahead and create a new Canvas instance for the file hello.pdf:

```
Python >>>
>>> canvas = Canvas("hello.pdf")
```

You now have a canvas instance that you've assigned to the variable name canvas and that is associated with a file in your current working directory called hello.pdf. The file hello.pdf does not exist yet, though.

Let's add some text to the PDF. To do that, you use .drawString():

```
Python >>>
>>> canvas.drawString(72, 72, "Hello, World")
```

The first two arguments passed to .drawString() determine the location on the canvas where the text is written. The first specifies the distance from the left edge of the canvas, and the second specifies the distance from the bottom edge.

The values passed to .drawString() are measured in points. Since a point equals 1/72 of an inch, .drawString(72, 72, "Hello, World") draws the string "Hello, World" one inch from the left and one inch from the bottom of the page.

To save the PDF to a file, use .save():

```
Python >>>
>>> canvas.save()
```

You now have a PDF file in your current working directory called hello.pdf. You can open it with a PDF reader and see the text Hello, World at the bottom of the page!

There are a few things to notice about the PDF you just created:

1. The default page size is A4, which is not the same as the standard US letter page size.
2. The font defaults to Helvetica with a font size of 12 points.

You're not stuck with these settings.

Setting the Page Size

When you instantiate a Canvas object, you can change the page size with the optional pageSize parameter. This

When you instantiate a `Canvas` object, you can change the page size with the optional `pagesize` parameter. This parameter accepts a tuple of [floating-point values](#) representing the width and height of the page in points.

For example, to set the page size to 8.5 inches wide by 11 inches tall, you would create the following canvas:

Python

```
canvas = Canvas("hello.pdf", pagesize=(612.0, 792.0))
```

(612, 792) represents a letter-sized paper because 8.5 times 72 is 612, and 11 times 72 is 792.

If doing the math to convert points to inches or centimeters isn't your cup of tea, then you can use the `reportlab.lib.units` module to help you with the conversions. The `.units` module contains several helper objects, such as `inch` and `cm`, that simplify your conversions.

Go ahead and import the `inch` and `cm` objects from the `reportlab.lib.units` module:

Python

>>>

```
>>> from reportlab.lib.units import inch, cm
```

Now you can inspect each object to see what they are:

Python

>>>

```
>>> cm  
28.346456692913385  
>>> inch  
72.0
```

Both `cm` and `inch` are floating-point values. They represent the number of points contained in each unit. `inch` is 72.0 points and `cm` is 28.346456692913385 points.

To use the units, multiply the unit name by the number of units that you want to convert to points. For example, here's how to use `inch` to set the page size to 8.5 inches wide by 11 inches tall:

Python

>>>

```
>>> canvas = Canvas("hello.pdf", pagesize=(8.5 * inch, 11 * inch))
```

By passing a tuple to `pagesize`, you can create any size of page that you want. However, the `reportlab` package has some standard built-in page sizes that are easier to work with.

The page sizes are located in the `reportlab.lib.pagesizes` module. For example, to set the page size to letter, you can import the `LETTER` object from the `pagesizes` module and pass it to the `pagesize` parameter when instantiating your `Canvas`:

Python

>>>

```
>>> from reportlab.lib.pagesizes import LETTER  
>>> canvas = Canvas("hello.pdf", pagesize=LETTER)
```

If you inspect the `LETTER` object, then you'll see that it's a tuple of floats:

Python

>>>

```
>>> LETTER  
(612.0, 792.0)
```

The `reportlab.lib.pagesize` module contains many standard page sizes. Here are a few with their dimensions:

Page Size	Dimensions
A4	210 mm x 297 mm
LETTER	8.5 in x 11 in

In addition to these, the module contains definitions for all of the [ISO 216 standard paper sizes](#).

Setting Font Properties

You can also change the font, font size, and font color when you write text to the canvas.

To change the font and font size, you can use `.setFont()`. First, create a new Canvas instance with the filename `font-example.pdf` and a letter page size:

Python

>>>

```
>>> canvas = Canvas("font-example.pdf", pagesize=LETTER)
```

Then set the font to Times New Roman with a size of 18 points:

Python

>>>

```
>>> canvas.setFont("Times-Roman", 18)
```

Finally, write the string "Times New Roman (18 pt)" to the canvas and save it:

Python

>>>

```
>>> canvas.drawString(1 * inch, 10 * inch, "Times New Roman (18 pt)")  
>>> canvas.save()
```

With these settings, the text will be written one inch from the left side of the page and ten inches from the bottom. Open up the `font-example.pdf` file in your current working directory and check it out!

There are three fonts available by default:

1. "Courier"
2. "Helvetica"
3. "Times-Roman"

Each font has bolded and italicized variants. Here's a list of all the font variations available in `reportlab`:

- "Courier"
- "Courier-Bold"
- "Courier-BoldOblique"
- "Courier-Oblique"
- "Helvetica"
- "Helvetica-Bold"
- "Helvetica-BoldOblique"
- "Helvetica-Oblique"
- "Times-Bold"
- "Times-BoldItalic"
- "Times-Italic"
- "Times-Roman"

You can also set the font color using `.setFillColor()`. In the following example, you create a PDF file with blue text named `font-colors.pdf`:

Python

```
from reportlab.lib.colors import blue  
from reportlab.lib.pagesizes import LETTER
```

```

from reportlab.lib.units import inch
from reportlab.pdfgen.canvas import Canvas

canvas = Canvas("font-colors.pdf", pagesize=LETTER)

# Set font to Times New Roman with 12-point size
canvas.setFont("Times-Roman", 12)

# Draw blue text one inch from the left and ten
# inches from the bottom
canvas.setFillColor(blue)
canvas.drawString(1 * inch, 10 * inch, "Blue text")

# Save the PDF file
canvas.save()

```

blue is an object imported from the `reportlab.lib.colors` module. This module contains several common colors. A full list of colors can be found in the [reportlab source code](#).

The examples in this section highlight the basics of working with the `Canvas` object. But you've only scratched the surface. With `reportlab`, you can create tables, forms, and even high-quality graphics from scratch!

The [ReportLab User Guide](#) contains a plethora of examples of how to generate PDF documents from scratch. It's a great place to start if you're interested in learning more about creating PDFs with Python.

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Create a PDF From Scratch

Show/Hide

You can expand the block below to see a solution:

Solution: Create a PDF From Scratch

Show/Hide

When you're ready, you can move on to the next section.

Conclusion: Create and Modify PDF Files in Python

In this tutorial, you learned how to create and modify PDF files with the `PyPDF2` and `reportlab` packages. If you want to follow along with the examples you just saw, then be sure to download the materials by clicking the link below:

Download the sample materials: [Click here to get the materials you'll use](#) to learn about creating and modifying PDF files in this tutorial.

With PyPDF2, you learned how to:

- **Read** PDF files and **extract** text using the `PdfFileReader` class
- **Write** new PDF files using the `PdfFileWriter` class
- **Concatenate** and **merge** PDF files using the `PdfFileMerger` class
- **Rotate** and **crop** PDF pages
- **Encrypt** and **decrypt** PDF files with passwords

You also had an introduction to creating PDF files from scratch with the `reportlab` package. **You learned how to:**

- Use the `Canvas` class
- **Write** text to a `Canvas` with `.drawString()`
- Set the **font** and **font size** with `.setFont()`
- Change the **font color** with `.setFillColor()`

`reportlab` is a powerful PDF creation tool, and you only scratched the surface of what's possible. If you enjoyed what you learned in this sample from [Python Basics: A Practical Introduction to Python 3](#), then be sure to check out

[the rest of the book.](#)

Happy coding!

About David Amos

David is a mathematician by training, a data scientist/Python developer by profession, and a coffee junkie by choice.

[» More about David](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

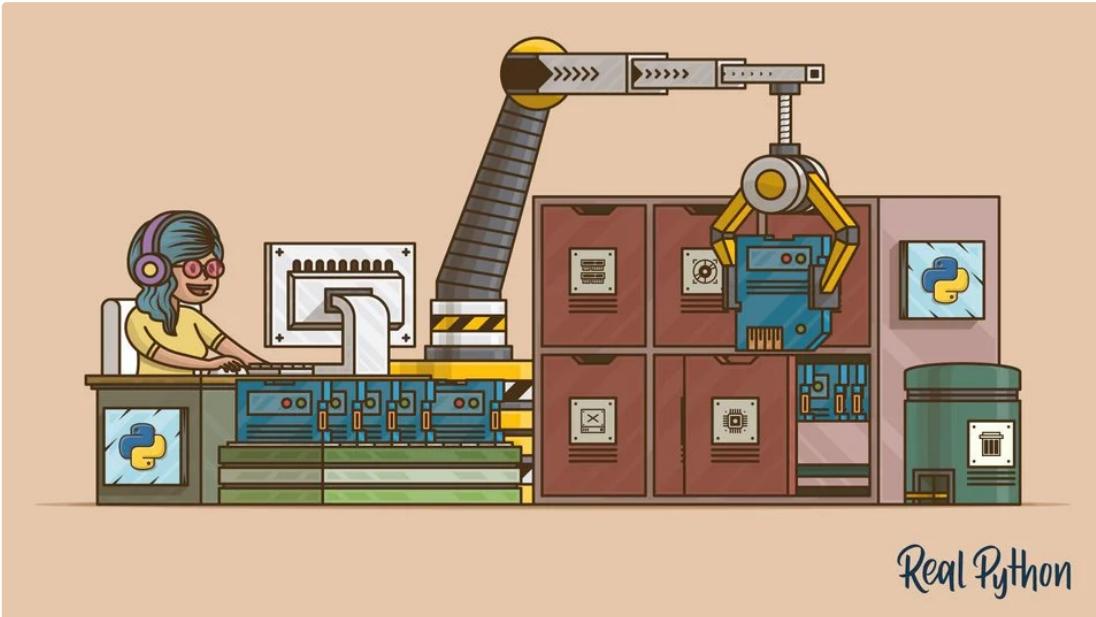
[Aldren](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Real Python

Memory Management in Python

by Alexander VanTol 28 Comments intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Memory Is an Empty Book](#)
- [Memory Management: From Hardware to Software](#)
- [The Default Python Implementation](#)
- [The Global Interpreter Lock \(GIL\)](#)
- [Garbage Collection](#)
- [CPython's Memory Management](#)
 - [Pools](#)
 - [Blocks](#)
 - [Arenas](#)
- [Conclusion](#)



Ever wonder how Python handles your data behind the scenes? How are your variables stored in memory? When do they get deleted?

In this article, we're going to do a deep dive into the internals of Python to understand how it handles memory management.

By the end of this article, you'll:

- Learn more about low-level computing, specifically as relates to memory
- Understand how Python abstracts lower-level operations
- Learn about Python's internal memory management algorithms

Understanding Python's internals will also give you better insight into some of Python's behaviors. Hopefully, you'll gain a new appreciation for Python as well. So much logic is happening behind the scenes to ensure your program

works the way you expect.

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Memory Is an Empty Book

You can begin by thinking of a computer's memory as an empty book intended for short stories. There's nothing written on the pages yet. Eventually, different authors will come along. Each author wants some space to write their story in.

Since they aren't allowed to write over each other, they must be careful about which pages they write in. Before they begin writing, they consult the manager of the book. The manager then decides where in the book they're allowed to write.

Since this book is around for a long time, many of the stories in it are no longer relevant. When no one reads or references the stories, they are removed to make room for new stories.

In essence, computer memory is like that empty book. In fact, it's common to call fixed-length contiguous blocks of memory **pages**, so this analogy holds pretty well.

The authors are like different applications or processes that need to store data in memory. The manager, who decides where the authors can write in the book, plays the role of a memory manager of sorts. The person who removed the old stories to make room for new ones is a garbage collector.

Memory Management: From Hardware to Software

Memory management is the process by which applications read and write data. A memory manager determines where to put an application's data. Since there's a finite chunk of memory, like the pages in our book analogy, the manager has to find some free space and provide it to the application. This process of providing memory is generally called memory **allocation**.

On the flip side, when data is no longer needed, it can be deleted, or **freed**. But freed to where? Where did this "memory" come from?

Somewhere in your computer, there's a physical device storing data when you're running your Python programs. There are many layers of abstraction that the Python code goes through before the objects actually get to the hardware though.

One of the main layers above the hardware (such as RAM or a hard drive) is the operating system (OS). It carries out (or denies) requests to read and write memory.

Above the OS, there are applications, one of which is the default Python implementation (included in your OS or downloaded from [python.org](https://www.python.org).) Memory management for your Python code is handled by the Python application. The algorithms and structures that the Python application uses for memory management is the focus of this article.

The Default Python Implementation

The default Python implementation, CPython, is actually written in the C programming language.

When I first heard this, it blew my mind. A language that's written in another language?! Well, not really, but sort of.

The Python language is defined in a [reference manual](#) written in English. However, that manual isn't all that useful by itself. You still need something to interpret written code based on the rules in the manual.

You also need something to actually execute interpreted code on a computer. The default Python implementation fulfills both of those requirements. It converts your Python code into instructions that it then runs on a virtual machine.

Note: Virtual machines are like physical computers, but they are implemented in software. They typically

Python is an interpreted programming language. Your Python code actually gets compiled down to more computer-readable instructions called [bytecode](#). These instructions get **interpreted** by a virtual machine when you run your code.

Have you ever seen a `.pyc` file or a `__pycache__` folder? That's the bytecode that gets interpreted by the virtual machine.

It's important to note that there are implementations other than CPython. [IronPython](#) compiles down to run on Microsoft's Common Language Runtime. [Jython](#) compiles down to Java bytecode to run on the Java Virtual Machine. Then there's [PyPy](#), but that deserves its own entire article, so I'll just mention it in passing.

For the purposes of this article, I'll focus on the memory management done by the default implementation of Python, CPython.

Disclaimer: While a lot of this information will carry through to new versions of Python, things may change in the future. Note that the referenced version for this article is the current latest version of Python, [3.7](#).

Okay, so CPython is written in C, and it interprets Python bytecode. What does this have to do with memory management? Well, the memory management algorithms and structures exist in the CPython code, in C. To understand the memory management of Python, you have to get a basic understanding of CPython itself.

CPython is written in C, which does not natively support [object-oriented programming](#). Because of that, there are quite a bit of interesting designs in the CPython code.

You may have heard that everything in Python is an object, even types such as `int` and `str`. Well, it's true on an implementation level in CPython. There is a struct called a `PyObject`, which every other object in CPython uses.

Note: A struct, or **structure**, in C is a custom data type that groups together different data types. To compare to object-oriented languages, it's like a class with attributes and no methods.

The `PyObject`, the grand-daddy of all objects in Python, contains only two things:

- **ob_refcnt**: reference count
- **ob_type**: pointer to another type

The reference count is used for garbage collection. Then you have a [pointer](#) to the actual object type. That object type is just another struct that describes a Python object (such as a `dict` or `int`).

Each object has its own object-specific memory allocator that knows how to get the memory to store that object. Each object also has an object-specific memory deallocator that "frees" the memory once it's no longer needed.

However, there's an important factor in all this talk about allocating and freeing memory. Memory is a shared resource on the computer, and bad things can happen if two different processes try to write to the same location at the same time.

The Global Interpreter Lock (GIL)

The GIL is a solution to the common problem of dealing with shared resources, like memory in a computer. When two threads try to modify the same resource at the same time, they can step on each other's toes. The end result can be a garbled mess where neither of the threads ends up with what they wanted.

Consider the book analogy again. Suppose that two authors stubbornly decide that it's their turn to write. Not only that, but they both need to write on the same page of the book at the same time.

They each ignore the other's attempt to craft a story and begin writing on the page. The end result is two stories on top of each other, which makes the whole page completely unreadable.

One solution to this problem is a single, global lock on the interpreter when a thread is interacting with the shared resource (the page in the book). In other words, only one author can write at a time.

Python's GIL accomplishes this by locking the entire interpreter, meaning that it's not possible for another thread to

step on the current one. When CPython handles memory, it uses the GIL to ensure that it does so safely.

There are pros and cons to this approach, and the GIL is heavily debated in the Python community. To read more about the GIL, I suggest checking out [What is the Python Global Interpreter Lock \(GIL\)?](#).

Garbage Collection

Let's revisit the book analogy and assume that some of the stories in the book are getting very old. No one is reading or referencing those stories anymore. If no one is reading something or referencing it in their own work, you could get rid of it to make room for new writing.

That old, unreferenced writing could be compared to an object in Python whose reference count has dropped to 0. Remember that every object in Python has a reference count and a pointer to a type.

The reference count gets increased for a few different reasons. For example, the reference count will increase if you assign it to another variable:

Python

```
numbers = [1, 2, 3]
# Reference count = 1
more_numbers = numbers
# Reference count = 2
```

It will also increase if you pass the object as an argument:

Python

```
total = sum(numbers)
```

As a final example, the reference count will increase if you include the object in a list:

Python

```
matrix = [numbers, numbers, numbers]
```

Python allows you to inspect the current reference count of an object with the `sys` module. You can use `sys.getrefcount(numbers)`, but keep in mind that passing in the object to `getrefcount()` increases the reference count by 1.

In any case, if the object is still required to hang around in your code, its reference count is greater than 0. Once it drops to 0, the object has a specific deallocation function that is called which “frees” the memory so that other objects can use it.

But what does it mean to “free” the memory, and how do other objects use it? Let's jump right into CPython's memory management.

CPython's Memory Management

We're going to dive deep into CPython's memory architecture and algorithms, so buckle up.

As mentioned before, there are layers of abstraction from the physical hardware to CPython. The operating system (OS) abstracts the physical memory and creates a virtual memory layer that applications (including Python) can access.

An OS-specific virtual memory manager carves out a chunk of memory for the Python process. The darker gray boxes in the image below are now owned by the Python process.

Python uses a portion of the memory for internal use and non-object memory. The other portion is dedicated to object storage (your `int`, `dict`, and the like). Note that this was somewhat simplified. If you want the full picture, you can check out the [CPython source code](#), where all this memory management happens.

CPython has an object allocator that is responsible for allocating memory within the object memory area. This object allocator is where most of the magic happens. It gets called every time a new object needs space allocated or deleted.

Typically, the adding and removing of data for Python objects like `list` and `int` doesn't involve too much data at a time. So the design of the allocator is tuned to work well with small amounts of data at a time. It also tries not to allocate memory until it's absolutely required.

The comments in the [source code](#) describe the allocator as “a fast, special-purpose memory allocator for small blocks, to be used on top of a general-purpose malloc.” In this case, `malloc` is C’s library function for memory allocation.

Now we’ll look at CPython’s memory allocation strategy. First, we’ll talk about the 3 main pieces and how they relate to each other.

Arenas are the largest chunks of memory and are aligned on a page boundary in memory. A page boundary is the edge of a fixed-length contiguous chunk of memory that the OS uses. Python assumes the system’s page size is 256 kilobytes.

Within the arenas are pools, which are one virtual memory page (4 kilobytes). These are like the pages in our book analogy. These pools are fragmented into smaller blocks of memory.

All the blocks in a given pool are of the same “size class.” A size class defines a specific block size, given some amount of requested data. The chart below is taken directly from the [source code](#) comments:

Request in bytes	Size of allocated block	Size class idx
1-8	8	0
9-16	16	1
17-24	24	2
25-32	32	3
33-40	40	4
41-48	48	5
49-56	56	6
57-64	64	7
65-72	72	8
...
497-504	504	62
505-512	512	63

For example, if 42 bytes are requested, the data would be placed into a size 48-byte block.

Pools

Pools are composed of blocks from a single size class. Each pool maintains a [double-linked list](#) to other pools of the same size class. In that way, the algorithm can easily find available space for a given block size, even across different pools.

A `usedpools` list tracks all the pools that have some space available for data for each size class. When a given block size is requested, the algorithm checks this `usedpools` list for the list of pools for that block size.

Pools themselves must be in one of 3 states: `used`, `full`, or `empty`. A `used` pool has available blocks for data to be stored. A `full` pool's blocks are all allocated and contain data. An `empty` pool has no data stored and can be assigned any size class for blocks when needed.

A `freepools` list keeps track of all the pools in the `empty` state. But when do empty pools get used?

Assume your code needs an 8-byte chunk of memory. If there are no pools in `usedpools` of the 8-byte size class, a fresh `empty` pool is initialized to store 8-byte blocks. This new pool then gets added to the `usedpools` list so it can be used for future requests.

Say a `full` pool frees some of its blocks because the memory is no longer needed. That pool would get added back to the `usedpools` list for its size class.

You can see now how pools can move between these states (and even memory size classes) freely with this

algorithm.

Blocks

As seen in the diagram above, pools contain a pointer to their “free” blocks of memory. There’s a slight nuance to the way this works. This allocator “strives at all levels (arena, pool, and block) never to touch a piece of memory until it’s actually needed,” according to the comments in the source code.

That means that a pool can have blocks in 3 states. These states can be defined as follows:

- **untouched:** a portion of memory that has not been allocated
- **free:** a portion of memory that was allocated but later made “free” by CPython and that no longer contains relevant data
- **allocated:** a portion of memory that actually contains relevant data

The `freeblock` pointer points to a singly linked list of free blocks of memory. In other words, a list of available places to put data. If more than the available free blocks are needed, the allocator will get some untouched blocks in the pool.

As the memory manager makes blocks “free,” those now free blocks get added to the front of the `freeblock` list. The actual list may not be contiguous blocks of memory, like the first nice diagram. It may look something like the diagram below:

Arenas

Arenas contain pools. Those pools can be used, full, or empty. Arenas themselves don't have as explicit states as pools do though.

Arenas are instead organized into a doubly linked list called `usable_arenas`. The list is sorted by the number of free pools available. The fewer free pools, the closer the arena is to the front of the list.

This means that the arena that is the most full of data will be selected to place new data into. But why not the opposite? Why not place data where there's the most available space?

This brings us to the idea of truly freeing memory. You'll notice that I've been saying "free" in quotes quite a bit. The reason is that when a block is deemed "free", that memory is not actually freed back to the operating system. The Python process keeps it allocated and will use it later for new data. Truly freeing memory returns it to the operating system to use.

Arenas are the only things that can truly be freed. So, it stands to reason that those arenas that are closer to being empty should be allowed to become empty. That way, that chunk of memory can be truly freed, reducing the overall memory footprint of your Python program.

Conclusion

Memory management is an integral part of working with computers. Python handles nearly all of it behind the scenes, for better or for worse.

In this article, you learned:

- What memory management is and why it's important
- How the default Python implementation, CPython, is written in the C programming language
- How the data structures and algorithms work together in CPython's memory management to handle your data

Python abstracts away a lot of the gritty details of working with computers. This gives you the power to work on a higher level to develop your code without the headache of worrying about how and where all those bytes are getting stored.

About Alexander VanTol

Alexander is an avid Pythonista who spends his time on various creative projects involving programming, music, and creative writing.

[» More about Alexander](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

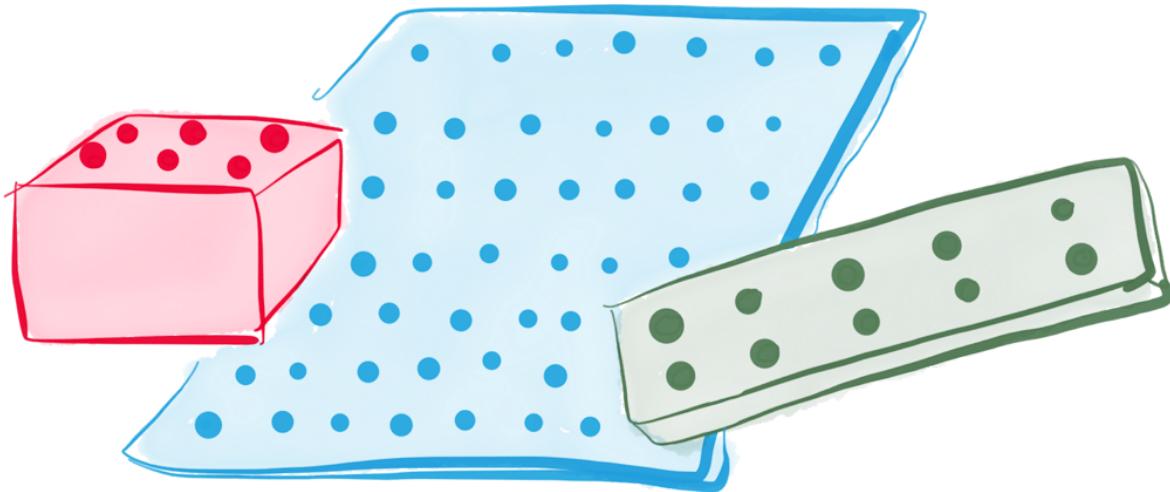
[Aldren](#)

[David](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Model-View-Controller (MVC) Explained – With Legos

by Real Python 31 Comments intermediate web-dev

[Twitter](#) [Share](#) [Email](#)

Table of Contents

- [Legos!](#)
 - [It all starts with a request...](#)
 - [The request reaches the controller...](#)
 - [Those building blocks are known as models...](#)
 - [So the request comes in...](#)
 - [The final product is known as the view...](#)
- [To summarize...](#)
- [From a more technical standpoint](#)
 - [Routes](#)
 - [Models and Controllers](#)
 - [Views](#)
 - [Summary](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

To demonstrate how a web application structured using the [Model-View-Controller](#) pattern (or **MVC**) works in practice, let's take a trip down memory lane...

Free Bonus: [Click here to get access to a free Python OOP Cheat Sheet](#) that points you to the best tutorials, videos, and books to learn more about Object-Oriented Programming with Python.

Legos!

You're ten years old, sitting on your family room floor, and in front of you is a big bucket of Legos. There are Legos of all different shapes and sizes. Some blue, tall, and long. Like a tractor trailer. Some red and almost cube shaped. And some are yellow - big wide planes, like sheets of glass. With all these different types of Legos, there's no telling what you could build.

But surprise, surprise, there's already a **request**. Your older brother runs up and says, "Hey! Build me a spaceship!"

"Alright," you think, "that could actually be pretty cool!" A spaceship it is.

So you get to work. You start pulling out the Legos you think you're going to need. Some big, some small. Different colors for the outside of the spaceship, different colors for the engines. Oh, and different colors for the blaster guns. (You gotta have blaster guns!)

Now that you have all of your **building blocks** in place, it's time to assemble the spaceship. And after a few hours of hard work, you now have in front of you - a spaceship!

You run to find your brother to show him the finished product. "Wow, nice work!", he says. "Huh," he thinks, "I just asked for that a few hours ago, didn't have to do a thing, and there it is. I wish *everything* was that easy."

What if I were to tell you that building a web application is exactly like building with Legos?

It all starts with a *request*...

In the case of the Legos, it was your brother who asked you to build something. In the case of a web app, it's a user entering a URL, requesting to view a certain page.

So your brother is the user.

The request reaches the *controller*...

With the Legos, you are the controller.

The controller is responsible for grabbing all of the necessary **building blocks** and organizing them as necessary.

Those building blocks are known as *models*...

The different types of Legos are the models. You have all different sizes and shapes, and you grab the ones you need to build the spaceship. In a web app, models help the controller retrieve all of the information it needs from the database.

So the request comes in...

The controller (you) receives the request.

It goes to the models (Legos) to retrieve the necessary items.

And now everything is in place to produce the final product.

The final product is known as the *view*...

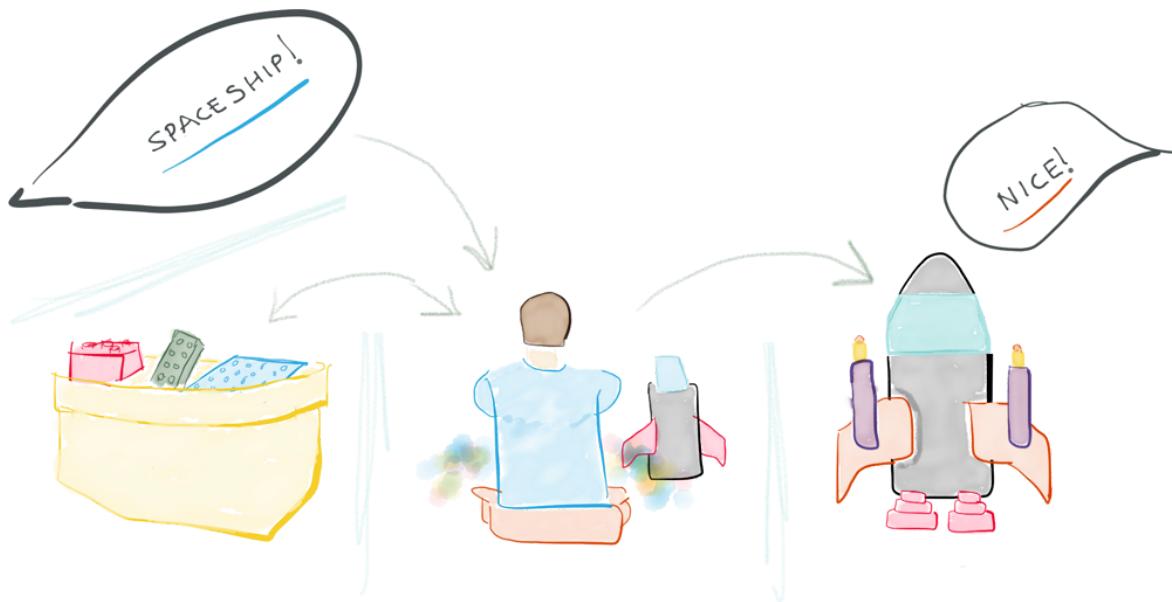
The spaceship is the view. It's the final product that's ultimately shown to the person who made the request (your brother).

In a web application, the view is the final page the user sees in their browser.

To summarize...

When building with Legos:

1. Your brother makes a request that you build a spaceship.
2. You receive the request.
3. You retrieve and organize all the Legos you need to construct the spaceship.
4. You use the Legos to build the spaceship and present the finished spaceship back to your brother.



And in a web app:

1. A user requests to view a page by entering a URL.
2. The **Controller** receives that request.
3. It uses the **Models** to retrieve all of the necessary data, organizes it, and sends it off to the...
4. **View**, which then uses that data to render the final webpage presented to the user in their browser.

From a more technical standpoint

With the MVC functionality summarized, let's dive a bit deeper and see how everything functions on a more technical level.

When you type in a URL in your browser to access a web application, you're making a request to view a certain page within the application. But how does the application know which page to display/render?

When building a web app, you define what are known as **routes**. Routes are, essentially, URL patterns associated with different pages. So when someone enters a URL, behind the scenes, the application tries to match that URL to one of these predefined routes.

So, in fact, there are really *four* major components in play: **routes**, **models**, **views**, and **controllers**.

Routes

Each route is associated with a controller - more specifically, a certain function *within* a controller, known as a **controller action**. So when you enter a URL, the application attempts to find a matching route, and, if it's successful, it calls that route's associated controller action.

Let's look at a basic [Flask](#) route as an example:

Python

```
@app.route('/')
def main_page():
    pass
```

Here we establish the / route associated with the `main_page()` view function.

Models and Controllers

Within the controller action, two main things typically occur: the models are used to retrieve all of the necessary data from a database; and that data is passed to a view, which renders the requested page. The data retrieved via the models is generally added to a data structure (like a list or dictionary), and that structure is what's sent to the view.

Back to our Flask example:

Python

```
@app.route('/')
def main_page():
    """Searches the database for entries, then displays them."""
    db = get_db()
    cur = db.execute('select * from entries order by id desc')
    entries = cur.fetchall()
    return render_template('index.html', entries=entries)
```

Now within the view function, we grab data from the database and perform some basic logic. This returns a list, which we assign to the variable `entries`, that is accessible within the `index.html` template.

Views

Finally, in the view, that structure of data is accessed and the information contained within is used to render the HTML content of the page the user ultimately sees in their browser.

Again, back to our Flask app, we can loop through the `entries`, displaying each one using the Jinja syntax:

HTML

```
{% for entry in entries %}
<li>
  <h2>{{ entry.title }}</h2>
  <div>{{ entry.text|safe }}</div>
</li>
{% else %}
<li><em>No entries yet. Add some!</em></li>
{% endfor %}
```

Summary

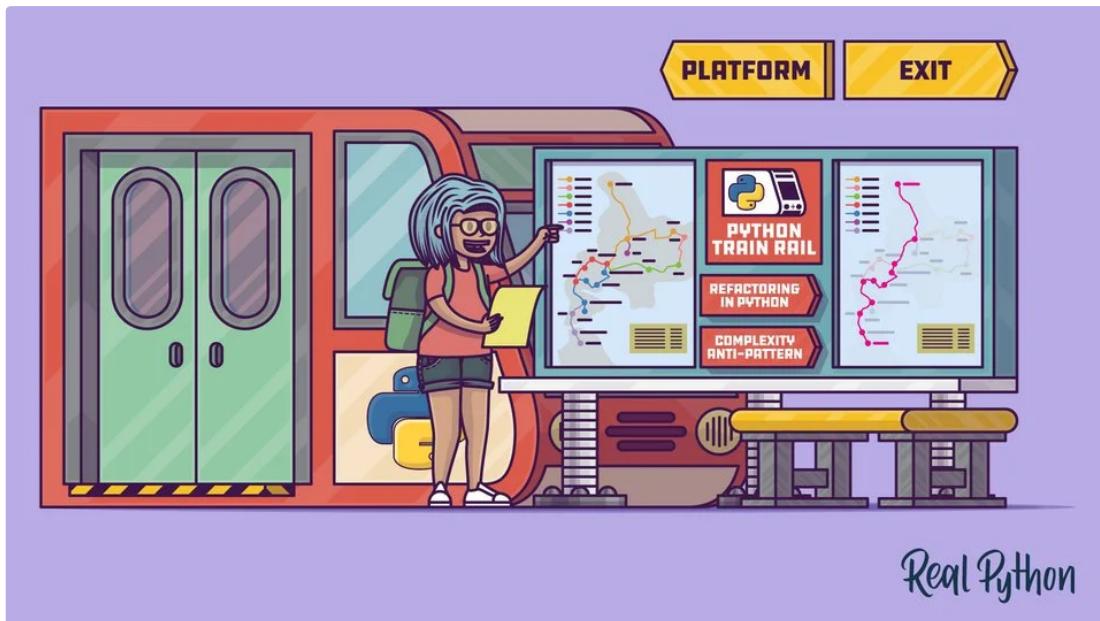
So a more detailed, technical summary of the MVC request process is as follows:

1. A user requests to view a page by entering a URL.
2. The application matches the URL to a predefined **route**.
3. The **controller action** associated with the route is called.
4. The controller action uses the **models** to retrieve all of the necessary data from a database, places the data in an array, and loads a **view**, passing along the data structure.
5. The **view** accesses the structure of data and uses it to render the requested page, which is then presented to the user in their browser.

This is a guest post by Alex Coleman, a coding instructor and consulting web developer.

Keep Learning

Related Tutorial Categories: [intermediate](#) [web-dev](#)



Refactoring Python Applications for Simplicity

by [Anthony Shaw](#) 21 Comments [best-practices](#) [intermediate](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Code Complexity in Python](#)
 - [Metrics for Measuring Complexity](#)
 - [Using wily to Capture and Track Your Projects' Complexity](#)
- [Refactoring in Python](#)
 - [Avoiding Risks With Refactoring: Leveraging Tools and Having Tests](#)
 - [Using rope for Refactoring](#)
 - [Using Visual Studio Code for Refactoring](#)
 - [Using PyCharm for Refactoring](#)
 - [Summary](#)
- [Complexity Anti-Patterns](#)
 - [1. Functions That Should Be Objects](#)
 - [2. Objects That Should Be Functions](#)
 - [3. Converting “Triangular” Code to Flat Code](#)
 - [4. Handling Complex Dictionaries With Query Tools](#)
 - [5. Using attrs and dataclasses to Reduce Code](#)
- [Conclusion](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

Do you want simpler Python code? You always start a project with the best intentions, a clean codebase, and a nice structure. But over time, there are changes to your apps, and things can get a little messy.

If you can write and maintain clean, simple Python code, then it'll save you lots of time in the long term. You can spend less time testing, finding bugs, and making changes when your code is well laid out and simple to follow.

In this tutorial you'll learn:

- How to measure the complexity of Python code and your applications
- How to change your code without breaking it

- What the common issues in Python code that cause extra complexity are and how you can fix them

Throughout this tutorial, I'm going to use the theme of subterranean railway networks to explain complexity because navigating a subway system in a large city can be complicated! Some are well designed, and others seem overly complex.

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Code Complexity in Python

The complexity of an application and its codebase is relative to the task it's performing. If you're writing code for NASA's jet propulsion laboratory (literally [rocket science](#)), then it's going to be complicated.

The question isn't so much, "Is my code complicated?" as, "Is my code more complicated than it needs to be?"

The Tokyo railway network is one of the most extensive and complicated in the world. This is partly because Tokyo is a metropolis of over 30 million people, but it's also because there are 3 networks overlapping each other.



The author of this article getting lost on the Tokyo Metro

There are the Toei and Tokyo Metro rapid-transport networks as well as the Japan Rail East trains going through Central Tokyo. To even the most experienced traveler, navigating central Tokyo can be mind-bogglingly complicated.

Here is a map of the Tokyo railway network to give you some perspective:

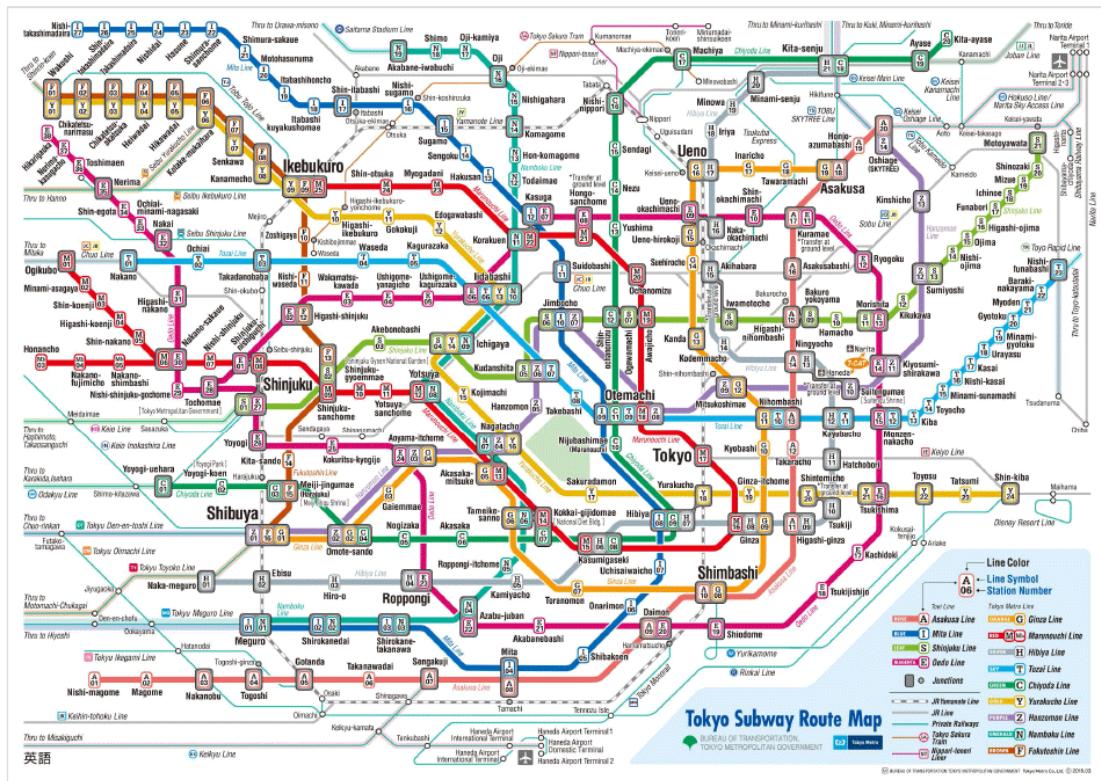
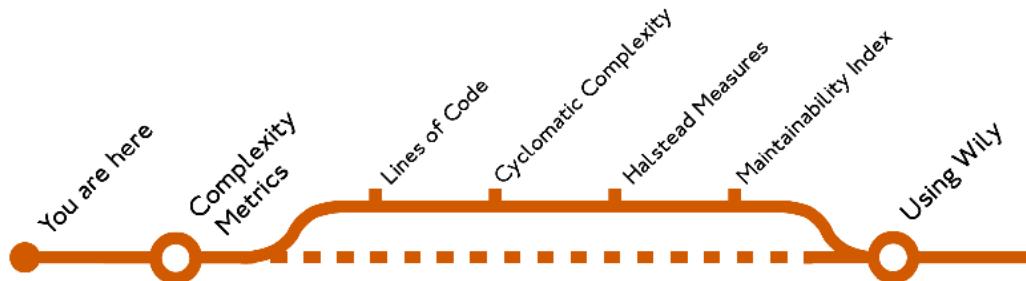


Image: [Tokyo Metro Co.](#)

If your code is starting to look a bit like this map, then this is the tutorial for you.

First, we'll go through 4 metrics of complexity that can give you a scale to measure your relative progress in the mission to make your code simpler:



After you've explored the metrics, you'll learn about a tool called `wily` to automate calculating those metrics.

Metrics for Measuring Complexity

Much time and research have been put into analyzing the complexity of computer software. Overly complex and unmaintainable applications can have a very real cost.

The complexity of software correlates to the quality. Code that is easy to read and understand is more likely to be updated by developers in the future.

Here are some metrics for programming languages. They apply to many languages, not just Python.

Lines of Code

LOC, or Lines of Code, is the crudest measure of complexity. It is debatable whether there is any direct correlation between the lines of code and the complexity of an application, but the indirect correlation is clear. After all, a program with 5 lines is likely simpler than one with 5 million.

When looking at Python metrics, we try to ignore blank lines and lines containing comments.

Lines of code can be calculated using the `wc` command on Linux and Mac OS, where `file.py` is the name of the file you want to measure:

Shell

```
$ wc -l file.py
```

If you want to add the combined lines in a folder by recursively searching for all `.py` files, you can combine `wc` with the `find` command:

Shell

```
$ find . -name \*.py | xargs wc -l
```

For Windows, PowerShell offers a word count command in `Measure-Object` and a recursive file search in `Get-ChildItem`:

Windows PowerShell

```
$ Get-ChildItem -Path *.py -Recurse | Measure-Object -Line
```

In the response, you will see the total number of lines.

Why are lines of code used to quantify the amount of code in your application? The assumption is that a line of code roughly equates to a statement. Lines is a better measure than characters, which would include whitespace.

In Python, we are encouraged to put a single statement on each line. This example is 9 lines of code:

Python

```
1 x = 5
2 value = input("Enter a number: ")
3 y = int(value)
4 if x < y:
5     print(f"{x} is less than {y}")
6 elif x == y:
7     print(f"{x} is equal to {y}")
8 else:
9     print(f"{x} is more than {y}")
```

If you used only lines of code as your measure of complexity, it could encourage the wrong behaviors.

Python code should be easy to read and understand. Taking that last example, you could reduce the number of lines of code to 3:

Python

```
1 x = 5; y = int(input("Enter a number:"))
2 equality = "is equal to" if x == y else "is less than" if x < y else "is more than"
3 print(f"{x} {equality} {y}")
```

But the result is hard to read, and PEP 8 has guidelines around maximum line length and line breaking. You can check out [How to Write Beautiful Python Code With PEP 8](#) for more on PEP 8.

This code block uses 2 Python language features to make the code shorter:

- **Compound statements:** using ;
- **Chained conditional or ternary statements:** `name = value if condition else value if condition2 else value2`

We have reduced the number of lines of code but violated one of the fundamental laws of Python:

“Readability counts”

— Tim Peters, Zen of Python

This shortened code is potentially harder to maintain because code maintainers are humans, and this short code is harder to read. We will explore some more advanced and useful metrics for complexity.

Cyclomatic Complexity

Cyclomatic complexity is the measure of how many independent code paths there are through your application. A path is a sequence of statements that the interpreter can follow to get to the end of the application.

One way to think of cyclomatic complexity and code paths is imagine your code is like a railway network.

For a journey, you may need to change trains to reach your destination. The Lisbon Metropolitan railway system in Portugal is simple and easy to navigate. The cyclomatic complexity for any trip is equal to the number of lines you need to travel on:



Image: [Metro Lisboa](#)

If you needed to get from *Alvalade* to *Anjos*, then you would travel 5 stops on the *linha verde* (green line):

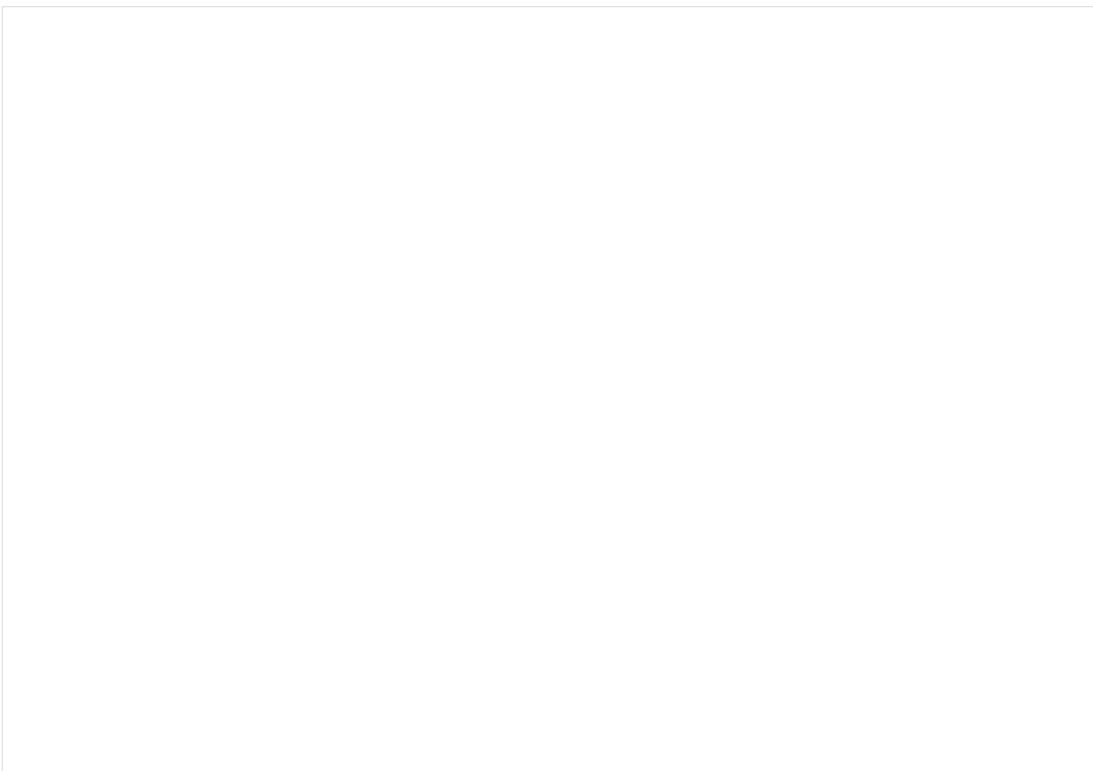


Image: [Metro Lisboa](#)

This trip has a cyclomatic complexity of 1 because you only take 1 train. It's an easy trip. That train is equivalent in this analogy to a code branch.

If you needed to travel from the *Aeroporto* (airport) to sample the [food in the district of Belém](#), then it's a more complicated journey. You would have to change trains at *Alameda* and *Cais do Sodré*:

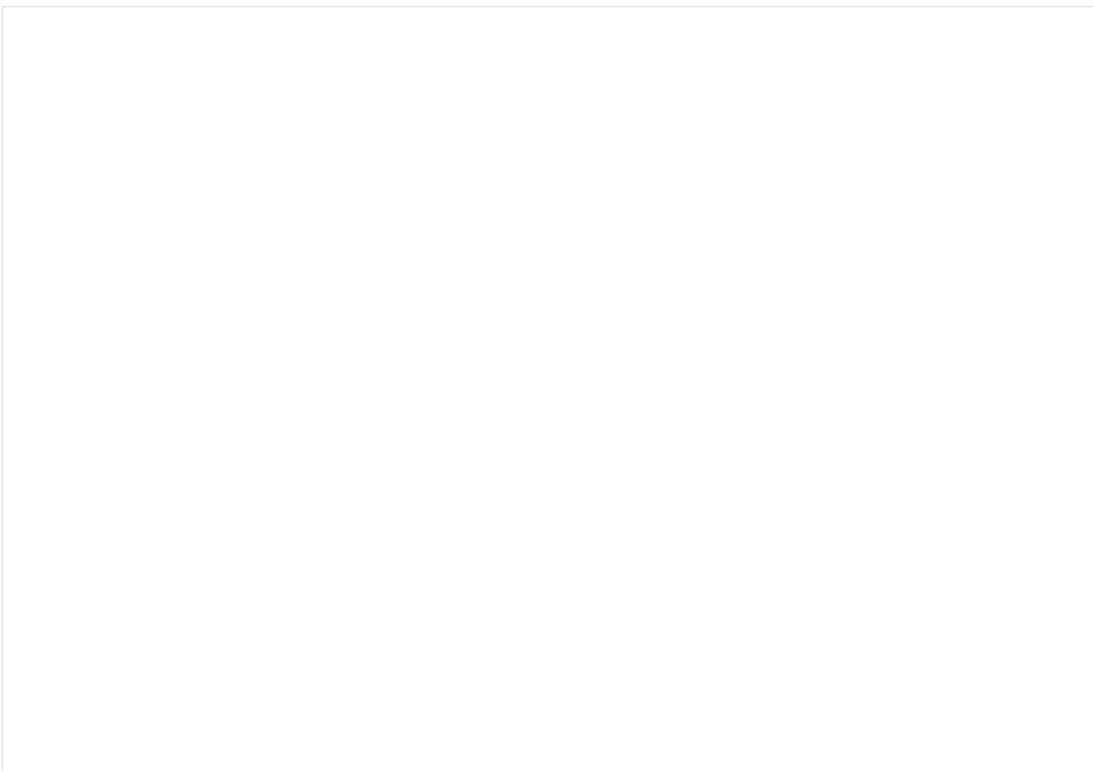


Image: [Metro Lisboa](#)

This trip has a cyclomatic complexity of 3, because you take 3 trains. You might be better off taking a taxi!

Seeing as how you're not navigating Lisbon, but rather writing code, the changes of train line become a branch in execution, like an `if` statement. Let's explore this example:

Python

```
x = 1
```

There is only 1 way this code can be executed, so it has a cyclomatic complexity of 1.

If we add a decision, or branch to the code as an `if` statement, it increases the complexity:

Python

```
x = 1
if x < 2:
    x += 1
```

Even though there is only 1 way this code can be executed, as `x` is a constant, this has a cyclomatic complexity of 2. All of the cyclomatic complexity analyzers will treat an `if` statement as a branch.

This is also an example of overly complex code. The `if` statement is useless as `x` has a fixed value. You could simply refactor this example to the following:

Python

```
x = 2
```

That was a toy example, so let's explore something a little more real.

`main()` has a cyclomatic complexity of 5. I'll comment each branch in the code so you can see where they are:

Python

```
# cyclomatic_example.py
import sys

def main():
    if len(sys.argv) > 1: # 1
        filepath = sys.argv[1]
    else:
        print("Provide a file path")
        exit(1)
    if filepath: # 2
        with open(filepath) as fp: # 3
            for line in fp.readlines(): # 4
                if line != "\n": # 5
                    print(line, end="")

if __name__ == "__main__": # Ignored.
    main()
```

There are certainly ways that code can be refactored into a far simpler alternative. We'll get to that later.

Note: The Cyclomatic Complexity measure was [developed by Thomas J. McCabe, Sr](#) in 1976. You may see it referred to as the **McCabe metric** or **McCabe number**.

In the following examples, we will use the [radon](#) library [from PyPi](#) to calculate metrics. You can install it now:

Shell

```
$ pip install radon
```

To calculate cyclomatic complexity using `radon`, you can save the example into a file called `cyclomatic_example.py` and use `radon` from the command line.

The `radon` command takes 2 main arguments:

1. The type of analysis (cc for cyclomatic complexity)
2. A path to the file or folder to analyze

Execute the radon command with the cc analysis against the `cyclomatic_example.py` file. Adding `-s` will give the cyclomatic complexity in the output:

Shell

```
$ radon cc cyclomatic_example.py -s
cyclomatic_example.py
F 4:0 main - B (6)
```

The output is a little cryptic. Here is what each part means:

- F means function, M means method, and C means class.
- main is the name of the function.
- 4 is the line the function starts on.
- B is the rating from A to F. A is the best grade, meaning the least complexity.
- The number in parentheses, 6, is the cyclomatic complexity of the code.

Halstead Metrics

The Halstead complexity metrics relate to the size of a program's codebase. They were developed by Maurice H. Halstead in 1977. There are 4 measures in the Halstead equations:

- **Operands** are values and names of variables.
- **Operators** are all of the built-in keywords, like `if`, `else`, `for` or `while`.
- **Length (N)** is the number of operators plus the number of operands in your program.
- **Vocabulary (h)** is the number of *unique* operators plus the number of *unique* operands in your a program.

There are then 3 additional metrics with those measures:

- **Volume (V)** represents a product of the **length** and the **vocabulary**.
- **Difficulty (D)** represents a product of half the unique operands and the reuse of operands.
- **Effort (E)** is the overall metric that is a product of **volume** and **difficulty**.

All of this is very abstract, so let's put it in relative terms:

- The effort of your application is highest if you use a lot of operators and unique operands.
- The effort of your application is lower if you use a few operators and fewer variables.

For the `cyclomatic_complexity.py` example, operators and operands both occur on the first line:

Python

```
import sys # import (operator), sys (operand)
```

`import` is an operator, and `sys` is the name of the module, so it's an operand.

In a slightly more complex example, there are a number of operators and operands:

Python

```
if len(sys.argv) > 1:
    ...
```

There are 5 operators in this example:

1. `if`
2. `(`
3. `)`
4. `>`
5. `:`

Furthermore, there are 2 operands:

1. sys.argv

2. 1

Be aware that radon only counts a subset of operators. For example, parentheses are excluded in any calculations.

To calculate the Halstead measures in radon, you can run the following command:

Shell

```
$ radon hal cyclomatic_example.py
cyclomatic_example.py:
    h1: 3
    h2: 6
    N1: 3
    N2: 6
    vocabulary: 9
    length: 9
    calculated_length: 20.264662506490406
    volume: 28.529325012980813
    difficulty: 1.5
    effort: 42.793987519471216
    time: 2.377443751081734
    bugs: 0.009509775004326938
```

Why does radon give a metric for time and bugs?

Halstead theorized that you could estimate the time taken in seconds to code by dividing the effort (E) by 18.

Halstead also stated that the expected number of bugs could be estimated dividing the volume (V) by 3000. Keep in mind this was written in 1977, before Python was even invented! So don't panic and start looking for bugs just yet.

Maintainability Index

The maintainability index brings the McCabe Cyclomatic Complexity and the Halstead Volume measures in a scale roughly between zero and one-hundred.

If you're interested, the original equation is as follows:

In the equation, V is the Halstead volume metric, C is the cyclomatic complexity, and L is the number of lines of code.

If you're as baffled as I was when I first saw this equation, here's it means: it calculates a scale that includes the number of variables, operations, decision paths, and lines of code.

It is used across many tools and languages, so it's one of the more standard metrics. However, there are numerous revisions of the equation, so the exact number shouldn't be taken as fact. radon, wily, and Visual Studio cap the number between 0 and 100.

On the maintainability index scale, all you need to be paying attention to is when your code is getting significantly lower (toward 0). The scale considers anything lower than 25 as **hard to maintain**, and anything over 75 as **easy to maintain**. The Maintainability Index is also referred to as **MI**.

The maintainability index can be used as a measure to get the current maintainability of your application and see if you're making progress as you refactor it.

To calculate the maintainability index from radon, run the following command:

Shell

```
$ radon mi cyclomatic_example.py -s
cyclomatic_example.py - A (87.42)
```

In this result, A is the grade that radon has applied to the number 87.42 on a scale. On this scale, A is most maintainable and F the least.

Using wily to Capture and Track Your Projects' Complexity

wily is an open-source software project for collecting code-complexity metrics, including the ones we've covered so far like Halstead, Cyclomatic, and LOC. wily integrates with Git and can automate the collection of metrics across Git branches and revisions.

The purpose of wily is to give you the ability to see trends and changes in the complexity of your code over time. If you were trying to fine-tune a car or improve your fitness, you'd start off with measuring a baseline and tracking improvements over time.

Installing wily

wily is available [on PyPi](#) and can be installed using pip:

Shell

```
$ pip install wily
```

Once wily is installed, you have some commands available in your command-line:

- **wily build:** iterate through the Git history and analyze the metrics for each file
- **wily report:** see the historical trend in metrics for a given file or folder
- **wily graph:** graph a set of metrics in an HTML file

Building a Cache

Before you can use wily, you need to analyze your [project](#). This is done using the `wily build` command.

For this section of the tutorial, we will analyze the very popular `requests` package, used for talking to HTTP APIs. Because this project is open-source and available on GitHub, we can easily access and download a copy of the source code:

Shell

```
$ git clone https://github.com/requests/requests
$ cd requests
$ ls
AUTHORS.rst      CONTRIBUTING.md    LICENSE        Makefile
Pipfile.lock     _appveyor          docs           pytest.ini
setup.cfg       tests              CODE_OF_CONDUCT.md HISTORY.md
MANIFEST.in      Pipfile           README.md      appveyor.yml
ext             requests          setup.py       tox.ini
```

Note: Windows users should use the PowerShell command prompt for the following examples instead of traditional MS-DOS Command-Line. To start the PowerShell CLI press  +  and type `powershell` then .

You will see a number of folders here, for tests, documentation, and configuration. We're only interested in the source code for the `requests` Python package, which is in a folder called `requests`.

Call the `wily build` command from the cloned source code and provide the name of the source code folder as the first argument:

Shell

```
$ wily build requests
```

This will take a few minutes to analyze, depending on how much CPU power your computer has:

Collecting Data on Your Project

Once you have analyzed the `requests` source code, you can query any file or folder to see key metrics. Earlier in the tutorial, we discussed the following:

- Lines of Code
- Maintainability Index
- Cyclomatic Complexity

Those are the 3 default metrics in `wily`. To see those metrics for a specific file (such as `requests/api.py`), run the following command:

Shell

```
$ wily report requests/api.py
```

`wily` will print a tabular report on the default metrics for each Git commit in reverse date order. You will see the most recent commit at the top and the oldest at the bottom:

Revision	Author	Date	MI	Lines of Code	Cyclomatic Complexity
f37daf2	Nate Prewitt	2019-01-13	100 (0.0)	158 (0)	9 (0)
6dd410f	Ofek Lev	2019-01-13	100 (0.0)	158 (0)	9 (0)
5c1f72e	Nate Prewitt	2018-12-14	100 (0.0)	158 (0)	9 (0)
c4d7680	Matthieu Moy	2018-12-14	100 (0.0)	158 (0)	9 (0)
c452e3b	Nate Prewitt	2018-12-11	100 (0.0)	158 (0)	9 (0)
5a1e738	Nate Prewitt	2018-12-10	100 (0.0)	158 (0)	9 (0)

This tells us that the `requests/api.py` file has:

- 158 lines of code
- A perfect maintainability index of 100
- A cyclomatic complexity of 9

To see other metrics, you first need to know the names of them. You can see this by running the following command:

Shell

```
$ wily list-metrics
```

You will see a list of operators, modules that analyze the code, and the metrics they provide.

To query alternative metrics on the report command, add their names after the filename. You can add as many metrics as you wish. Here's an example with the Maintainability Rank and the Source Lines of Code:

Shell

```
$ wily report requests/api.py maintainability.rank raw.sloc
```

You will see the table now has 2 different columns with the alternative metrics.

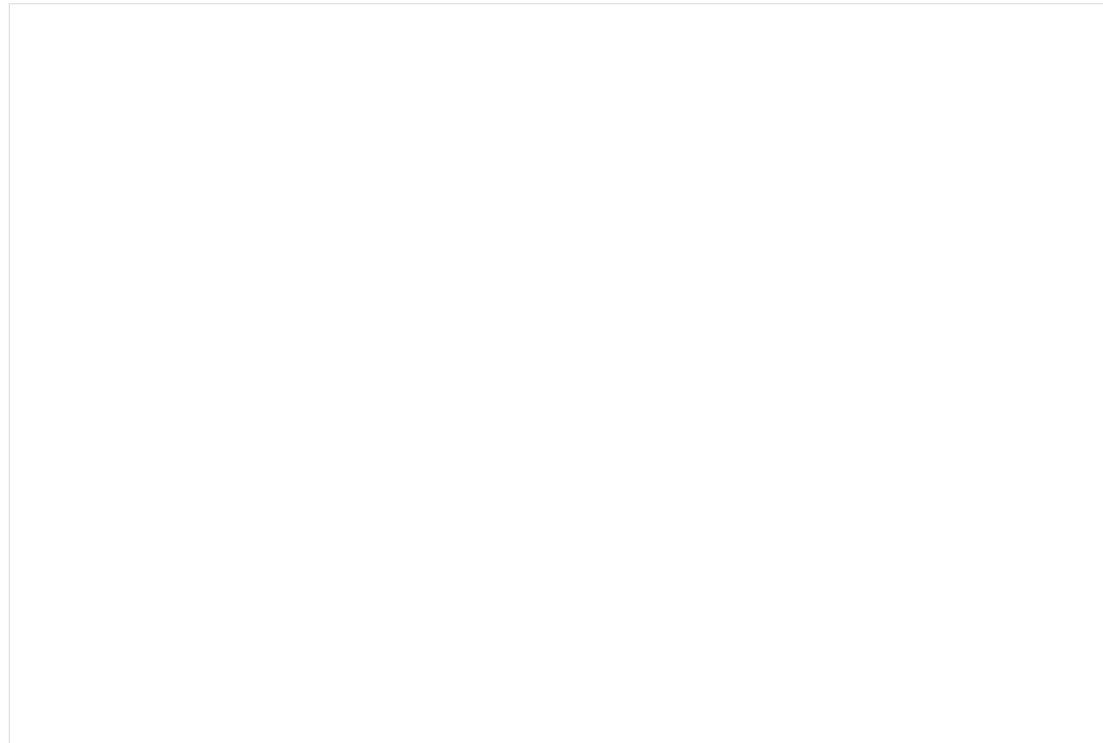
Graphing Metrics

Now that you know the names of the metrics and how to query them on the command line, you can also visualize them in graphs. `wily` supports HTML and interactive charts with a similar interface as the report command:

Shell

```
$ wily graph requests/sessions.py maintainability.mi
```

Your default browser will open with an interactive chart like this:



You can hover over specific data points, and it will show the Git commit message as well as the data.

If you want to save the HTML file in a folder or repository, you can add the `-o` flag with the path to a file:

Shell

```
$ wily graph requests/sessions.py maintainability.mi -o my_report.html
```

There will now be a file called `my_report.html` that you can share with others. This command is ideal for team dashboards.

wily as a pre-commit Hook

`wily` can be configured so that before you commit changes to your project, it can alert you to improvements or degradations in complexity.

`wily` has a `wily diff` command, that compares the last indexed data with the current working copy of a file.

To run a `wily diff` command, provide the names of the files you have changed. For example, if I made some changes to `requests/api.py` you will see the impact on the metrics by running `wily diff` with the file path:

Shell

```
$ wily diff requests/api.py
```

In the response, you will see all of the changed metrics, as well as the functions or classes that have changed for cyclomatic complexity:

The `diff` command can be paired with a tool called `pre-commit`. `pre-commit` inserts a hook into your Git configuration that calls a script every time you run the `git commit` command.

To install `pre-commit`, you can install from PyPI:

Shell

```
$ pip install pre-commit
```

Add the following to a `.pre-commit-config.yaml` in your projects root directory:

YAML

```
repos:
- repo: local
  hooks:
    - id: wily
      name: wily
      entry: wily diff
      verbose: true
      language: python
      additional_dependencies: [wily]
```

Once setting this, you run the `pre-commit install` command to finalize things:

Shell

```
$ pre-commit install
```

Whenever you run the `git commit` command, it will call `wily diff` along with the list of files you've added to your staged changes.

`wily` is a useful utility to baseline the complexity of your code and measure the improvements you make when you start to refactor.

Refactoring in Python

Refactoring is the technique of changing an application (either the code or the architecture) so that it behaves the same way on the outside, but internally has improved. These improvements can be stability, performance, or reduction in complexity.

One of the world's oldest underground railways, the London Underground, started in 1863 with the opening of the Metropolitan line. It had gas-lit wooden carriages hauled by steam locomotives. On the opening of the railway, it was fit for purpose. 1900 brought the invention of the electric railways.

By 1908, the London Underground had expanded to 8 railways. During the Second World War, the London Underground stations were closed to trains and used as air-raid shelters. The modern London Underground carries millions of passengers a day with over 270 stations:



Joint London Underground Railways Map, c. 1908 (Image: [Wikipedia](#))

It's almost impossible to write perfect code the first time, and requirements change frequently. If you would have asked the original designers of the railway to design a network fit for 10 million passengers a day in 2020, they would not design the network that exists today.

Instead, the railway has undergone a series of continuous changes to optimize its operation, design, and layout to match the changes in the city. It has been refactored.

In this section, you'll explore how to safely refactor by leveraging tests and tools. You'll also see how to use the refactoring functionality in [Visual Studio Code](#) and [PyCharm](#):

Avoiding Risks With Refactoring: Leveraging Tools and Having Tests

If the point of refactoring is to improve the internals of an application without impacting the externals, how do you ensure the externals haven't changed?

Before you charge into a major refactoring project, you need to make sure you have a solid test suite for your application. Ideally, that test suite should be mostly automated, so that as you make changes, you see the impact on the user and address it quickly.

If you want to learn more about testing in Python, [Getting Started With Testing in Python](#) is a great place to start.

There is no perfect number of tests to have on your application. But, the more robust and thorough the test suite, the more aggressively you can refactor your code.

The two most common tasks you will perform when doing refactoring are:

- Renaming modules, functions, classes, and methods
- Finding usages of functions, classes, and methods to see where they are called

You can simply do this by hand using **search and replace**, but it is both time consuming and risky. Instead, there are some great tools to perform these tasks.

Using rope for Refactoring

rope is a free Python utility for refactoring Python code. It comes with an [extensive](#) set of APIs for refactoring and renaming components in your Python codebase.

rope can be used in two ways:

1. By using an editor plugin, for [Visual Studio Code](#), [Emacs](#), or [Vim](#)
2. Directly by writing scripts to refactor your application

To use rope as a library, first install rope by executing pip:

Shell

```
$ pip install rope
```

It is useful to work with rope on the REPL so that you can explore the project and see changes in real time. To start, import the Project type and instantiate it with the path to the project:

Python

>>>

```
>>> from rope.base.project import Project  
  
>>> proj = Project('requests')
```

The proj variable can now perform a series of commands, like get_files and get_file, to get a specific file. Get the file api.py and assign it to a variable called api:

Python

>>>

```
>>> [f.name for f in proj.get_files()]  
['structures.py', 'status_codes.py', ..., 'api.py', 'cookies.py']  
  
>>> api = proj.get_file('api.py')
```

If you wanted to rename this file, you could simply rename it on the filesystem. However, any other Python files in your project that imported the old name would now be broken. Let's rename the api.py to new_api.py:

Python

>>>

```
>>> from rope.refactor.rename import Rename  
  
>>> change = Rename(proj, api).get_changes('new_api')  
  
>>> proj.do(change)
```

Running git status, you will see that rope made some changes to the repository:

Shell

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   requests/__init__.py
    deleted:   requests/api.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    requests/.ropeproject/
    requests/new_api.py

no changes added to commit (use "git add" and/or "git commit -a")
```

The three changes made by rope are the following:

1. Deleted `requests/api.py` and created `requests/new_api.py`
2. Modified `requests/__init__.py` to import from `new_api` instead of `api`
3. Created a project folder named `.ropeproject`

To reset the change, run `git reset`.

There are [hundreds of other refactorings](#) that can be done with rope.

Using Visual Studio Code for Refactoring

Visual Studio Code opens up a small subset of the refactoring commands available in rope through its own UI.

You can:

1. Extract variables from a statement
2. Extract methods from a block of code
3. Sort imports into a logical order

Here is an example of using the *Extract methods* command from the command palette:

Using PyCharm for Refactoring

If you use or are considering using PyCharm as a Python editor, it's worth taking note of the powerful refactoring capabilities it has.

You can access all the refactoring shortcuts with the `^Ctrl + T` command on Windows and macOS. The shortcut to access refactoring in Linux is `^Ctrl + ↑Shift + Alt + T`.

Finding Callers and Usages of Functions and Classes

Before you remove a method or class or change the way it behaves, you'll need to know what code depends on it. PyCharm can search for all usages of a method, function, or class within your project.

To access this feature, select a method, class, or variable by right-clicking and select *Find Usages*:

All of the code that uses your search criteria is shown in a panel at the bottom. You can double-click on any item to navigate directly to the line in question.

Using the PyCharm Refactoring Tools

Some of the other refactoring commands include the ability to:

- Extract methods, variables, and constants from existing code
- Extract abstract classes from existing class signatures, including the ability to specify abstract methods
- Rename practically anything, from a variable to a method, file, class, or module

Here is an example of renaming the same `api.py` module you renamed earlier using the `rope` module to `new_api.py`:

The rename command is contextualized to the UI, which makes refactoring quick and simple. It has updated the imports automatically in `__init__.py` with the new module name.

Another useful refactor is the *Change Signature* command. This can be used to add, remove, or rename arguments to a function or method. It will search for usages and update them for you:

You can set default values and also decide how the refactoring should handle the new arguments.

[ⓘ Remove ads](#)

Summary

Refactoring is an important skill for any developer. As you've learned in this chapter, you aren't alone. The tools and IDEs already come with powerful refactoring features to be able to make changes quickly.

Complexity Anti-Patterns

Now that you know how complexity can be measured, how to measure it, and how to refactor your code, it's time to learn 5 common anti-patterns that make code more complex than it need be:

If you can master these patterns and know how to refactor them, you'll soon be on track (pun intended) to a more maintainable Python application.

1. Functions That Should Be Objects

Python supports [procedural programming](#) using functions and also [inheritable classes](#). Both are very powerful and should be applied to different problems.

Take this example of a module for working with images. The logic in the functions has been removed for brevity:

Python

```
# imagerlib.py

def load_image(path):
    with open(path, "rb") as file:
        fb = file.load()
    image = img_lib.parse(fb)
    return image

def crop_image(image, width, height):
    ...
    return image

def get_image_thumbnail(image, resolution=100):
    ...
    return image
```

There are a few issues with this design:

1. It's not clear if `crop_image()` and `get_image_thumbnail()` modify the original `image` variable or create new images. If you wanted to load an image then create both a cropped and thumbnail image, would you have to copy the instance first? You could read the source code in the functions, but you can't rely on every developer doing this.

2. You have to pass the `image` variable as an argument in every call to the image functions.

This is how the calling code might look:

Python

```
from imagelib import load_image, crop_image, get_image_thumbnail

image = load_image('~/face.jpg')
image = crop_image(image, 400, 500)
thumb = get_image_thumbnail(image)
```

Here are some symptoms of code using functions that could be refactored into classes:

- Similar arguments across functions
- Higher number of Halstead h2 **unique operands**
- Mix of mutable and immutable functions
- Functions spread across multiple Python files

Here is a refactored version of those 3 functions, where the following happens:

- `__init__()` replaces `load_image()`.
- `crop()` becomes a class method.
- `get_image_thumbnail()` becomes a property.

The thumbnail resolution has become a class property, so it can be changed globally or on that particular instance:

Python

```
# imagelib.py

class Image(object):
    thumbnail_resolution = 100
    def __init__(self, path):
        ...

    def crop(self, width, height):
        ...

    @property
    def thumbnail(self):
        ...
        return thumb
```

If there were many more image-related functions in this code, the refactoring to a class could make a drastic change. The next consideration would be the complexity of the consuming code.

This is how the refactored example would look:

Python

```
from imagelib import Image

image = Image('~/face.jpg')
image.crop(400, 500)
thumb = image.thumbnail
```

In the resulting code, we have solved the original problems:

- It is clear that `thumbnail` returns a thumbnail since it is a property, and that it doesn't modify the instance.
- The code no longer requires creating new variables for the crop operation.

[Remove ads](#)

2. Objects That Should Be Functions

Sometimes, the reverse is true. There is object-oriented code which would be better suited to a simple function or two.

Here are some tell-tale signs of incorrect use of classes:

- Classes with 1 method (other than `__init__()`)
- Classes that contain only static methods

Take this example of an authentication class:

Python

```
# authenticate.py

class Authenticator(object):
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def authenticate(self):
        ...
        return result
```

It would make more sense to just have a simple function named `authenticate()` that takes `username` and `password` as arguments:

Python

```
# authenticate.py

def authenticate(username, password):
    ...
    return result
```

You don't have to sit down and look for classes that match these criteria by hand: `pylint` comes with a rule that classes should have a minimum of 2 public methods. For more on PyLint and other code quality tools, you can check out [Python Code Quality](#).

To install `pylint`, run the following command in your console:

Shell

```
$ pip install pylint
```

`pylint` takes a number of optional arguments and then the path to one or more files and folders. If you run `pylint` with its default settings, it's going to give a lot of output as `pylint` has a huge number of rules. Instead, you can run specific rules. The `too-few-public-methods` rule id is `R0903`. You can look this up on the [documentation website](#):

Shell

```
$ pylint --disable=all --enable=R0903 requests
*****
Module requests.auth
requests/auth.py:72:0: R0903: Too few public methods (1/2) (too-few-public-methods)
requests/auth.py:100:0: R0903: Too few public methods (1/2) (too-few-public-methods)
*****
Module requests.models
requests/models.py:60:0: R0903: Too few public methods (1/2) (too-few-public-methods)

-----
Your code has been rated at 9.99/10
```

This output tells us that auth.py contains 2 classes that have only 1 public method. Those classes are on lines 72 and 100. There is also a class on line 60 of models.py with only 1 public method.

3. Converting “Triangular” Code to Flat Code

If you were to zoom out on your source code and tilt your head 90 degrees to the right, does the whitespace look flat like Holland or mountainous like the Himalayas? Mountainous code is a sign that your code contains a lot of nesting.

Here's one of the principles in the [Zen of Python](#):

“Flat is better than nested”

— Tim Peters, Zen of Python

Why would flat code be better than nested code? Because nested code makes it harder to read and understand what is happening. The reader has to understand and memorize the conditions as they go through the branches.

These are the symptoms of highly nested code:

- A high cyclomatic complexity because of the number of code branches
- A low Maintainability Index because of the high cyclomatic complexity relative to the number of lines of code

Take this example that looks at the argument data for strings that match the word error. It first checks if the data argument is a list. Then, it iterates over each and checks if the item is a string. If it is a string and the value is "error", then it returns True. Otherwise, it returns False:

Python

```
def contains_errors(data):
    if isinstance(data, list):
        for item in data:
            if isinstance(item, str):
                if item == "error":
                    return True
    return False
```

This function would have a low maintainability index because it is small, but it has a high cyclomatic complexity.

Instead, we can refactor this function by “returning early” to remove a level of nesting and returning False if the value of data is not list. Then using .count() on the list object to count for instances of "error". The return value is then an evaluation that the .count() is greater than zero:

Python

```
def contains_errors(data):
    if not isinstance(data, list):
        return False
    return data.count("error") > 0
```

Another technique for reducing nesting is to leverage list comprehensions. This common pattern of creating a new list, going through each item in a list to see if it matches a criterion, then adding all matches to the new list:

Python

```
results = []
for item in iterable:
    if item == match:
        results.append(item)
```

This code can be replaced with a faster and more efficient list comprehension.

Refactor the last example into a list comprehension and an `if` statement:

Python

```
results = [item for item in iterable if item == match]
```

This new example is smaller, has less complexity, and is more performant.

If your data is not a single dimension list, then you can leverage the `itertools` package in the standard library, which contains functions for creating iterators from data structures. You can use it for chaining iterables together, mapping structures, cycling or repeating over existing iterables.

Itertools also contains functions for filtering data, like `filterfalse()`. For more on Itertools, check out [Itertools in Python 3, By Example](#).

 Remove ads

4. Handling Complex Dictionaries With Query Tools

One of Python's most powerful and widely used core types is the dictionary. It's fast, efficient, scalable, and highly flexible.

If you're new to dictionaries, or think you could leverage them more, you can read [Dictionaries in Python](#) for more information.

It does have one major side-effect: when dictionaries are highly nested, the code that queries them becomes nested too.

Take this example piece of data, a sample of the Tokyo Metro lines you saw earlier:

Python

```
data = {
    "network": {
        "lines": [
            {
                "name.en": "Ginza",
                "name.jp": "銀座線",
                "color": "orange",
                "number": 3,
                "sign": "G"
            },
            {
                "name.en": "Marunouchi",
                "name.jp": "丸ノ内線",
                "color": "red",
                "number": 4,
                "sign": "M"
            }
        ]
    }
}
```

If you wanted to get the line that matched a certain number, this could be achieved in a small function:

Python

```
def find_line_by_number(data, number):
    matches = [line for line in data if line['number'] == number]
    if len(matches) > 0:
        return matches[0]
    else:
        raise ValueError(f"Line {number} does not exist.")
```

Even though the function itself is small, calling the function is unnecessarily complicated because the data is so nested:

Python

>>>

```
>>> find_line_by_number(data["network"]["lines"], 3)
```

There are third party tools for querying dictionaries in Python. Some of the most popular are [JMESPath](#), [glom](#), [asq](#), and [flupy](#).

JMESPath can help with our train network. JMESPath is a querying language designed for JSON, with a plugin available for Python that works with Python dictionaries. To install JMESPath, do the following:

Shell

```
$ pip install jmespath
```

Then open up a Python REPL to explore the JMESPath API, copying in the data dictionary. To get started, import `jmespath` and call `search()` with a query string as the first argument and the data as the second. The query string `"network.lines"` means return `data['network']['lines']`:

Python

>>>

```
>>> import jmespath

>>> jmespath.search("network.lines", data)
[{"name.en": "Ginza", "name.jp": "銀座線",
 "color": "orange", "number": 3, "sign": "G"},
 {"name.en": "Marunouchi", "name.jp": "丸ノ内線",
 "color": "red", "number": 4, "sign": "M"}]
```

When working with lists, you can use square brackets and provide a query inside. The “everything” query is simply `*`. You can then add the name of the attribute inside each matching item to return. If you wanted to get the line number for every line, you could do this:

Python

>>>

```
>>> jmespath.search("network.lines[*].number", data)
[3, 4]
```

You can provide more complex queries, like `a ==` or `<`. The syntax is a little unusual for Python developers, so keep the [documentation](#) handy for reference.

If we wanted to find the line with the number 3, this can be done in a single query:

Python

>>>

```
>>> jmespath.search("network.lines[?number=='3']", data)
[{"name.en": "Ginza", "name.jp": "銀座線", "color": "orange", "number": 3, "sign": "G"}]
```

If we wanted to get the color of that line, you could add the attribute in the end of the query:

Python

>>>

```
>>> jmespath.search("network.lines[?number=='3'].color", data)
['orange']
```

JMESPath can be used to reduce and simplify code that queries and searches through complex dictionaries.

5. Using attrs and dataclasses to Reduce Code

Another goal when refactoring is to simply reduce the amount of code in the codebase while achieving the same behaviors. The techniques shown so far can go a long way to refactoring code into smaller and simpler modules.

Some other techniques require a knowledge of the standard library and some third party libraries.

What Is Boilerplate?

Boilerplate code is code that has to be used in many places with little or no alterations.

Taking our train network as an example, if we were to convert that into types using Python classes and Python 3 type hints, it might look something like this:

Python

```
from typing import List

class Line(object):
    def __init__(self, name_en: str, name_jp: str, color: str, number: int, sign: str):
        self.name_en = name_en
        self.name_jp = name_jp
        self.color = color
        self.number = number
        self.sign = sign

    def __repr__(self):
        return f"<Line {self.name_en} color='{self.color}' number={self.number} sign='{self.sign}'>"

    def __str__(self):
        return f"The {self.name_en} line"

class Network(object):
    def __init__(self, lines: List[Line]):
        self._lines = lines

    @property
    def lines(self) -> List[Line]:
        return self._lines
```

Now, you might also want to add other magic methods, like `__eq__()`. This code is boilerplate. There's no business logic or any other functionality here: we're just copying data from one place to another.

A Case for dataclasses

Introduced into the standard library in Python 3.7, with a backport package for Python 3.6 on PyPI, the `dataclasses` module can help remove a lot of boilerplate for these types of classes where you're just storing data.

To convert the `Line` class above to a dataclass, convert all of the fields to class attributes and ensure they have type annotations:

Python

```
from dataclasses import dataclass

@dataclass
class Line(object):
    name_en: str
    name_jp: str
    color: str
    number: int
    sign: str
```

You can then create an instance of the `Line` type with the same arguments as before, with the same fields, and even `__str__()`, `__repr__()`, and `__eq__()` are implemented:

Python

>>>

```
>>> line = Line('Marunouchi', "丸ノ内線", "red", 4, "M")  
  
>>> line.color  
red  
  
>>> line2 = Line('Marunouchi', "丸ノ内線", "red", 4, "M")  
  
>>> line == line2  
True
```

Dataclasses are a great way to reduce code with a single import that's already available in the standard library. For a full walkthrough, you can checkout [The Ultimate Guide to Data Classes in Python 3.7](#).

Some attrs Use Cases

attrs is a third party package that's been around a lot longer than dataclasses. attrs has a lot more functionality, and it's available on Python 2.7 and 3.4+.

If you are using Python 3.5 or below, attrs is a great alternative to dataclasses. Also, it provides many more features.

The equivalent dataclasses example in attrs would look similar. Instead of using type annotations, the class attributes are assigned with a value from `attrib()`. This can take additional arguments, such as default values and callbacks for validating input:

Python

```
from attr import attrs, attrib  
  
@attrs  
class Line(object):  
    name_en = attrib()  
    name_jp = attrib()  
    color = attrib()  
    number = attrib()  
    sign = attrib()
```

attrs can be a useful package for removing boilerplate code and input validation on data classes.

 [Remove ads](#)

Conclusion

Now that you've learned how to identify and tackle complicated code, think back to the steps you can now take to make your application easier to change and manage:

- Start off by creating a baseline of your project using a tool like `wily`.
- Look at some of the metrics and start with the module that has the lowest maintainability index.
- Refactor that module using the safety provided in tests and the knowledge of tools like PyCharm and `rope`.

Once you follow these steps and the best practices in this article, you can do other exciting things to your application, like adding new features and improving performance.

About Anthony Shaw

Anthony is an avid Pythonista and writes for Real Python. Anthony is a Fellow of the Python Software Foundation and member of the Open-Source Apache Foundation.

[» More about Anthony](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

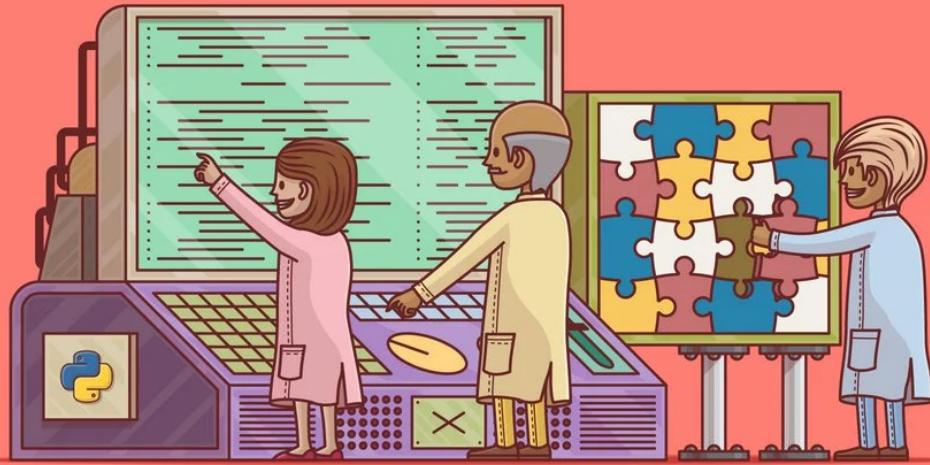
[Aldren](#)

[Geir Arne](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#)



Real Python

Continuous Integration With Python: An Introduction

by Kristijan Ivancic 18 Comments best-practices devops intermediate testing

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [What Is Continuous Integration?](#)
- [Why Should I Care?](#)
- [Core Concepts](#)
 - [Single Source Repository](#)
 - [Automating the Build](#)
 - [Automated Testing](#)
 - [Using an External Continuous Integration Service](#)
 - [Testing in a Staging Environment](#)
- [Your Turn!](#)
 - [Problem Definition](#)
 - [Create a Repo](#)
 - [Set Up a Working Environment](#)
 - [Write a Simple Python Example](#)
 - [Write Unit Tests](#)
 - [Connect to CircleCI](#)
 - [Make Changes](#)
 - [Notifications](#)
- [Next Steps](#)
 - [Git Workflows](#)
 - [Dependency Management and Virtual Environments](#)
 - [Testing](#)
 - [Packaging](#)
 - [Continuous Integration](#)
 - [Continuous Deployment](#)
- [Overview of Continuous Integration Services](#)
- [Conclusion](#)

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Continuous Integration With Python](#)

When writing code on your own, the only priority is making it work. However, working in a team of professional software developers brings a plethora of challenges. One of those challenges is coordinating many people working on the same code.

How do professional teams make dozens of changes per day while making sure everyone is coordinated and nothing is broken? Enter continuous integration!

In this tutorial you'll:

- Learn the core concepts behind continuous integration
- Understand the benefits of continuous integration
- Set up a basic continuous integration system
- Create a simple Python example and connect it to the continuous integration system

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

What Is Continuous Integration?

Continuous integration (CI) is the practice of frequently building and testing each change done to your code automatically and as early as possible. Prolific developer and author Martin Fowler defines CI as follows:

“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.” ([Source](#))

Let's unpack this.

Programming is iterative. The source code lives in a repository that is shared by all members of the team. If you want to work on that product, you must obtain a copy. You will make changes, test them, and integrate them back into the main repo. Rinse and repeat.

Not so long ago, these integrations were big and weeks (or months) apart, causing headaches, wasting time, and losing money. Armed with experience, developers started making minor changes and integrating them more frequently. This reduces the chances of introducing conflicts that you need to resolve later.

After every integration, you need to build the source code. Building means transforming your high-level code into a format your computer knows how to run. Finally, the result is systematically tested to ensure your changes did not introduce errors.

Why Should I Care?

On a personal level, continuous integration is really about how you and your colleagues spend your time.

Using CI, you'll spend less time:

- Worrying about introducing a bug every time you make changes
- Fixing the mess someone else made so you can integrate your code
- Making sure the code works on every machine, operating system, and browser

Conversely, you'll spend more time:

- Solving interesting problems
- Writing awesome code with your team

- Co-creating amazing products that provide value to users

How does that sound?

On a team level, it allows for a better engineering culture, where you deliver value early and often. Collaboration is encouraged, and bugs are caught much sooner. Continuous integration will:

- Make you and your team faster
- Give you confidence that you're building stable software with fewer bugs
- Ensure that your product works on other machines, not just your laptop
- Eliminate a lot of tedious overhead and let you focus on what matters
- Reduce the time spent resolving conflicts (when different people modify the same code)

Core Concepts

There are several key ideas and practices that you need to understand to work effectively with continuous integration. Also, there might be some words and phrases you aren't familiar with but are used often when you're talking about CI. This chapter will introduce you to these concepts and the jargon that comes with them.

Single Source Repository

If you are collaborating with others on a single code base, it's typical to have a shared repository of source code. Every developer working on the project creates a local copy and makes changes. Once they are satisfied with the changes, they merge them back into the central repository.

It has become a standard to use version control systems (VCS) like Git to handle this workflow for you. Teams typically use an external service to host their source code and handle all the moving parts. The most popular are GitHub, BitBucket, and GitLab.

Git allows you to create multiple **branches** of a repository. Each branch is an independent copy of the source code and can be modified without affecting other branches. This is an essential feature, and most teams have a mainline branch (often called a master branch) that represents the current state of the project.

If you want to add or modify code, you should create a copy of the main branch and work in your new, development branch. Once you are done, merge those changes back into the master branch.

Git branching

Version control holds more than just code. Documentation and test scripts are usually stored along with the source code. Some programs look for external files used to configure their parameters and initial settings. Other applications need a database schema. All these files should go into your repository.

If you have never used Git or need a refresher, check out our [Introduction to Git and GitHub for Python Developers](#).

Automating the Build

As previously mentioned, building your code means taking the raw source code, and everything necessary for its execution, and translating it into a format that computers can run directly. Python is an [interpreted language](#), so its "build" mainly revolves around test execution rather than compilation.

Running those steps manually after every small change is tedious and takes valuable time and attention from the actual problem-solving you're trying to do. A big part of continuous integration is automating that process and moving it out of sight (and out of mind).

What does that mean for Python? Think about a more complicated piece of code you have written. If you used a library, package, or framework that doesn't come with the Python standard library (think anything you'll need to

install with pip or conda), Python needs to know about that, so the program knows where to look when it finds commands that it doesn't recognize.

You store a list of those packages in requirements.txt or a Pipfile. These are the dependencies of your code and are necessary for a successful build.

You will often hear the phrase “breaking the build.” When you break the build, it means you introduced a change that rendered the final product unusable. Don’t worry. It happens to everyone, even battle-hardened senior developers. You want to avoid this primarily because it will block everyone else from working.

The whole point of CI is to have everyone working on a known stable base. If they clone a repository that is breaking the build, they will work with a broken version of the code and won’t be able to introduce or test their changes. When you break the build, the top priority is fixing it so everyone can resume work.

Introducing a breaking change to the master branch

When the build is automated, you are encouraged to commit frequently, usually multiple times per day. It allows people to quickly find out about changes and notice if there’s a conflict between two developers. If there are numerous small changes instead of a few massive updates, it’s much easier to locate where the error originated. It will also encourage you to break your work down into smaller chunks, which is easier to track and test.

Automated Testing

Since everyone is committing changes multiple times per day, it’s important to know that your change didn’t break anything else in the code or introduce bugs. In many companies, testing is now a responsibility of every developer. If you write code, you should write tests. At a bare minimum, you should cover every new function with a unit test.

Running tests automatically, with every change committed, is a great way to catch bugs. A failing test automatically causes the build to fail. It will draw your attention to the problems revealed by testing, and the failed build will make you fix the bug you introduced. Tests don’t guarantee that your code is free of bugs, but it does guard against a lot of careless changes.

Automating test execution gives you some peace of mind because you know the server will test your code every time you commit, even if you forgot to do it locally.

Using an External Continuous Integration Service

If something works on your computer, will it work on every computer? Probably not. It’s a cliché excuse and a sort of inside joke among developers to say, “Well, it worked on my machine!” Making the code work locally is not the end of your responsibility.

To tackle this problem, most companies use an external service to handle integration, much like using GitHub for hosting your source code repository. External services have servers where they build code and run tests. They act as monitors for your repository and stop anyone from merging to the master branch if their changes break the build.

Merging changes triggers the CI server

There are many such services out there, with various features and pricing. Most have a free tier so that you can experiment with one of your repositories. You will use a service called CircleCI in an example later in the tutorial.

Testing in a Staging Environment

A production environment is where your software will ultimately run. Even after successfully building and testing your application, you can't be sure that your code will work on the target computer. That's why teams deploy the final product in an environment that mimics the production environment. Once you are sure everything works, the application is deployed in the production environment.

Note: This step is more relevant to application code than library code. Any Python libraries you write still need to be tested on a build server, to ensure they work in environments different from your local computer.

You will hear people talking about this **clone** of the production environment using terms like development environment, staging environment, or testing environment. It's common to use abbreviations like DEV for the development environment and PROD for the production environment.

The development environment should replicate production conditions as closely as possible. This setup is often called **DEV/PROD parity**. Keep the environment on your local computer as similar as possible to the DEV and PROD environments to minimize anomalies when deploying applications.

We mention this to introduce you to the vocabulary, but continuously deploying software to DEV and PROD is a whole other topic. The process is called, unsurprisingly, continuous deployment (CD). You can find more resources about it in the [Next Steps](#) section of this article.

Your Turn!

The best way to learn is by doing. You now understand all the essential practices of continuous integration, so it's time to get your hands dirty and create the whole chain of steps necessary to use CI. This chain is often called a **CI pipeline**.

This is a hands-on tutorial, so fire up your editor and get ready to work through these steps as you read!

We assume that you know the basics of Python and Git. We will use [Github](#) as our hosting service and [CircleCI](#) as our external continuous integration service. If you don't have accounts with these services, go ahead and register. Both of these have free tiers!

Problem Definition

Remember, your focus here is adding a new tool to your utility belt, continuous integration. For this example, the Python code itself will be straightforward. You want to spend the bulk of your time internalizing the steps of building a pipeline, instead of writing complicated code.

Imagine your team is working on a simple calculator app. Your task is to write a library of basic mathematical functions: addition, subtraction, multiplication, and division. You don't care about the actual application, because that's what your peers will be developing, using functions from your library.

Create a Repo

Log in to your GitHub account, create a new repository and call it *CalculatorLibrary*. Add a README and .gitignore, then clone the repository to your local machine. If you need more help with this process, have a look at GitHub's [walkthrough](#) on creating a new repository.

Set Up a Working Environment

For others (and the CI server) to replicate your working conditions, you need to set up an environment. Create a virtual environment somewhere outside your repo and activate it:

Shell

```
$ # Create virtual environment
$ python3 -m venv calculator

$ # Activate virtual environment (Mac and Linux)
$ . calculator/bin/activate
```

The previous commands work on macOS and Linux. If you are a Windows user, check the Platforms table in the [official documentation](#). This will create a directory that contains a Python installation and tell the interpreter to use it. Now we can install packages knowing that it will not influence your system's default Python installation.

Write a Simple Python Example

Create a new file called `calculator.py` in the top-level directory of your repository, and copy the following code:

Python

```
"""
Calculator library containing basic math operations.
"""

def add(first_term, second_term):
```

```
def add(first_term, second_term):
    return first_term + second_term

def subtract(first_term, second_term):
    return first_term - second_term
```

This is a bare-bones example containing two of the four functions we will be writing. Once we have our CI pipeline up and running, you will add the remaining two functions.

Go ahead and commit those changes:

Shell

```
$ # Make sure you are in the correct directory
$ cd CalculatorLibrary
$ git add calculator.py
$ git commit -m "Add functions for addition and subtraction"
```

Your *CalculatorLibrary* folder should have the following files right now:

```
CalculatorLibrary/
|
├── .git
├── .gitignore
├── README.md
└── calculator.py
```

Great, you have completed one part of the required functionality. The next step is adding tests to make sure your code works the way it's supposed to.

Write Unit Tests

You will test your code in two steps.

The first step involves linting—running a program, called a linter, to analyze code for potential errors. [flake8](#) is commonly used to check if your code conforms to the standard Python coding style. Linting makes sure your code is easy to read for the rest of the Python community.

The second step is unit testing. A unit test is designed to check a single function, or unit, of code. Python comes with a standard unit testing library, but other libraries exist and are very popular. This example uses [pytest](#).

A standard practice that goes hand in hand with testing is calculating code coverage. Code coverage is the percentage of source code that is “covered” by your tests. pytest has an extension, `pytest-cov`, that helps you understand your code coverage.

These are external dependencies, and you need to install them:

Shell

```
$ pip install flake8 pytest pytest-cov
```

These are the only external packages you will use. Make sure to store those dependencies in a `requirements.txt` file so others can replicate your environment:

Shell

```
$ pip freeze > requirements.txt
```

To run your linter, execute the following:

Shell

```
$ flake8 --statistics
./calculator.py:3:1: E302 expected 2 blank lines, found 1
./calculator.py:6:1: E302 expected 2 blank lines, found 1
2      E302 expected 2 blank lines, found 1
```

The `--statistics` option gives you an overview of how many times a particular error happened. Here we have two PEP 8 violations, because `flake8` expects two blank lines before a function definition instead of one. Go ahead and add an empty line before each functions definition. Run `flake8` again to check that the error messages no longer appear.

Now it's time to write the tests. Create a file called `test_calculator.py` in the top-level directory of your repository and copy the following code:

Python

```
"""
Unit tests for the calculator library
"""

import calculator

class TestCalculator:

    def test_addition(self):
        assert 4 == calculator.add(2, 2)

    def test_subtraction(self):
        assert 2 == calculator.subtract(4, 2)
```

These tests make sure that our code works as expected. It is far from extensive because you haven't tested for potential misuse of your code, but keep it simple for now.

The following command runs your test:

Shell

```
$ pytest -v --cov
collected 2 items

test_calculator.py::TestCalculator::test_addition PASSED [50%]

test_calculator.py::TestCalculator::test_subtraction PASSED [100%]

----- coverage: platform darwin, python 3.6.6-final-0 -----
Name                  Stmts   Miss  Cover
-----
calculator.py           4       0   100%
test_calculator.py      6       0   100%
/Users/kristijan.ivancic/code/learn/__init__.py      0       0   100%
-----
TOTAL                 10      0   100%
```

`pytest` is excellent at test discovery. Because you have a file with the prefix `test`, `pytest` knows it will contain unit tests for it to run. The same principles apply to the class and method names inside the file.

The `-v` flag gives you a nicer output, telling you which tests passed and which failed. In our case, both tests passed. The `--cov` flag makes sure `pytest-cov` runs and gives you a code coverage report for `calculator.py`.

You have completed the preparations. Commit the test file and push all those changes to the master branch:

Shell

```
$ git add test_calculator.py
$ git commit -m "Add unit tests for calculator"
$ git push
```

At the end of this section, your *CalculatorLibrary* folder should have the following files:

```
CalculatorLibrary/
|
├── .git
├── .gitignore
├── README.md
├── calculator.py
├── requirements.txt
└── test_calculator.py
```

Excellent, both your functions are tested and work correctly.

Connect to CircleCI

At last, you are ready to set up your continuous integration pipeline!

CircleCI needs to know how to run your build and expects that information to be supplied in a particular format. It requires a `.circleci` folder within your repo and a configuration file inside it. A configuration file contains instructions for all the steps that the build server needs to execute. CircleCI expects this file to be called `config.yml`.

A `.yml` file uses a data serialization language, YAML, and it has its own [specification](#). The goal of YAML is to be human readable and to work well with modern programming languages for common, everyday tasks.

In a YAML file, there are three basic ways to represent data:

- Mappings (key-value pairs)
- Sequences (lists)
- Scalars (strings or numbers)

It is very simple to read:

- Indentation may be used for structure.
- Colons separate key-value pairs.
- Dashes are used to create lists.

Create the `.circleci` folder in your repo and a `config.yml` file with the following content:

YAML

```
# Python CircleCI 2.0 configuration file
version: 2
jobs:
  build:
    docker:
      - image: circleci/python:3.7

    working_directory: ~/repo

    steps:
      # Step 1: obtain repo from GitHub
      - checkout
      # Step 2: create virtual env and install dependencies
      - run:
          name: install dependencies
          command: |
            python3 -m venv venv
            . venv/bin/activate
            pip install -r requirements.txt
      # Step 3: run linter and tests
      - run:
          name: run tests
          command: |
            . venv/bin/activate
            flake8 --exclude=venv* --statistics
            pytest -v --cov=calculator
```

Let's go back in time a bit.

Remember the problem programmers face when something works on their laptop but nowhere else? Before, developers used to create a program that isolates a part of the computer's physical resources (memory, hard drive, and so on) and turns them into a **virtual machine**.

A virtual machine pretends to be a whole computer on its own. It would even have its own operating system. On that operating system, you deploy your application or install your library and test it.

Virtual machines take up a lot of resources, which sparked the invention of containers. The idea is analogous to shipping containers. Before shipping containers were invented, manufacturers had to ship goods in a wide variety of sizes, packaging, and modes (trucks, trains, ships).

By standardizing the shipping container, these goods could be transferred between different shipping methods without any modification. The same idea applies to software containers.

Containers are a lightweight unit of code and its runtime dependencies, packaged in a standardized way, so they can quickly be plugged in and run on the Linux OS. You don't need to create a whole virtual operating system, as you would with a virtual machine.

Containers only replicate parts of the operating system they need in order to work. This reduces their size and gives them a big performance boost.

Docker is currently the leading container platform, and it's even able to run Linux containers on Windows and macOS. To create a Docker container, you need a Docker image. Images provide blueprints for containers much like classes provide blueprints for objects. You can read more about Docker in their [Get Started](#) guide.

CircleCI maintains [pre-built Docker images](#) for several programming languages. In the above configuration file, you have specified a Linux image that has Python already installed. That image will create a container in which everything else happens.

Let's look at each line of the configuration file in turn:

1. **version:** Every config.yml starts with the CircleCI version number, used to issue warnings about breaking changes.
2. **jobs:** Jobs represent a single execution of the build and are defined by a collection of steps. If you have only one job, it must be called `build`.
3. **build:** As mentioned before, `build` is the name of your job. You can have multiple jobs, in which case they need to have unique names.
4. **docker:** The steps of a job occur in an environment called an executor. The common executor in CircleCI is a [Docker container](#). It is a [cloud-hosted](#) execution environment but other options exist, like a macOS environment.
5. **image:** A Docker image is a file used to create a running Docker container. We are using an image that has Python 3.7 preinstalled.
6. **working_directory:** Your repository has to be checked out somewhere on the build server. The working directory represents the file path where the repository will be stored.
7. **steps:** This key marks the start of a list of steps to be performed by the build server.
8. **checkout:** The first step the server needs to do is check the source code out to the working directory. This is performed by a special step called `checkout`.
9. **run:** Executing command-line programs or commands is done inside the `command` key. The actual shell commands will be nested within.
10. **name:** The CircleCI user interface shows you every build step in the form of an expandable section. The title of the section is taken from the value associated with the `name` key.
11. **command:** This key represents the command to run via the shell. The `|` symbol specifies that what follows is a

literal set of commands, one per line, exactly like you'd see in a shell/bash script.

You can read the [CircleCI configuration reference](#) document for more information.

Our pipeline is very simple and consists of 3 steps:

1. Checking out the repository
2. Installing the dependencies in a virtual environment
3. Running the linter and tests while inside the virtual environment

We now have everything we need to start our pipeline. Log in to your CircleCI account and click on *Add Projects*. Find your *CalculatorLibrary* repo and click *Set Up Project*. Select Python as your language. Since we already have a *config.yml*, we can skip the next steps and click *Start building*.

CircleCI will take you to the execution dashboard for your job. If you followed all the steps correctly, you should see your job succeed.

The final version of your *CalculatorLibrary* folder should look like this:

```
calculatorRepository/
|
├── .circleci
├── .git
├── .gitignore
├── README.md
├── calculator.py
├── requirements.txt
└── test_calculator.py
```

Congratulations! You have created your first continuous integration pipeline. Now, every time you push to the master branch, a job will be triggered. You can see a list of your current and past jobs by clicking on *Jobs* in the CircleCI sidebar.

Make Changes

Time to add multiplication to our calculator library.

This time, we will first add a unit test without writing the function. Without the code, the test will fail, which will also fail the CircleCI job. Add the following code to the end of your *test_calculator.py*:

Python

```
def test_multiplication(self):
    assert 100 == calculator.multiply(10, 10)
```

Push the code to the master branch and see the job fail in CircleCI. This shows that continuous integration works and watches your back if you make a mistake.

Now add the code to *calculator.py* that will make the test pass:

Python

```
def multiply(first_term, second_term):
    return first_term * second_term
```

Make sure there are two empty spaces between the multiplication function and the previous one, or else your code will fail the linter check.

The job should be successful this time. This workflow of writing a failing test first and then adding the code to pass

the test is called [test driven development](#) (TDD). It's a great way to work because it makes you think about your code structure in advance.

Now try it on your own. Add a test for the division function, see it fail, and write the function to make the test pass.

Notifications

When working on big applications that have a lot of moving parts, it can take a while for the continuous integration job to run. Most teams set up a notification procedure to let them know if one of their jobs fail. They can continue working while waiting for the job to run.

The most popular options are:

- Sending an email for each failed build
- Sending failure notifications to a Slack channel
- Displaying failures on a dashboard visible to everyone

By default, CircleCI should send you an email when a job fails.

Next Steps

You have understood the basics of continuous integration and practiced setting up a pipeline for a simple Python program. This is a big step forward in your journey as a developer. You might be asking yourself, “What now?”

To keep things simple, this tutorial skimmed over some big topics. You can grow your skill set immensely by spending some time going more in-depth into each subject. Here are some topics you can look into further.

Git Workflows

There is much more to Git than what you used here. Each developer team has a workflow tailored to their specific needs. Most of them include branching strategies and something called **peer review**. They make changes on branches separate from the `master` branch. When you want to merge those changes with `master`, other developers must first look at your changes and approve them before you're allowed to merge.

Note: If you want to learn more about different workflows teams use, have a look at the tutorials on [GitHub](#) and [BitBucket](#).

If you want to sharpen your Git skills, we have an article called [Advanced Git Tips for Python Developers](#).

Dependency Management and Virtual Environments

Apart from `virtualenv`, there are other popular package and environment managers. Some of them deal with just virtual environments, while some handle both package installation and environment management. One of them is Conda:

“Conda is an open source package management system and environment management system that runs on Windows, macOS, and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was designed for Python programs, but it can package and distribute software for any language.” ([Source](#))

Another option is `Pipenv`, a younger contender that is rising in popularity among application developers. Pipenv brings together `pip` and `virtualenv` into a single tool and uses a `Pipfile` instead of `requirements.txt`. Pipfiles offer deterministic environments and more security. This introduction doesn't do it justice, so check out [Pipenv: A Guide to the New Python Packaging Tool](#).

Testing

Simple unit tests with `pytest` are only the tip of the iceberg. There's a whole world out there to explore! Software can be tested on many levels, including integration testing, acceptance testing, regression testing, and so forth. To take your knowledge of testing Python code to the next level, head over to [Getting Started With Testing in Python](#).

Packaging

In this tutorial, you started to build a library of functions for other developers to use in their project. You need to package that library into a format that is easy to distribute and install using, for example pip.

Creating an installable package requires a different layout and some additional files like `__init__.py` and `setup.py`. Read [Python Application Layouts: A Reference](#) for more information on structuring your code.

To learn how to turn your repository into an installable Python package, read [Packaging Python Projects](#) by the [Python Packaging Authority](#).

Continuous Integration

You covered all the basics of CI in this tutorial, using a simple example of Python code. It's common for the final step of a CI pipeline to create a **deployable artifact**. An artifact represents a finished, packaged unit of work that is ready to be deployed to users or included in complex products.

For example, to turn your calculator library into a deployable artifact, you would organize it into an installable package. Finally, you would add a step in CircleCI to package the library and store that artifact where other processes can pick it up.

For more complex applications, you can create a workflow to schedule and connect multiple CI jobs into a single execution. Feel free to explore the [CircleCI documentation](#).

Continuous Deployment

You can think of continuous deployment as an extension of CI. Once your code is tested and built into a deployable artifact, it is deployed to production, meaning the live application is updated with your changes. One of the goals is to minimize lead time, the time elapsed between writing a new line of code and putting it in front of users.

Note: To add a bit of confusion to the mix, the acronym CD is not unique. It can also mean Continuous Delivery, which is almost the same as continuous deployment but has a manual verification step between integration and deployment. You can integrate your code at any time but have to push a button to release it to the live application.

Most companies use CI/CD in tandem, so it's worth your time to learn more about [Continuous Delivery/Deployment](#).

Overview of Continuous Integration Services

You have used CircleCI, one of the most popular continuous integration services. However, this is a big market with a lot of strong contenders. CI products fall into two basic categories: remote and self-hosted services.

[Jenkins](#) is the most popular self-hosted solution. It is open-source and flexible, and the community has developed a lot of extensions.

In terms of remote services, there are many popular options like [TravisCI](#), [CodeShip](#), and [Semaphore](#). Big enterprises often have their custom solutions, and they sell them as a service, such as [AWS CodePipeline](#), [Microsoft Team Foundation Server](#), and Oracle's [Hudson](#).

Which option you choose depends on the platform and features you and your team need. For a more detailed breakdown, have a look at [Best CI Software](#) by G2Crowd.

Conclusion

With the knowledge from this tutorial under your belt, you can now answer the following questions:

- What is continuous integration?
- Why is continuous integration important?
- What are the core practices of continuous integration?
- How can I set up continuous integration for my Python project?

You have acquired a programming superpower! Understanding the philosophy and practice of continuous integration will make you a valuable member of any team. Awesome work!

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Continuous Integration With Python](#)

About Kristijan Ivancic

Hey, I'm Kristijan! I'm a CV/ML engineer and member of the Real Python tutorial team.

[» More about Kristijan](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Brad

Joanna

Keep Learning

Related Tutorial Categories: [best-practices](#) [devops](#) [intermediate](#) [testing](#)

Recommended Video Course: [Continuous Integration With Python](#)



Real Python

How to Publish an Open-Source Python Package to PyPI

by Geir Arne Hjelle 24 Comments best-practices intermediate tools

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [A Small Python Package](#)
 - [Using the Real Python Reader](#)
 - [A Quick Look at the Code](#)
 - [Different Ways of Calling a Package](#)
- [Preparing Your Package for Publication](#)
 - [Naming Your Package](#)
 - [Configuring Your Package](#)
 - [Documenting Your Package](#)
 - [Versioning Your Package](#)
 - [Adding Files to Your Package](#)
- [Publishing to PyPI](#)
 - [Building Your Package](#)
 - [Testing Your Package](#)
 - [Uploading Your Package](#)
 - [pip install Your Package](#)
- [Other Useful Tools](#)
 - [Virtual Environments](#)
 - [Cookiecutter](#)
 - [Flit](#)
 - [Poetry](#)
- [Conclusion](#)



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Publish Your Own Python Package to PyPI](#)

Python is famous for coming with [batteries included](#). Sophisticated capabilities are available in the standard library. You can find modules for working with [sockets](#), parsing [CSV](#), [JSON](#), and [XML](#) files, and [working with files and file paths](#).

However great the [packages](#) included with Python are, there are many fantastic projects available outside the standard library. These are most often hosted at the [Python Packaging Index](#) (PyPI), historically known as the [Cheese Shop](#). At PyPI, you can find everything from [Hello World](#) to advanced [deep learning libraries](#).

In this tutorial, you'll cover how to **upload your own package to PyPI**. While getting your project published is easier than it used to be, there are still a few steps involved.

You'll learn how to:

- Prepare your Python package for publication
- Think about versioning
- Upload your package to PyPI

Throughout this tutorial, we'll use a simple example project: a reader package that can be used to read *Real Python* tutorials. The first section introduces this project.

Free Bonus: [Click here to get access to a chapter from Python Tricks: The Book](#) that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

A Small Python Package

This section will describe a small Python package that we'll use as an example that can be published to PyPI. If you already have a package that you want to publish, feel free to skim this section and join up again at the [beginning of the next section](#).

The package that we'll use is called `reader` and is an application that can be used to download and read *Real Python* articles. If you want to follow along, you can get the full source code from [our GitHub repository](#).

Note: The source code as shown and explained below is a simplified, but fully functional, version of the *Real Python* feed reader. Compared to the package published on [PyPI](#) and [GitHub](#), this version lacks some error handling and extra options.

First, have a look at the directory structure of `reader`. The package lives completely inside a directory that is also named `reader`:

```
reader/
|
├── reader/
│   ├── config.txt
│   ├── feed.py
│   ├── __init__.py
│   ├── __main__.py
│   └── viewer.py
|
├── tests/
│   ├── test_feed.py
│   └── test_viewer.py
|
└── MANIFEST.in
└── README.md
└── setup.py
```

The source code of the package is in a `reader` subdirectory together with a configuration file. There are a few tests in a separate subdirectory. The tests will not be covered here, but you can find them in the [GitHub repository](#). To learn more about testing, see Anthony Shaw's great tutorial on [Getting Started With Testing in Python](#).

If you're working with your own package, you may use a different structure or have other files in your package directory. Our [Python Application Layouts](#) reference discusses several different options. The instructions in this

guide will work independently of the [layout](#) you use.

In the rest of this section, you'll see how the reader package works. In the [next section](#), you'll get a closer look at the special files, including `setup.py`, `README.md`, and `MANIFEST.in`, that are needed to publish your package.

Using the Real Python Reader

reader is a very basic [web feed](#) reader that can download the latest Real Python articles from the [Real Python feed](#).

Here is an example of using the reader to get the list of the latest articles:

Shell

```
$ python -m reader
The latest tutorials from Real Python (https://realpython.com/)
  0 How to Publish an Open-Source Python Package to PyPI
  1 Python "while" Loops (Indefinite Iteration)
  2 Writing Comments in Python (Guide)
  3 Setting Up Python for Machine Learning on Windows
  4 Python Community Interview With Michael Kennedy
  5 Practical Text Classification With Python and Keras
  6 Getting Started With Testing in Python
  7 Python, Boto3, and AWS S3: Demystified
  8 Python's range() Function (Guide)
  9 Python Community Interview With Mike Grouchy
 10 How to Round Numbers in Python
 11 Building and Documenting Python REST APIs With Flask and Connexion – Part 2
 12 Splitting, Concatenating, and Joining Strings in Python
 13 Image Segmentation Using Color Spaces in OpenCV + Python
 14 Python Community Interview With Mahdi Yusuf
 15 Absolute vs Relative Imports in Python
 16 Top 10 Must-Watch PyCon Talks
 17 Logging in Python
 18 The Best Python Books
 19 Conditional Statements in Python
```

Notice that each article is numbered. To read one particular article, you use the same command but include the number of the article as well. For instance, to read [How to Publish an Open-Source Python Package to PyPI](#), you add `0` to the command:

Shell

```
$ python -m reader 0
# How to Publish an Open-Source Python Package to PyPI

Python is famous for coming with batteries included. Sophisticated
capabilities are available in the standard library. You can find modules
for working with sockets, parsing CSV, JSON, and XML files, and
working with files and file paths.

However great the packages included with Python are, there are many
fantastic projects available outside the standard library. These are
most often hosted at the Python Packaging Index (PyPI), historically
known as the Cheese Shop. At PyPI, you can find everything from Hello
World to advanced deep learning libraries.

[... The full text of the article ...]
```

This prints the full article to the console using the [Markdown](#) text format.

Note: `python -m` is used to run a [library module or package instead of a script](#). If you run a package, the contents of the file `__main__.py` will be executed. See [Different Ways of Calling a Package](#) for more info.

By changing the article number, you can read any of the available articles.

A Quick Look at the Code

The details of how reader works are not important for the purpose of this tutorial. However, if you are interested in seeing the implementation, you can expand the sections below. The package consists of five files:

Seeing the implementation, you can expand the sections below. The package consists of five files.

__main__.py

Show/Hide

__init__.py

Show/Hide

feed.py

Show/Hide

viewer.py

Show/Hide

Different Ways of Calling a Package

One challenge when your projects grow in complexity is communicating to the user how to use your project. Since the package consists of four different source code files, how does the user know which file to call to run reader?

The python interpreter program has an `-m` option that allows you to specify a module name instead of a file name. For instance, if you have a script called `hello.py`, the following two commands are equivalent:

Shell

```
$ python hello.py  
Hi there!  
  
$ python -m hello  
Hi there!
```

One advantage of the latter is that it allows you to call modules that are built into Python as well. One example is calling [antigravity](#):

Shell

```
$ python -m antigravity  
Created new window in existing browser session.
```

Another advantage of using `-m` is that it works for packages as well as modules. As you saw earlier, you can call the `reader` package with `-m`:

Shell

```
$ python -m reader  
[...]
```

Since `reader` is a package, the name only refers to a directory. How does Python decide which code inside that directory to run? It looks for a file named `__main__.py`. If such a file exists, it is executed. If `__main__.py` does not exist, then an error message is printed:

Shell

```
$ python -m math  
python: No code object available for math
```

In this example, you see that the `math` standard library has not defined a `__main__.py` file.

If you are creating a package that is supposed to be executed, you should include a `__main__.py` file. [Later](#), you'll see how you can also create entry points to your package that will behave like regular programs.

Preparing Your Package for Publication

Now you've got a package you want to publish, or maybe you [copied our package](#). Which steps are necessary before

you can upload the package to PyPI?

Naming Your Package

The first—and possibly the hardest—step is to come up with a good name for your package. All packages on PyPI need to have unique names. With more than 150,000 packages already on PyPI, chances are that your favorite name is already taken.

You might need to brainstorm and do some research to find the perfect name. Use the [PyPI search](#) to check if a name is already taken. The name that you come up with will be visible on PyPI.

To make the reader package easier to find on PyPI, we give it a more descriptive name and call it `realpython-reader`. The same name will be used to install the package using `pip`:

Shell

```
$ pip install realpython-reader
```

Even though we use `realpython-reader` as the PyPI name, the package is still called `reader` when it's imported:

Python

>>>

```
>>> import reader
>>> help(reader)

>>> from reader import feed
>>> feed.get_titles()
['How to Publish an Open-Source Python Package to PyPI', ...]
```

As you see, you can use different names for your package on PyPI and when importing. However, if you use the same name or very similar names, then it will be easier for your users.

Configuring Your Package

In order for your package to be uploaded to PyPI, you need to provide some basic information about it. This information is typically provided in the form of a `setup.py` file. There are [initiatives](#) that try to simplify this collection of information. At the moment though, `setup.py` is the only fully supported way of providing information about your package.

The `setup.py` file should be placed in the top folder of your package. A fairly minimal `setup.py` for `reader` looks like this:

Python

```
import pathlib
from setuptools import setup

# The directory containing this file
HERE = pathlib.Path(__file__).parent

# The text of the README file
README = (HERE / "README.md").read_text()

# This call to setup() does all the work
setup(
    name="realpython-reader",
    version="1.0.0",
    description="Read the latest Real Python tutorials",
    long_description=README,
    long_description_content_type="text/markdown",
    url="https://github.com/realpython/reader",
    author="Real Python",
    author_email="office@realpython.com"
```

```
author_email="officed@realpython.com",
license="MIT",
classifiers=[
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.7",
],
packages=["reader"],
include_package_data=True,
install_requires=["feedparser", "html2text"],
entry_points={
    "console_scripts": [
        "realpython=reader.__main__:main",
    ]
},
)
```

We will only cover some of the options available in `setuptools` here. The [documentation](#) does a good job of going into all the detail.

The parameters that are 100% necessary in the call to `setup()` are the following:

- **name:** the name of your package as it will appear on PyPI
- **version:** the current version of your package
- **packages:** the packages and subpackages containing your source code

We will talk [more about versions later](#). The `packages` parameter takes a list of packages. In our example, there is only one package: `reader`.

You also need to specify any subpackages. In more complicated projects, there might be many packages to list. To simplify this job, `setuptools` includes [`find_packages\(\)`](#), which does a good job of discovering all your subpackages. You could have used `find_packages()` in the `reader` project as follows:

Python

```
from setuptools import find_packages, setup

setup(
    ...
    packages=find_packages(exclude=("tests",)),
    ...
)
```

While only `name`, `version`, and `packages` are required, your package becomes much easier to find on PyPI if you add some more information. Have a look at the [realpython-reader page on PyPI](#) and compare the information with `setup.py` above. All the information comes from `setup.py` and `README.md`.

The last two parameters to `setup()` deserve special mention:

- `install_requires` is used to list any dependencies your package has to third party libraries. The reader depends on `feedparser` and `html2text`, so they should be listed here.
- `entry_points` is used to create scripts that call a function within your package. In our example, we create a new script `realpython` that calls `main()` within the `reader/__main__.py` file.

For another example of a typical setup file, see Kenneth Reitz's [setup.py repository on GitHub](#).

Documenting Your Package

Before releasing your package to the world, you should [add some documentation](#). Depending on your package, the documentation can be as small as a simple README file, or as big as a full web page with tutorials, example galleries, and an API reference.

At a minimum, you should include a README file with your project. A good README should quickly describe your project, as well as tell your users how to install and use your package. Typically, you want to include your README as the `long_description` argument to `setup()`. This will display your README on PyPI.

Traditionally, PyPI has used [reStructuredText](#) for package documentation. However, since March 2018 [Markdown](#) has [also been supported](#).

Outside of PyPI, Markdown is more widely supported than reStructuredText. If you don't need any of the special features of reStructuredText, you'll be better off keeping your README in Markdown. Note that you should use the `setup()` parameter `long_description_content_type` to [tell PyPI which format you are using](#). Valid values are `text/markdown`, `text/x-rst`, and `text/plain`.

For bigger projects, you might want to offer more documentation than can reasonably fit in a single file. In that case, you can use sites like [GitHub](#) or [Read the Docs](#), and link to the documentation using the `url` parameter. In the `setup.py` example above, `url` is used to link to the [reader GitHub repository](#).

Versioning Your Package

Your package needs to have a version, and PyPI will only let you do one upload of a particular version for a package. In other words, if you want to update your package on PyPI, you need to increase the version number first. This is a good thing, as it guarantees reproducibility: two systems with the same version of a package should behave the same.

There are [many different schemes](#) that can be used for your version number. For Python projects, [PEP 440](#) gives some recommendations. However, in order to be flexible, that PEP is complicated. For a simple project, stick with a simple versioning scheme.

[Semantic versioning](#) is a good default scheme to use. The version number is given as three numerical components,

for instance `0.1.2`. The components are called MAJOR, MINOR, and PATCH, and there are simple rules about when to increment each component:

- Increment the MAJOR version when you make incompatible API changes.
- Increment the MINOR version when you add functionality in a backwards-compatible manner.
- Increment the PATCH version when you make backwards-compatible bug fixes. ([Source](#))

You may need to specify the version in different files inside your project. In the reader project, we specified the version both in `setup.py` and in `reader/__init__.py`. To make sure the version numbers are kept consistent, you can use a tool called [Bumpversion](#).

You can install Bumpversion from PyPI:

Shell

```
$ pip install bumpversion
```

To increment the MINOR version of reader, you would do something like this:

Shell

```
$ bumpversion --current-version 1.0.0 minor setup.py reader/__init__.py
```

This would change the version number from `1.0.0` to `1.1.0` in both `setup.py` and `reader/__init__.py`. To simplify the command, you can also give most of the information in a configuration file. See the [Bumpversion documentation](#) for details.

Adding Files to Your Package

Sometimes, you'll have files inside your package that are not source code files. Examples include data files, binaries, documentation, and—as we have in this project—configuration files.

To tell `setup()` to include such files, you use a manifest file. For many projects, you don't need to worry about the manifest, as `setup()` creates one that includes all code files as well as `README` files.

If you need to change the manifest, you create a manifest template which must be named `MANIFEST.in`. This file specifies rules for what to include and exclude:

Text

```
include reader/*.txt
```

This example will include all `.txt` files in the `reader` directory, which in effect is the configuration file. See [the documentation](#) for a list of available rules.

In addition to creating `MANIFEST.in`, you also need to tell `setup()` to [copy these non-code files](#). This is done by setting the `include_package_data` argument to `True`:

Python

```
setup(  
    ...  
    include_package_data=True,  
    ...  
)
```

The `include_package_data` argument controls whether non-code files are copied when your package is installed.

Publishing to PyPI

Your package is finally ready to meet the world outside your computer! In this section, you'll see how to actually upload your package to PyPI.

If you don't already have an account on PyPI, now is the time to create one: [register your account on PyPI](#). While

you're at it, you should also [register an account on TestPyPI](#). TestPyPI is very useful, as you can try all the steps of publishing a package without any consequences if you mess up.

To upload your package to PyPI, you'll use a tool called [Twine](#). You can install Twine using Pip as usual:

Shell

```
$ pip install twine
```

Using Twine is quite simple, and you will soon see how to use it to check and publish your package.

Building Your Package

Packages on PyPI are not distributed as plain source code. Instead, they are wrapped into distribution packages. The most common formats for distribution packages are source archives and [Python wheels](#).

A source archive consists of your source code and any supporting files wrapped into one [tar file](#). Similarly, a wheel is essentially a zip archive containing your code. In contrast to the source archive, the wheel includes any extensions ready to use.

To create a source archive and a wheel for your package, you can run the following command:

Shell

```
$ python setup.py sdist bdist_wheel
```

This will create two files in a newly created dist directory, a source archive and a wheel:

```
reader/
  |
  └── dist/
      ├── realpython_reader-1.0.0-py3-none-any.whl
      └── realpython-reader-1.0.0.tar.gz
```

Note: On Windows, the source archive will be a .zip file by default. You can choose the format of the source archive [with the --format command line option](#).

You might wonder how setup.py knows what to do with the sdist and bdist_wheel arguments. If you [look back](#) to how setup.py was implemented, there is no mention of sdist, bdist_wheel, or any other command line arguments.

All the command line arguments are instead implemented in the upstream [distutils standard library](#). You can list all available arguments by adding the --help-commands option:

Shell

```
$ python setup.py --help-commands
Standard commands:
  build           build everything needed to install
  build_py        "build" pure Python modules (copy to build directory)
  build_ext       build C/C++ and Cython extensions (compile/link to build directory)
< ... many more commands ...>
```

For information about one particular command, you can do something like `python setup.py sdist --help`.

Testing Your Package

First, you should check that the newly built distribution packages contain the files you expect. On Linux and macOS, you should be able to list the contents of the tar source archive as follows:

Shell

```
$ tar tf realpython-reader-1.0.0.tar.gz
realpython-reader-1.0.0/
realpython-reader-1.0.0/setup.cfg
realpython-reader-1.0.0/README.md
realpython-reader-1.0.0/reader/
realpython-reader-1.0.0/reader/feed.py
realpython-reader-1.0.0/reader/__init__.py
realpython-reader-1.0.0/reader/viewer.py
realpython-reader-1.0.0/reader/__main__.py
realpython-reader-1.0.0/reader/config.txt
realpython-reader-1.0.0/PKG-INFO
realpython-reader-1.0.0/setup.py
realpython-reader-1.0.0/MANIFEST.in
realpython-reader-1.0.0/reallypython_reader.egg-info/
realpython-reader-1.0.0/reallypython_reader.egg-info/SOURCES.txt
realpython-reader-1.0.0/reallypython_reader.egg-info/requirements.txt
realpython-reader-1.0.0/reallypython_reader.egg-info/dependency_links.txt
realpython-reader-1.0.0/reallypython_reader.egg-info/PKG-INFO
realpython-reader-1.0.0/reallypython_reader.egg-info/entry_points.txt
realpython-reader-1.0.0/reallypython_reader.egg-info/top_level.txt
```

On Windows, you can use a utility like [7-zip](#) to look inside the corresponding zip file.

You should see all your source code listed, as well as a few new files that have been created containing information you provided in setup.py. In particular, make sure that all subpackages and supporting files are included.

You can also have a look inside the wheel by unzipping it as if it were a zip file. However, if your source archive contains the files you expect, the wheel should be fine as well.

Newer versions of Twine (1.12.0 and above) can also check that your package description will render properly on PyPI. You can run twine check on the files created in dist:

Shell

```
$ twine check dist/*
Checking distribution dist/reallypython_reader-1.0.0-py3-none-any.whl: Passed
Checking distribution dist/reallypython_reader-1.0.0.tar.gz: Passed
```

While it won't catch all problems you might run into, it will for instance let you know if you are using the wrong content type.

Uploading Your Package

Now you're ready to actually upload your package to PyPI. For this, you'll again use the Twine tool, telling it to upload the distribution packages you have built. First, you should upload to [TestPyPI](#) to make sure everything works as expected:

Shell

```
$ twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Twine will ask you for your username and password.

Note: If you've followed the tutorial using the reader package as an example, the previous command will probably fail with a message saying you are not allowed to upload to the realpython-reader project.

You can change the name in setup.py to something unique, for example test-your-username. Then build the project again and upload the newly built files to TestPyPI.

If the upload succeeds, you can quickly head over to [TestPyPI](#), scroll down, and look at your project being proudly displayed among the new releases! Click on your package and make sure everything looks okay.

If you have been following along using the reader package, the tutorial ends here! While you can play with TestPyPI as much as you want, you shouldn't upload dummy packages to PyPI just for testing.

However, if you have your own package to publish, then the moment has finally arrived! With all the preparations taken care of, this final step is short:

Shell

```
$ twine upload dist/*
```

Provide your username and password when requested. That's it!

Head over to [PyPI](#) and look up your package. You can find it either by [searching](#), by looking at the [Your projects page](#), or by going directly to the URL of your project: [pypi.org/project/your-package-name/](#).

Congratulations! Your package is published on PyPI!

pip install Your Package

Take a moment to bask in the blue glow of the PyPI web page and (of course) brag to your friends.

Then open up a terminal again. There is one more great pay off!

With your package uploaded to PyPI, you can install it with pip as well:

Shell

```
$ pip install your-package-name
```

Replace your -package-name with the name you chose for your package. For instance, to install the reader package, you would do the following:

Shell

```
$ pip install realpython-reader
```

Seeing your own code installed by pip is a wonderful feeling!

Other Useful Tools

Before wrapping up, there are a few other tools that are useful to know about when creating and publishing Python packages.

Virtual Environments

In this guide, we haven't talked about virtual environments. Virtual environments are very useful when working with different projects, each with their own differing requirements and dependencies.

See the following guides for more information:

- [Python Virtual Environments: A Primer](#)
- [Pipenv: A Guide to the New Python Packaging Tool](#)
- [Managing Python Dependencies With Pip and Virtual Environments](#)

In particular, it's useful to test your package inside a minimal virtual environment to make sure you're including all necessary dependencies in your `setup.py` file.

Cookiecutter

One great way to get started with your project is to use [Cookiecutter](#). It sets up your project by asking you a few questions based on a template. [Many different templates](#) are available.

First, make sure you have Cookiecutter installed on your system. You can install it from PyPI:

Shell

```
$ pip install cookiecutter
```

As an example, we'll use the [pypackage-minimal](#) template. To use a template, give Cookiecutter a link to the template:

Shell

```
$ cookiecutter https://github.com/krgniz/cookiecutter-pypackage-minimal
author_name [Louis Taylor]: Real Python
author_email [louis@krgniz.eu]: office@realpython.com
package_name [cookiecutter_pypackage_minimal]: realpython-reader
package_version [0.1.0]:
package_description [...]: Read Real Python tutorials
package_url [...]: https://github.com/realpython/reader
readme_pypi_badge [True]:
readme_travis_badge [True]: False
readme_travis_url [...]:
```

After you have answered a series of questions, Cookiecutter sets up your project. In this example, the template created the following files and directories:

```
realpython-reader/
|
|   realpython-reader/
|   |   __init__.py
|
|   tests/
|   |   __init__.py
|   |   test_sample.py
|
|   README.rst
|   setup.py
|   tox.ini
```

[Cookiecutter's documentation](#) is extensive and includes a long list of available cookiecutters, as well as tutorials on how to create your own template.

Flit

The [history of packaging in Python](#) is quite messy. One [common criticism](#) is that using an executable file like `setup.py` for configuration information is not ideal.

[PEP 518](#) defines an alternative: using a file called `pyproject.toml` instead. The [TOML format](#) is a simple configuration file format:

[...] it is human-readable (unlike [JSON](#)), it is flexible enough (unlike [configparser](#)), stems from a standard (also unlike [configparser](#)), and it is not overly complex (unlike [YAML](#)). ([Source](#))

While PEP 518 is already a few years old, the `pyproject.toml` configuration file is not yet fully supported in the standard tools.

However, there are a few new tools that can publish to PyPI based on `pyproject.toml`. One such tool is [Flit](#), a great little project that makes it easy to publish simple Python packages. Flit doesn't support advanced packages like those creating C extensions.

You can `pip install flit`, and then start using it as follows:

Shell

```
$ flit init
Module name [reader]:
Author []: Real Python
Author email []: office@realpython.com
Home page []: https://github.com/realpython/reader
Choose a license (see http://choosealicense.com/ for more info)
1. MIT - simple and permissive
```

```
2. Apache - explicitly grants patent rights
3. GPL - ensures that code based on this is shared with the same terms
4. Skip - choose a license later
Enter 1-4 [1]:
```

```
Written pyproject.toml; edit that file to add optional extra info.
```

The `flit init` command will create a `pyproject.toml` file based on the answers you give to a few questions. You might need to edit this file slightly before using it. For the reader project, the `pyproject.toml` file for Flit ends up looking as follows:

Config File

```
[build-system]
requires = ["flit"]
build-backend = "flit.buildapi"

[tool.flit.metadata]
module = "reader"
dist-name = "realpython-reader"
description-file = "README.md"
author = "Real Python"
author-email = "office@realpython.com"
home-page = "https://github.com/realpython/reader"
classifiers = [
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.7",
]
requires-python = ">=3.7"
requires = ["feedparser", "html2text"]

[tool.flit.scripts]
realpython = "reader.__main__:main"
```

You should recognize most of the items from our original `setup.py`. One thing to note though is that `version` and `description` are missing. This is not a mistake. Flit actually figures these out itself by using `__version__` and the `docstring` defined in the `__init__.py` file. [Flit's documentation](#) explains everything about the `pyproject.toml` file.

Flit can build your package and even publish it to PyPI. To build your package, simply do the following:

Shell

```
$ flit build
```

This creates a source archive and a wheel, exactly like `python setup.py sdist bdist_wheel` did earlier. To upload your package to PyPI, you can use Twine as earlier. However, you can also use Flit directly:

Shell

```
$ flit publish
```

The `publish` command will build your package if necessary, and then upload the files to PyPI, prompting you for your username and password if necessary.

To see Flit in action, have a look at the [2 minute lightning talk](#) from EuroSciPy 2017. The [Flit documentation](#) is a great resource for more information. Brett Cannon's [tutorial on packaging up your Python code for PyPI](#) includes a section about Flit.

[Poetry](#) is another tool that can be used to build and upload your package. It's quite similar to Flit, especially for the things we're looking at here.

Before you use Poetry, you need to install it. It's possible to `pip install poetry` as well. However, the [author recommends](#) that you use a custom installation script to avoid potential dependency conflicts. See [the documentation](#) for installation instructions.

With Poetry installed, you start using it with an `init` command:

Shell

```
$ poetry init

This command will guide you through creating your pyproject.toml config.

Package name [code]: realpython-reader
Version [0.1.0]: 1.0.0
Description []: Read the latest Real Python tutorials
...
```

This will create a `pyproject.toml` file based on your answers to questions about your package. Unfortunately, the actual specifications inside the `pyproject.toml` differ between Flit and Poetry. For Poetry, the `pyproject.toml` file ends up looking like the following:

Config File

```
[tool.poetry]
name = "realpython-reader"
version = "1.0.0"
description = "Read the latest Real Python tutorials"
readme = "README.md"
homepage = "https://github.com/realpython/reader"
authors = ["Real Python <office@realpython.com>"]
license = "MIT"
packages = [{include = "reader"}]
include = ["reader/*.txt"]

[tool.poetry.dependencies]
python = ">=3.7"
feedparser = ">=5.2"
html2text = ">=2018.1"

[tool.poetry.scripts]
realpython = "reader.__main__:main"

[build-system]
requires = ["poetry>=0.12"]
build-backend = "poetry.masonry.api"
```

Again, you should recognize all these items from the earlier discussion of `setup.py`. One thing to note is that Poetry will automatically add classifiers based on the license and the version of Python you specify. Poetry also requires you to be explicit about versions of your dependencies. In fact, dependency management is one of the strong points of Poetry.

Just like Flit, Poetry can build and upload packages to PyPI. The `build` command creates a source archive and a wheel:

Shell

```
$ poetry build
```

This will create the two usual files in the `dist` subdirectory, which you can upload using Twine as earlier. You can also use Poetry to publish to PyPI:

Shell

```
$ poetry publish
```

This will upload your package to PyPI. In addition to building and publishing, Poetry can help you earlier in the process. Similar to Cookiecutter, Poetry can help you start a new project with the new command. It also supports working with virtual environments. See [Poetry's documentation](#) for all the details.

Apart from the slightly different configuration files, Flit and Poetry work very similarly. Poetry is broader in scope as it also aims to help with dependency management, while Flit has been around a little longer. Andrew Pinkham's article [Python's New Package Landscape](#) covers both Flit and Poetry. Poetry was one of the topics at the special [100th episode of the Python Bytes podcast](#).

Conclusion

You now know how to prepare your project and upload it to PyPI, so that it can be installed and used by other people. While there are a few steps you need to go through, seeing your own package on PyPI is a great pay off. Having others find your project useful is even better!

In this tutorial, you've seen the steps necessary to publish your own package:

- Find a good name for your package
- Configure your package using `setup.py`
- Build your package
- Upload your package to PyPI

In addition, you've also seen a few new tools for publishing packages that use the new `pyproject.toml` configuration file to simplify the process.

If you still have questions, feel free to reach out in the comments section below. Also, the [Python Packaging Authority](#) has a lot of information with more detail than we covered here.

 This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [How to Publish Your Own Python Package to PyPI](#)

About Geir Arne Hjelle

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

[» More about Geir Arne](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Adriana](#)

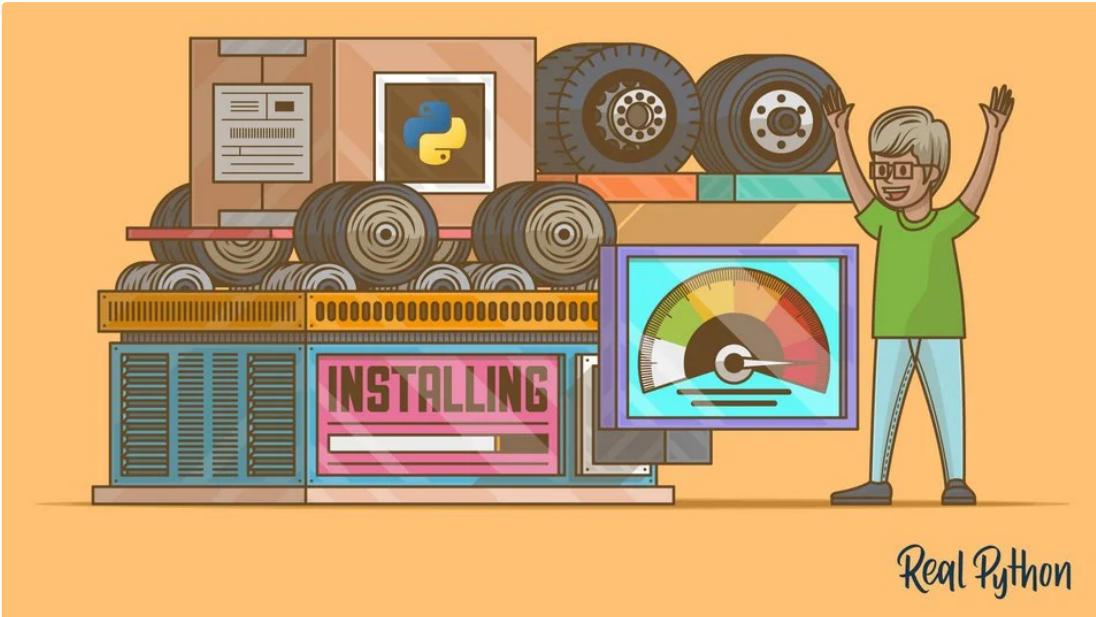
[David](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [tools](#)

Recommended Video Course: [How to Publish Your Own Python Package to PyPI](#)



Real Python

What Are Python Wheels and Why Should You Care?

by Brad Solomon ⌂ Aug 05, 2020 6 Comments 🗑 intermediate python

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Setup](#)
- [Python Packaging Made Better: An Intro to Python Wheels](#)
 - [Wheels Make Things Go Fast](#)
 - [What Is a Python Wheel?](#)
 - [Advantages of Python Wheels](#)
 - [Telling pip What to Download](#)
 - [The manylinux Wheel Tag](#)
 - [Security Considerations With Platform Wheels](#)
- [Calling All Developers: Build Your Wheels](#)
 - [Different Types of Wheels](#)
 - [Building a Pure-Python Wheel](#)
 - [Specifying a Universal Wheel](#)
 - [Building a Platform Wheel \(macOS and Windows\)](#)
 - [Linux: Building manylinux Wheels](#)
 - [Bundling Shared Libraries](#)
 - [Building Wheels in Continuous Integration](#)
 - [Making Sure Your Wheels Spin Right](#)
 - [Uploading Python Wheels to PyPI](#)
- [Conclusion](#)
- [Resources](#)



Python .whl files, or [wheels](#), are a little-discussed part of Python, but they've been a boon to the installation process for Python packages. If you've installed a Python package using [pip](#), then chances are that a wheel has made the installation faster and more efficient.

Wheels are a component of the Python ecosystem that helps to make package installs *just work*. They allow for faster installations and more stability in the package distribution process. In this tutorial, you'll dive into what

wheels are, what good they serve, and how they've gained traction and made Python even more of a joy to work with.

In this tutorial, you'll learn:

- What wheels are and how they compare to **source distributions**
- How you can use wheels to control the **package installation** process
- How to **create and distribute** wheels for your own Python packages

You'll see examples using popular open source Python packages from both the user's and the developer's perspective.

Free Bonus: [Click here to get a Python Cheat Sheet](#) and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Setup

To follow along, activate a [virtual environment](#) and make sure you have the latest versions of pip, wheel, and setuptools installed:

Shell

```
$ python -m venv env && source ./env/bin/activate
$ python -m pip install -U pip wheel setuptools
Successfully installed pip 20.1 setuptools-46.1.3 wheel-0.34.2
```

That's all you need to experiment with installing and building wheels!

Python Packaging Made Better: An Intro to Python Wheels

Before you learn how to package a project into a wheel, it helps to know what using one looks like from the user's side. It may sound backward, but a good way to learn how wheels work is to start by installing something that *isn't* a wheel.

You can start this experiment by installing a Python package into your environment just as you might normally do. In this case, install [uWSGI](#) version 2.0.x:

Shell

```
1 $ python -m pip install 'uwsgi==2.0.*'
2 Collecting uwsgi==2.0.*
3   Downloading uwsgi-2.0.18.tar.gz (801 kB)
4     ██████████| 801 kB 1.1 MB/s
5 Building wheels for collected packages: uwsgi
6   Building wheel for uwsgi (setup.py) ... done
7     Created wheel for uwsgi ... uWSGI-2.0.18-cp38-cp38-macosx_10_15_x86_64.whl
8     Stored in directory: /private/var/folders/jc/8_hqsz0x1tdbp05 ...
9 Successfull built uwsgi
10 Installing collected packages: uwsgi
11 Successfully installed uwsgi-2.0.18
```

To fully install uWSGI, pip progresses through several distinct steps:

1. On **line 3**, it downloads a TAR file (tarball) named uwsgi-2.0.18.tar.gz that's been compressed with [gzip](#).
2. On **line 6**, it takes the tarball and builds a .whl file through a call to setup.py.
3. On **line 7**, it labels the wheel uWSGI-2.0.18-cp38-cp38-macosx_10_15_x86_64.whl.
4. On **line 10**, it installs the actual package after having built the wheel.

The tar.gz tarball that pip retrieves is a **source distribution**, or sdist, rather than a wheel. In some ways, a sdist is the opposite of a wheel.

Note: If you see an error with the uWSGI installation, you may need to [install the Python development headers](#).

A [source distribution](#) contains source code. That includes not only Python code but also the source code of any extension modules (usually in C or [C++](#)) bundled with the package. With source distributions, extension modules are compiled on the user's side rather than the developer's.

Source distributions also contain a bundle of metadata sitting in a directory called `<package-name>.egg-info`. This metadata helps with building and installing the package, but user's don't really need to do anything with it.

From the developer's perspective, a source distribution is what gets created when you run the following command:

Shell

```
$ python setup.py sdist
```

Now try installing a different package, [chardet](#):

Shell

```
1 $ python -m pip install 'chardet==3.*'
2 Collecting chardet
3   Downloading chardet-3.0.4-py2.py3-none-any.whl (133 kB)
4     ██████████ | 133 kB 1.5 MB/s
5 Installing collected packages: chardet
6 Successfully installed chardet-3.0.4
```

You can see a noticeably different output than the uWSGI install.

Installing chardet downloads a .whl file directly from PyPI. The wheel name chardet-3.0.4-py2.py3-none-any.whl follows a specific naming convention that you'll see later. What's more important from the user's perspective is that there's no build stage when pip finds a compatible wheel on PyPI.

From the developer's side, a wheel is the result of running the following command:

Shell

```
$ python setup.py bdist_wheel
```

Why does uWSGI hand you a source distribution while chardet provides a wheel? You can see the reason for this by taking a look at each project's page on PyPI and navigating to the *Download files* area. This section will show you what pip actually sees on the PyPI index server:

- **uWSGI** [provides only a source distribution](#) (`uwsgi-2.0.18.tar.gz`) for reasons related to the complexity of the project.
- **chardet** [provides both a wheel and a source distribution](#), but pip will prefer the wheel if it's compatible with your system. You'll see how that compatibility is determined later on.

Another example of the compatibility check used for wheel installation is [psycopg2](#), which provides a wide set of wheels for Windows but doesn't provide any for Linux or macOS clients. This means that `pip install psycopg2` could fetch a wheel or a source distribution depending on your specific setup.

To avoid these types of compatibility issues, some packages offer multiple wheels, with each wheel geared toward a specific Python implementation and underlying operating system.

So far, you've seen some of the visible distinctions between a wheel and sdist, but what matters more is the impact those differences have on the installation process.

Wheels Make Things Go Fast

Above, you saw a comparison of an installation that fetches a prebuilt wheel and one that downloads a sdist. Wheels make the end-to-end installation of Python packages faster for two reasons:

1. All else being equal, wheels are typically **smaller in size** than source distributions, meaning they can move faster across a network.
2. Installing from wheels directly avoids the intermediate step of **building** packages off of the source distribution.

It's almost guaranteed that the chardet install occurred in a fraction of the time required for uWSGI. However, that's

arguably an unfair apples-to-oranges comparison since chardet is a significantly smaller and less complex package. With a different command, you can create a more direct comparison that will demonstrate just how much of a difference wheels make.

You can make pip ignore its inclination towards wheels by passing the `--no-binary` option:

Shell

```
$ time python -m pip install \
    --no-cache-dir \
    --force-reinstall \
    --no-binary=:all: \
    cryptography
```

This command times the installation of the [cryptography](#) package, telling pip to use a source distribution even if a suitable wheel is available. Including `:all:` makes the rule apply to `cryptography` and all of its dependencies.

On my machine, this takes around *thirty-two seconds* from start to finish. Not only does the install take a long time, but building `cryptography` also requires that you have the OpenSSL development headers present and available to Python.

Note: With `--no-binary`, you may very well see an error about missing header files required for the `cryptography` install, which is part of what can make using source distributions frustrating. If so, the [installation section](#) of the `cryptography` docs advises on which libraries and header files you'll need for a particular operating system.

Now you can reinstall `cryptography`, but this time make sure that pip uses wheels from PyPI. Because pip will prefer a wheel, this is similar to just calling `pip install` with no arguments at all. But in this case, you can make the intent explicit by requiring a wheel with `--only-binary`:

Shell

```
$ time python -m pip install \
    --no-cache-dir \
    --force-reinstall \
    --only-binary=cryptography \
    cryptography
```

This option takes just over four seconds, or *one-eighth* the time that it took when using only source distributions for `cryptography` and its dependencies.

What Is a Python Wheel?

A Python `.whl` file is essentially a ZIP (`.zip`) archive with a specially crafted filename that tells installers what Python versions and platforms the wheel will support.

A wheel is a type of [built distribution](#). In this case, *built* means that the wheel comes in a ready-to-install format and allows you to skip the build stage required with source distributions.

Note: It's worth mentioning that despite the use of the term *built*, a wheel doesn't contain `.pyc` files, or compiled Python bytecode.

A wheel filename is broken down into parts separated by hyphens:

Text

```
{dist}-{version}({-{build}})?-{python}-{abi}-{platform}.whl
```

Each section in `{brackets}` is a **tag**, or a component of the wheel name that carries some meaning about what the wheel contains and where the wheel will or will not work.

Here's an illustrative example using a `cryptography` wheel:

Text

```
cryptography-2.9.2-cp35-abi3-macosx_10_9_x86_64.whl
```

cryptography distributes multiple wheels. Each wheel is a **platform wheel**, meaning it supports only specific combinations of Python versions, Python ABIs, operating systems, and machine architectures. You can break down the naming convention into parts:

- **cryptography** is the package name.
- **2.9.2** is the package version of cryptography. A version is a [PEP 440](#)-compliant string such as `2.9.2`, `3.4`, or `3.9.0.a3`.
- **cp35** is the [Python tag](#) and denotes the Python implementation and version that the wheel demands. The cp stands for [CPython](#), the reference implementation of Python, while the 35 denotes Python [3.5](#). This wheel wouldn't be compatible with [Jython](#), for instance.
- **abi3** is the ABI tag. ABI stands for [application binary interface](#). You don't really need to worry about what it entails, but abi3 is a separate version for the binary compatibility of the Python C API.
- **macosx_10_9_x86_64** is the platform tag, which happens to be quite a mouthful. In this case it can be broken down further into sub-parts:
 - **macosx** is the [macOS](#) operating system.
 - **10_9** is the macOS developer tools SDK version used to compile the Python that in turn built this wheel.
 - **x86_64** is a reference to x86-64 instruction set architecture.

The final component isn't technically a tag but rather the standard `.whl` file extension. Combined, the above components indicate the target machine that this cryptography wheel is designed for.

Now let's turn to a different example. Here's what you saw in the above case for chardet:

Text

```
chardet-3.0.4-py2.py3-none-any.whl
```

You can break this down into its tags:

- **chardet** is the package name.
- **3.0.4** is the package version of chardet.
- **py2.py3** is the Python tag, meaning the wheel supports Python 2 and 3 with any Python implementation.
- **none** is the ABI tag, meaning the ABI isn't a factor.
- **any** is the platform. This wheel will work on virtually any platform.

The `py2.py3-none-any.whl` segment of the wheel name is common. This is a **universal wheel** that will install with Python 2 or 3 on any platform with any [ABI](#). If the wheel ends in `none-any.whl`, then it's very likely a pure-Python package that doesn't care about a specific Python ABI or CPU architecture.

Another example is the `jinja2` templating engine. If you navigate to the [downloads page](#) for the Jinja 3.x alpha release, then you'll see the following wheel:

Text

```
Jinja2-3.0.0a1-py3-none-any.whl
```

Notice the lack of `py2` here. This is a pure-Python project that will work on any Python 3.x version, but it's not a universal wheel because it doesn't support Python 2. Instead, it's called a **pure-Python wheel**.

Note: In 2020, a number of projects are also [dropping support for Python 2](#), which reached end-of-life (EOL) on January 1, 2020. Jinja version 3.x [dropped Python 2 support](#) in February 2020.

Here are a few more examples of .whl names distributed for some popular open source packages:

Wheel	What It Is
PyYAML-5.3.1-cp38-cp38-win_amd64.whl	PyYAML for CPython 3.8 on Windows with AMD64 (x86-64) architecture
numpy-1.18.4-cp38-cp38-win32.whl	NumPy for CPython 3.8 on Windows 32-bit
scipy-1.4.1-cp36-cp36m-macosx_10_6_intel.whl	SciPy for CPython 3.6 on macOS 10.6 SDK with fat binary (multiple instruction sets)

Now that you have a thorough understanding of what wheels are, it's time to talk about what good they serve.

Advantages of Python Wheels

Here's a testament to wheels from the [Python Packaging Authority](#) (PyPA):

Not all developers have the right tools or experiences to build these components written in these compiled languages, so Python created the wheel, a package format designed to ship libraries with compiled artifacts. In fact, Python's package installer, pip, always prefers wheels because installation is always faster, so even pure-Python packages work better with wheels. ([Source](#))

A fuller description is that wheels [benefit both users and maintainers](#) of Python packages alike in a handful of ways:

- **Wheels install faster** than source distributions for both pure-Python packages and [extension modules](#).
- **Wheels are smaller** than source distributions. For example, the [six](#) wheel is about [one-third the size](#) of the corresponding source distribution. This differential becomes even more important when you consider that a `pip install` for a single package may actually kick off downloading a chain of dependencies.
- **Wheels cut `setup.py` execution out of the equation.** Installing from a source distribution runs *whatever* is contained in that project's `setup.py`. As pointed out by [PEP 427](#), this amounts to arbitrary code execution. Wheels avoid this altogether.
- **There's no need for a compiler** to install wheels that contain compiled extension modules. The extension module comes included with the wheel targeting a specific platform and Python version.
- **pip automatically generates .pyc files** in the wheel that match the right Python interpreter.
- **Wheels provide consistency** by cutting many of the variables involved in installing a package out of the equation.

You can use a project's *Download files* tab on PyPI to view the different distributions that are available. For example, [pandas](#) distributes a wide array of wheels.

Telling pip What to Download

It's possible to exert fine-grained control over pip and tell it which format to prefer or avoid. You can use the `--only-binary` and `--no-binary` options to do this. You saw these used in an earlier section on installing the `cryptography` package, but it's worth taking a closer look at what they do:

Shell

```
$ pushd "$(mktemp -d)"
$ python -m pip download --only-binary :all: --dest . --no-cache six
Collecting six
  Downloading six-1.14.0-py2.py3-none-any.whl (10 kB)
    Saved ./six-1.14.0-py2.py3-none-any.whl
Successfully downloaded six
```

In this example, you change to a temporary directory to store the download with `pushd "$(mktemp -d)"`. You use `pip download` rather than `pip install` so that you can inspect the resulting wheel, but you can replace `download` with `install` while keeping the same set of options.

You download the [six](#) module with several flags:

- `--only-binary :all`: tells pip to constrain itself to using wheels and ignore source distributions. Without this option, pip will only *prefer* wheels but will fall back to source distributions in some scenarios.
- `--dest .` tells pip to download six to the current directory.
- `--no-cache` tells pip not to look in its local download cache. You use this option just to illustrate a live download from PyPI since it's likely you do have a six cache somewhere.

I mentioned earlier that a wheel file is essentially a `.zip` archive. You can take this statement literally and treat wheels as such. For instance, if you want to view a wheel's contents, you can use `unzip`:

Shell

```
$ unzip -l six*.whl
Archive:  six-1.14.0-py2.py3-none-any.whl
      Length      Date  Time    Name
----- -----
  34074  01-15-2020 18:10  six.py
   1066  01-15-2020 18:10  six-1.14.0.dist-info/LICENSE
   1795  01-15-2020 18:10  six-1.14.0.dist-info/METADATA
    110  01-15-2020 18:10  six-1.14.0.dist-info/WHEEL
      4  01-15-2020 18:10  six-1.14.0.dist-info/top_level.txt
   435  01-15-2020 18:10  six-1.14.0.dist-info/RECORD
----- -----
  37484                           6 files
```

six is a special case: it's actually a single Python module rather than a complete package. Wheel files can also be significantly more complex, as you'll see later on.

In contrast to `--only-binary`, you can use `--no-binary` to do the opposite:

Shell

```
$ python -m pip download --no-binary :all: --dest . --no-cache six
Collecting six
  Downloading six-1.14.0.tar.gz (33 kB)
    Saved ./six-1.14.0.tar.gz
Successfully downloaded six
$ popd
```

The only change in this example is the switch to `--no-binary :all:`. This tells pip to ignore wheels even if they're available and instead download a source distribution.

When might `--no-binary` be useful? Here are a few cases:

- **The corresponding wheel is broken.** This is an irony of wheels. They're designed to make things break less often, but in some cases a wheel can be misconfigured. In this case, downloading and building the source distribution for yourself may be a working alternative.
- **You want to apply a small change or patch file** to the project and then install it. This is an alternative to cloning the project from its [version control system](#) URL.

You can also use the flags described above with `pip install`. Additionally, instead of `:all:`, which will apply the `--only-binary` rule not just to the package you're installing but to all of its dependencies, you can pass `--only-binary` and `--no-binary` a list of specific packages to apply that rule to.

Here are a few examples for installing the URL library [var1](#). It contains Cython code and depends on [multidict](#).

which contains pure C code. There are several options to strictly use or strictly ignore wheels for `yarl` and its dependencies:

Shell

```
$ # Install `yarl` and use only wheels for yarl and all dependencies
$ python -m pip install --only-binary :all: yarl

$ # Install `yarl` and use wheels only for the `multidict` dependency
$ python -m pip install --only-binary multidict yarl

$ # Install `yarl` and don't use wheels for yarl or any dependencies
$ python -m pip install --no-binary :all: yarl

$ # Install `yarl` and don't use wheels for the `multidict` dependency
$ python -m pip install --no-binary multidict yarl
```

In this section, you got a glimpse of how to fine-tune the distribution types that `pip install` will use. While a regular `pip install` should work with no options, it's helpful to know these options for special cases.

The `manylinux` Wheel Tag

Linux comes in many variants and flavors, such as Debian, CentOS, Fedora, and Pacman. Each of these may use slight variations in shared libraries, such as `libncurses`, and core C libraries, such as `glibc`.

If you're writing a C/C++ extension, then this could create a problem. A source file written in C and compiled on Ubuntu Linux isn't guaranteed to be executable on a CentOS machine or an Arch Linux distribution. Do you need to build a separate wheel for each and every Linux variant?

Luckily, the answer is no, thanks to a specially designed set of tags called the `manylinux` platform tag family. There are currently three variations:

1. `manylinux1` is the original format specified in [PEP 513](#).
2. `manylinux2010` is an update specified in [PEP 571](#) that upgrades to CentOS 6 as the underlying OS on which the Docker images are based. The rationale is that CentOS 5.11, which is where the list of allowed libraries in `manylinux1` comes from, reached EOL in March 2017 and stopped receiving security patches and bug fixes.
3. `manylinux2014` is an update specified in [PEP 599](#) that upgrades to CentOS 7 since CentOS 6 is scheduled to reach EOL in November 2020.

You can find an example of `manylinux` distributions within the `pandas` project. Here are two (out of many) from the list of available [pandas downloads from PyPI](#):

Text

```
pandas-1.0.3-cp37-cp37m-manylinux1_x86_64.whl
pandas-1.0.3-cp37-cp37m-manylinux1_i686.whl
```

In this case, `pandas` has built `manylinux1` wheels for CPython 3.7 supporting both x86-64 and [i686](#) architectures.

At its core, `manylinux` is a [Docker image](#) built off a certain version of the CentOS operating system. It comes bundled with a compiler suite, multiple versions of Python and `pip`, and an allowed set of shared libraries.

Note: The term **allowed** indicates a low-level library that is [assumed to be present by default](#) on almost all Linux systems. The idea is that the dependency should exist on the base operating system without the need for an additional install.

As of mid-2020, `manylinux1` is still the predominant `manylinux` tag. One reason for this might just be habit. Another might be that support on the client (user) side for `manylinux2010` and above is [limited to more recent versions](#) of `pip`:

Tag	Requirement
<code>manylinux1</code>	<code>pip</code> 8.1.0 or later

Tag	Requirement
manylinux2010	pip 19.0 or later
manylinux2014	pip 19.3 or later

In other words, if you’re a package developer building manylinux2010 wheels, then someone using your package will need pip 19.0 (released in January 2019) or later to let pip find and install manylinux2010 wheels from PyPI.

Luckily, virtual environments have become more common, meaning that developers can update a virtual environment’s pip without touching the system pip. This isn’t always the case, however, and some Linux distributions still ship with outdated versions of pip.

All that is to say, if you’re installing Python packages on a Linux host, then consider yourself fortunate if the package maintainer has gone out of their way to create manylinux wheels. This will almost guarantee a hassle-free installation of the package regardless of your specific Linux variant or version.

Caution: Be advised that [PyPI wheels don’t work on Alpine Linux \(or BusyBox\)](#). This is because Alpine uses [musl](#) in place of the standard [glibc](#). The musl libc library bills itself as “a new libc striving to be fast, simple, lightweight, free, and correct.” Unfortunately, when it comes to wheels, glibc it is not.

Security Considerations With Platform Wheels

One feature of wheels worth considering from a user security standpoint is that wheels are [potentially subject to version rot](#) because they bundle a binary dependency rather than allowing that dependency to be updated by your system package manager.

For example, if a wheel incorporates the [libfortran](#) shared library, then distributions of that wheel will use the libfortran version that they were bundled with even if you upgrade your own machine’s version of libfortran with a package manager such as apt, yum, or brew.

If you’re developing in an environment with heightened security precautions, this feature of some platform wheels is something to be mindful of.

Calling All Developers: Build Your Wheels

The title of this tutorial asks, “Why Should You Care?” As a developer, if you plan to distribute a Python package to the community, then you should care immensely about distributing wheels for your project because they make the installation process cleaner and less complex for end users.

The more target platforms that you can support with compatible wheels, the fewer GitHub issues you’ll see titled something like “Installation broken on Platform XYZ.” Distributing wheels for your Python package makes it objectively less likely that users of the package will encounter issues during installation.

The first thing you need to do to build a wheel locally is to install wheel. It doesn’t hurt to make sure that setuptools is up to date, too:

Shell

```
$ python -m pip install -U wheel setuptools
```

The next few sections will walk you through building wheels for a variety of different scenarios.

Different Types of Wheels

As touched on throughout this tutorial, there are several different [variations of wheels](#), and the wheel’s type is reflected in its filename:

- A **universal wheel** contains py2.py3-none-any.whl. It supports both Python 2 and Python 3 on any OS and platform. The majority of wheels listed on the [Python Wheels](#) website are universal wheels.
- A **pure-Python wheel** contains either py3-none-any.whl or py2.none-any.whl. It supports either Python 3 or Python 2, but not both. It’s otherwise the same as a universal wheel, but it’ll be labeled with either py3 or py2.

i_ython_2, but not both. It's otherwise the same as a universal wheel, but it'll be labeled with either py2 or py3 rather than the py2.py3 label.

- A **platform wheel** supports a specific Python version and platform. It contains segments indicating a specific Python version, ABI, operating system, or architecture.

The differences between wheel types are determined by which version(s) of Python they support and whether they target a specific platform. Here's a condensed summary of the differences between wheel variations:

Wheel Type	Supports Python 2 and 3	Supports Every ABI, OS, and Platform
Universal	✓	✓
Pure-Python		✓
Platform		

As you'll see next, you can build universal wheels and pure-Python wheels with relatively little setup, but platform wheels may require a few additional steps.

Building a Pure-Python Wheel

You can build a pure-Python wheel or a universal wheel for any [project using setuptools](#) with just a single command:

Shell

```
$ python setup.py sdist bdist_wheel
```

This will create both a source distribution (`sdist`) and a wheel (`bdist_wheel`). By default, both will be placed in `dist/` under the current directory. To see for yourself, you can build a wheel for [HTTPie](#), a command-line HTTP client written in Python, alongside a `sdist`.

Here's the result of building both types of distributions for the `HTTPie` package:

Shell

```
$ git clone -q git@github.com:jakubroztocil/httpie.git
$ cd httpie
$ python setup.py -q sdist bdist_wheel
$ ls -1 dist/
httpie-2.2.0.dev0-py3-none-any.whl
httpie-2.2.0.dev0.tar.gz
```

That's all it takes. You clone the project, move into its root directory, and then call `python setup.py sdist bdist_wheel`. You can see that `dist/` contains both a wheel and a source distribution.

The resulting distributions get put in `dist/` by default, but you can change that with the `-d/--dist-dir` option. You could put them in a temporary directory instead for build isolation:

Shell

```
$ tempdir="$(mktemp -d)" # Create a temporary directory
$ file "$tempdir"
/var/folders/jc/8_kd8uusys7ak09_1pmn30rw0000gk/T/tmp.GIXy7XKV: directory

$ python setup.py sdist -d "$tempdir"
$ python setup.py bdist_wheel --dist-dir "$tempdir"
$ ls -1 "$tempdir"
httpie-2.2.0.dev0-py3-none-any.whl
httpie-2.2.0.dev0.tar.gz
```

You can combine the `sdist` and `bdist_wheel` steps into one because `setup.py` can take multiple subcommands:

Shell

```
$ python setup.py sdist -d "$tempdir" bdist_wheel -d "$tempdir"
```

As shown here, you'll need to pass options such as `-d` to each subcommand.

Specifying a Universal Wheel

A universal wheel is a wheel for a pure-Python project that supports both Python 2 and 3. There are multiple ways to tell `setuptools` and `distutils` that a wheel should be universal.

Option 1 is to specify the option in your project's `setup.cfg` file:

Config File

```
[bdist_wheel]
universal = 1
```

Option 2 is to pass the aptly named `--universal` flag at the command line:

Shell

```
$ python setup.py bdist_wheel --universal
```

Option 3 is to tell `setup()` itself about the flag using its `options` parameter:

Python

```
# setup.py
from setuptools import setup

setup(
    # ...
    options={"bdist_wheel": {"universal": True}}
    # ...
)
```

While any of these three options should work, the first two are used most frequently. You can see an example of this in the [chardet setup configuration](#). After that, you can use the `bdist_wheel` command as shown previously:

Shell

```
$ python setup.py sdist bdist_wheel
```

The resulting wheel will be equivalent no matter which option you choose. The choice largely comes down to developer preference and which workflow is best for you.

Building a Platform Wheel (macOS and Windows)

[**Binary distributions**](#) are a subset of [**built distributions**](#) that contain compiled extensions. [**Extensions**](#) are non-Python dependencies or components of your Python package.

Usually, that means your package contains an extension module or depends on a library written in a statically typed language such as C, C++, Fortran, or even [Rust](#) or Go. [**Platform wheels**](#) exist to target individual platforms primarily because they contain or depend on extension modules.

With all that said, it's high time to build a platform wheel!

Depending on your existing development environment, you may need to go through an additional prerequisite step or two to build platform wheels. The steps below will help you to get set up for building C and C++ extension modules, which are by far the most common types.

On macOS, you'll need the command-line developer tools available through [xcode](#):

Shell

```
$ xcode-select --install
```

On Windows, you'll need to install [Microsoft Visual C++](#):

1. Open the [Visual Studio downloads page](#) in your browser.
2. Select *Tools for Visual Studio* → *Build Tools for Visual Studio* → *Download*.
3. Run the resulting .exe installer.
4. In the installer, select *C++ Build Tools* → *Install*.
5. Restart your machine.

On Linux, you need a compiler such as [gcc](#) or g++/c++.

With that squared away, you're ready to build a platform wheel for UltraJSON (`ujson`), a [JSON](#) encoder and decoder written in pure C with Python 3 [bindings](#). Using `ujson` is a good toy example because it covers a few bases:

1. It contains an extension module, [ujson](#).
2. It depends on the Python development headers to compile (#include <Python.h>) but is not otherwise overly complicated. `ujson` is designed to do one thing and do it well, which is to read and write JSON!

You can clone the project from GitHub, navigate into its directory, and build it:

Shell

```
$ git clone -q --branch 2.0.3 git@github.com:ultrajson/ultrajson.git  
$ cd ultrajson  
$ python setup.py bdist_wheel
```

You should see a whole lot of output. Here's a trimmed version on macOS, where the [Clang](#) compiler driver is used:

Text

```
clang -Wno-unused-result -Wsign-compare -Wunreachable-code -DNDEBUG -g ...  
...  
creating 'dist/ujson-2.0.3-cp38-cp38-macosx_10_15_x86_64.whl'  
adding 'ujson.cpython-38-darwin.so'
```

The lines starting with `clang` show the actual call to the compiler complete with a trove of compilation flags. You might also see tools such as `MSVC` (Windows) or `gcc` (Linux) depending on the operating system.

If you run into a `fatal error` after executing the above code, don't worry. You can expand the box below to learn how to deal with this problem.

If you inspect UltraJSON's [setup.py](#), then you'll see that it customizes some compiler flags such as `-D_GNU_SOURCE`. The intricacies of controlling the compilation process through `setup.py` are beyond the scope of this tutorial, but you should know that it's possible to have [fine-grained control over how the compiling and linking occurs](#).

If you look in `dist`, then you should see the created wheel:

Shell

```
$ ls dist/  
ujson-2.0.3-cp38-cp38-macosx_10_15_x86_64.whl
```

Note that the name may vary based on your platform. For example, you'd see `win_amd64.whl` on 64-bit Windows.

You can peek into the wheel file and see that it contains the compiled extension:

Shell

```
$ unzip -l dist/ujson-*.whl  
...  
Length      Date      Time      Name  
-----  -----  -----  
105812  05-10-2020 19:47  ujson.cpython-38-darwin.so
```

This example shows an output for macOS, `ujson.cpython-38-darwin.so`, which is a shared object (.so) file, also called a dynamic library.

Linux: Building manylinux Wheels

As a package developer, you'll rarely want to build wheels for a single Linux variant. Linux wheels demand a specialized set of conventions and tools so that they can work across different Linux environments.

Unlike wheels for macOS and Windows, wheels built on one Linux variant have no guarantee of working on another Linux variant, even one with the same machine architecture. In fact, if you build a wheel on an out-of-the-box Linux container, then PyPI won't even accept that wheel if you try to upload it!

If you want your package to be available across a range of Linux clients, then you want a `manylinux` wheel. A `manylinux` wheel is a particular type of a platform wheel that is accepted by most Linux variants. It must be built in a specific environment, and it requires a tool called `auditwheel` that renames the wheel file to indicate that it's a `manylinux` wheel.

Note: Even if you're approaching this tutorial from the developer rather than the user perspective, make sure that you've read the section on [the `manylinux` wheel tag](#) before continuing with this section.

Building a `manylinux` wheel allows you to target a wider range of user platforms. [PEP 513](#) specifies a particular (and archaic) version of CentOS with an array of Python versions available. The choice between CentOS and Ubuntu or any other distribution doesn't carry any special distinction. The point is for the build environment to consist of a stock Linux operating system with a limited set of external shared libraries that are common to different Linux variants.

Thankfully, you don't have to do this yourself. PyPA [provides a set of Docker images](#) that give you this environment with a few mouse clicks:

- **Option 1** is to run `docker` from your development machine and mount your project using a Docker volume so that it's accessible in the container filesystem.
- **Option 2** is to use a [CI/CD](#) solution such as CircleCI, GitHub Actions, Azure DevOps, or Travis-CI, which will pull your project and run the build on an action such as a push or tag.

The Docker images are provided for the different `manylinux` flavors:

manylinux Tag	Architecture	Docker Image
manylinux1	x86-64	quay.io/pypa/manylinux1_x86_64
manylinux1	i686	quay.io/pypa/manylinux1_i686
manylinux2010	x86-64	quay.io/pypa/manylinux2010_x86_64
manylinux2010	i686	quay.io/pypa/manylinux2010_i686
manylinux2014	x86-64	quay.io/pypa/manylinux2014_x86_64
manylinux2014	i686	quay.io/pypa/manylinux2014_i686
manylinux2014	aarch64	quay.io/pypa/manylinux2014_aarch64
manylinux2014	ppc64le	quay.io/pypa/manylinux2014_ppc64le
manylinux2014	s390x	quay.io/pypa/manylinux2014_s390x

To get started, PyPA also provides an example repository, [python-manylinux-demo](#), which is a demo project for building manylinux wheels in conjunction with [Travis-CI](#).

While it's common to build wheels as a part of a remote-hosted CI solution, you can also build manylinux wheels locally. To do so, you'll need [Docker](#) installed. Docker Desktop is available for macOS, Windows, and Linux.

First, clone the demo project:

Shell

```
$ git clone -q git@github.com:pypa/python-manylinux-demo.git  
$ cd python-manylinux-demo
```

Next, define a few shell variables for the manylinux1 Docker image and platform, respectively:

Shell

```
$ DOCKER_IMAGE='quay.io/pypa/manylinux1_x86_64'  
$ PLAT='manylinux1_x86_64'
```

The DOCKER_IMAGE variable is the image maintained by PyPA for building manylinux wheels, hosted at [Quay.io](#). The platform (PLAT) is a necessary piece of information to feed to auditwheel, letting it know what platform tag to apply.

Now you can pull the Docker image and run the wheel-builder script within the container:

Shell

```
$ docker pull "$DOCKER_IMAGE"  
$ docker container run -t --rm \  
-e PLAT=$PLAT \  
-v "$(pwd)":/io \  
"$DOCKER_IMAGE" /io/travis/build-wheels.sh
```

This tells Docker to run the build-wheels.sh shell script inside the manylinux1_x86_64 Docker container, passing PLAT as an environment variable available in the container. Since you used -v (or --volume) to [bind-mount a volume](#), the wheels produced in the container will now be accessible on your host machine in the wheelhouse directory:

Shell

```
$ ls -1 wheelhouse  
python_manylinux_demo-1.0-cp27-cp27m-manylinux1_x86_64.whl  
python_manylinux_demo-1.0-cp27-cp27mu-manylinux1_x86_64.whl  
python_manylinux_demo-1.0-cp35-cp35m-manylinux1_x86_64.whl  
python_manylinux_demo-1.0-cp36-cp36m-manylinux1_x86_64.whl  
python_manylinux_demo-1.0-cp37-cp37m-manylinux1_x86_64.whl  
python_manylinux_demo-1.0-cp38-cp38-manylinux1_x86_64.whl
```

In a few short commands, you have a set of manylinux1 wheels for CPython 2.7 through 3.8. A common practice is also to iterate over different architectures. For instance, you could repeat this process for the quay.io/pypa/manylinux1_i686 Docker image. This would build manylinux1 wheels targeting 32-bit (i686) architecture.

If you'd like to dive deeper into building wheels, then a good next step is to learn from the best. Start at the [Python Wheels](#) page, pick a project, navigate to its source code (on a place like GitHub, GitLab, or Bitbucket), and see for yourself how it builds wheels.

Many of the projects on the Python Wheels page are pure-Python projects and distribute universal wheels. If you're looking for more complex cases, then keep an eye out for packages that use extension modules. Here are two examples to whet your appetite:

1. [lxml](#) uses a separate build script that's invoked from within the manylinux1 Docker container.
2. [ultrajson](#) does the same and uses GitHub Actions to call the build script.

Both of these are reputable projects that offer a great examples to learn from if you're interested in building manylinux wheels.

Bundling Shared Libraries

One other challenge is building wheels for packages that depend on external shared libraries. The `manylinux` images contain a prescreened set of libraries such as `libpthread.so.0` and `libc.so.6`. But what if you rely on something outside that list, such as [ATLAS](#) or [GFortran](#)?

In that case, several solutions come to the rescue:

- [auditwheel](#) will bundle external libraries into an already-built wheel.
- [delocate](#) does the same on macOS.

Conveniently, `auditwheel` is present on the `manylinux` Docker images. Using `auditwheel` and `delocate` takes just one command. Just tell them about the wheel file(s) and they'll do the rest:

Shell

```
$ auditwheel repair <path-to-wheel.whl> # For manylinux
$ delocate-wheel <path-to-wheel.whl> # For macOS
```

This will detect the needed external libraries through your project's `setup.py` and bundle them in to the wheel as if they were part of the project.

An example of a project that takes advantage of `auditwheel` and `delocate` is [pycld3](#), which provides Python bindings for the Compact Language Detector v3 (CLD3).

The `pycld3` package depends on [libprotobuf](#), which is not a commonly installed library. If you peek inside a [pycld3 macos wheel](#), then you'll see that `libprotobuf.22.dylib` is included there. This is a **dynamically linked shared library** that's bundled into the wheel:

Shell

```
$ unzip -l pycld3-0.20-cp38-cp38-macosx_10_15_x86_64.whl
...
   51  04-10-2020 11:46  cld3/__init__.py
  939984  04-10-2020 07:50  cld3/_cld3.cpython-38-darwin.so
 2375836  04-10-2020 07:50  cld3/.dylibs/libprotobuf.22.dylib
-----
                   8 files
```

The wheel comes prepackaged with `libprotobuf`. A `.dylib` is similar to a Unix `.so` file or Windows `.dll` file, but I admittedly don't know the nitty-gritty of the difference beyond that.

`auditwheel` and `delocate` know to include `libprotobuf` because [setup.py tells them to](#) through the `libraries` argument:

Python

```
setup(
    ...
    libraries=["protobuf"],
    ...
)
```

This means that `auditwheel` and `delocate` save users the trouble of installing `protobuf` as long as they're installing from a platform and Python combination that has a matching wheel.

If you're distributing a package that has external dependencies like this, then you can do your users a favor by using `auditwheel` or `delocate` to save them the extra step of installing the dependencies themselves.

Building Wheels in Continuous Integration

An alternative to building wheels on your local machine is to build them automatically within your project's [CI pipeline](#).

There are myriad CI solutions that integrate with the major code hosting services. Among them are [Appveyor](#), [Azure DevOps](#), [BitBucket Pipelines](#), [Circle CI](#), [GitHub Actions](#), [Jenkins](#), and [Travis CI](#), to name just a few.

The purpose of this tutorial is not to render a judgment as to which CI service is best for building wheels, and any listing of which CI services support which containers would quickly become outdated given the speed at which CI support is evolving. However, this section can help to get you started.

If you're developing a pure-Python package, the `bdist_wheel` step is a blissful one-liner: it's largely irrelevant which container OS and platform you build the wheel on. Virtually all major CI services should enable you to do this in a no-frills fashion by defining steps in a special [YAML file](#) within the project.

For example, here's the syntax you could use with [GitHub Actions](#):

YAML

```
1 name: Python wheels
2 on:
3   release:
4     types:
5       - created
6 jobs:
7   wheels:
8     runs-on: ubuntu-latest
9     steps:
10    - uses: actions/checkout@v2
11    - name: Set up Python 3.x
12      uses: actions/setup-python@v2
13      with:
14        python-version: '3.x'
15    - name: Install dependencies
16      run: python -m pip install --upgrade setuptools wheel
17    - name: Build wheels
18      run: python setup.py bdist_wheel
19    - uses: actions/upload-artifact@v2
20      with:
21        name: dist
22        path: dist
```

In this configuration file, you build a wheel using the following steps:

1. In **line 8**, you specify that the job should run on an Ubuntu machine.
2. In **line 10**, you use the [checkout](#) action to set up your project repository.
3. In **line 14**, you tell the CI runner to use the latest stable version of Python 3.
4. In **line 21**, you request that the resulting wheel be available as an artifact that you can download from the UI once the job completes.

However, if you have a complex project (maybe one with C extensions or Cython code) and you're working to craft a CI/CD pipeline to automatically build wheels, then there will likely be additional steps involved. Here are a few projects through which you can learn by example:

- [yarl](#)
- [msgpack](#)
- [markupsafe](#)
- [cryptography](#)

Many projects roll their own CI configuration. However, some solutions have emerged for reducing the amount of code specified in configuration files to build wheels. You can use the [cibuildwheel](#) tool directly on your CI server to cut down on the lines of code and configuration that it takes to build multiple platform wheels. There's also [multibuild](#), which provides a set of shell scripts for assisting with building wheels on Travis CI and AppVeyor.

Making Sure Your Wheels Spin Right

Building wheels that are structured correctly can be a delicate operation. For instance, if your Python package uses a [src layout](#) and you forget to [specify that properly in setup.py](#), then the resulting wheel might contain a directory in the wrong place.

One check that you can use after `bdist wheel` is the `check-wheel-contents` tool. It looks for common problems

such as the package directory having an abnormal structure or the presence of duplicate files:

Shell

```
$ check-wheel-contents dist/*.whl  
dist/ujson-2.0.3-cp38-cp38-macosx_10_15_x86_64.whl: OK
```

In this case, `check-wheel-contents` indicates that everything with the `ujson` wheel checks out. If not, `stdout` will show a summary of possible issues much like a linter such as `flake8`.

Another way to confirm that the wheel you've built has the right stuff is to use [TestPyPI](#). First, you can upload the package there:

Shell

```
$ python -m twine upload \  
  --repository-url https://test.pypi.org/legacy/ \  
  dist/*
```

Then, you can download the same package for testing as if it were the real thing:

Shell

```
$ python -m pip install \  
  --index-url https://test.pypi.org/simple/ \  
  <pkg-name>
```

This allows you to test your wheel by uploading and then downloading your own project.

Uploading Python Wheels to PyPI

Now it's time to [upload your Python package](#). Since a `sdist` and a `wheel` both get put in the `dist/` directory by default, you can upload them both using the `twine` tool, which is a utility for publishing packages to PyPI:

Shell

```
$ python -m pip install -U twine  
$ python -m twine upload dist/*
```

Since both `sdist` and `bdist_wheel` output to `dist/` by default, you can safely tell `twine` to upload everything under `dist/` using a shell wildcard (`dist/*`).

Conclusion

Understanding the pivotal role that wheels play in the Python ecosystem can make your life easier as both a user and developer of Python packages. Furthermore, increasing your Python literacy when it comes to wheels will help you to better understand what's happening when you install a package and when, in increasingly rare cases, that operation goes awry.

In this tutorial, you learned:

- What wheels are and how they compare to **source distributions**
- How you can use wheels to control the **package installation** process
- What the differences are between **universal**, **pure-Python**, and **platform** wheels
- How to **create and distribute** wheels for your own Python packages

You now have a solid understanding of wheels from both a user's and a developer's perspective. You're well equipped to build your own wheels and make your project's installation process quick, convenient, and stable.

See the section below for some additional reading to dive deeper into the rapidly-expanding wheel ecosystem.

Resources

The [Python Wheels page](#) is dedicated to tracking support for wheels among the 360 most downloaded packages on

PyPI. The adoption rate is pretty respectable at the time of this tutorial, at 331 out of 360, or around 91 percent.

There have been a number of Python Enhancement Proposals (PEPs) that have helped with the specification and evolution of the wheel format:

- [PEP 425 - Compatibility Tags for Built Distributions](#)
- [PEP 427 - The Wheel Binary Package Format 1.0](#)
- [PEP 491 - The Wheel Binary Package Format 1.9](#)
- [PEP 513 - A Platform Tag for Portable Linux Built Distributions](#)
- [PEP 571 - The manylinux2010 Platform Tag](#)
- [PEP 599 - The manylinux2014 Platform Tag](#)

Here's a shortlist of the various wheel packaging tools mentioned in this tutorial:

- [pypa/wheel](#)
- [pypa/auditwheel](#)
- [pypa/manylinux](#)
- [pypa/python-manylinux-demo](#)
- [jwodder/check-wheel-contents](#)
- [matthew-brett/delocate](#)
- [matthew-brett/multibuild](#)
- [joerick/cibuildwheel](#)

The Python documentation has several articles covering wheels and source distributions:

- [Generating Distribution Archives](#)
- [Creating a Source Distribution](#)

Finally, here are a few more useful links from PyPA:

- [Packaging your Project](#)
- [An Overview of Packaging for Python](#)

About Brad Solomon

Brad is a software engineer and a member of the Real Python Tutorial Team.

[» More about Brad](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

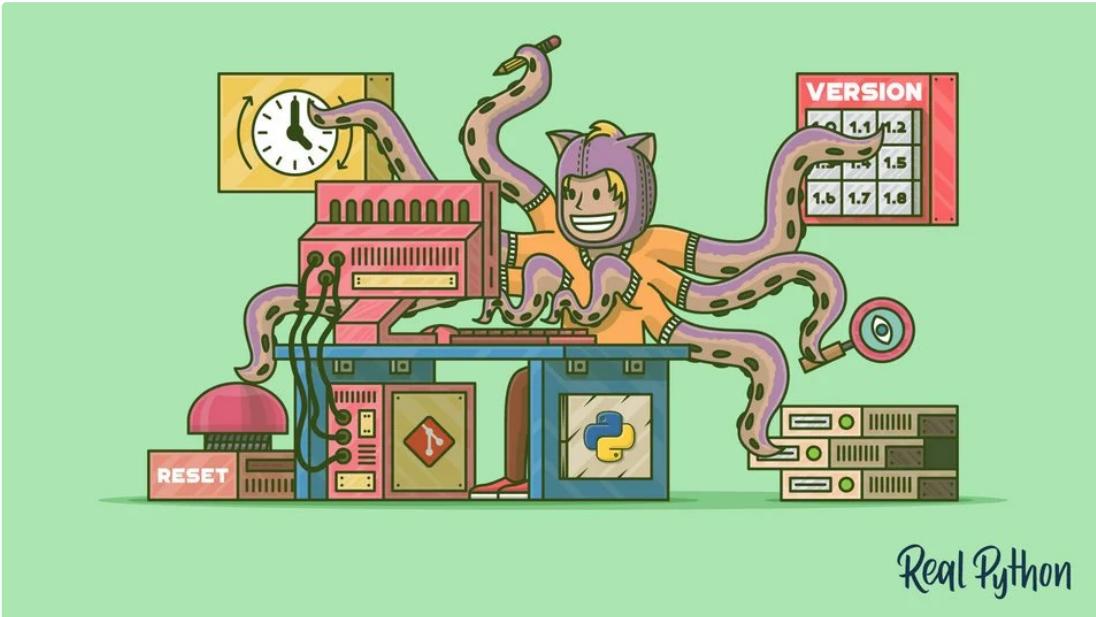
[Jon](#)

[Joanna](#)

[Jacob](#)

Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)



Real Python

Advanced Git Tips for Python Developers

by Jim Anderson 9 Comments advanced tools

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Revision Selection](#)
 - [Relative Referencing](#)
 - [Revision Ranges](#)
- [Handling Interruptions: git stash](#)
 - [git stash save and git stash pop](#)
 - [git stash list](#)
 - [git stash show](#)
 - [git stash pop vs. git stash apply](#)
 - [git stash drop](#)
 - [git stash Example: Pulling Into a Dirty Tree](#)
- [Comparing Revisions: git diff](#)
- [git difftool](#)
- [Changing History](#)
 - [git commit --amend](#)
 - [git rebase](#)
 - [git pull -r](#)
 - [git rebase -i](#)
 - [git revert vs. git reset: Cleaning Up](#)
 - [git clean](#)
- [Resolving Merge Conflicts](#)
 - [diff3 Format](#)
 - [git mergetool](#)
- [Conclusion](#)

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You

[pythonistacafe.com](#)

 PYTHONISTACAFE



If you've done a little work in Git and are starting to understand the basics we covered in [our introduction to Git](#), but you want to learn to be more efficient and have more control, then this is the place for you!

In this tutorial, we'll talk about how to address specific commits and entire ranges of commits, using the stash to save temporary work, comparing different commits, changing history, and how to clean up the mess if something doesn't work out.

This article assumes you've worked through our first Git tutorial or at a minimum understand the basics of what Git is and how it works.

There's a lot of ground to cover, so let's get going.

Revision Selection

There are several options to tell Git which revision (or commit) you want to use. We've already seen that we can use a full SHA (25b09b9ccfe9110aed2d09444f1b50fa2b4c979c) and a short SHA (25b09b9cc) to indicate a revision.

We've also seen how you can use HEAD or a branch name to specify a particular commit as well. There are a few other tricks that Git has up its sleeve, however.

Relative Referencing

Sometimes it's useful to be able to indicate a revision relative to a known position, like HEAD or a branch name. Git provides two operators that, while similar, behave slightly differently.

The first of these is the tilde (~) operator. Git uses tilde to point to a parent of a commit, so HEAD~ indicates the revision before the last one committed. To move back further, you use a number after the tilde: HEAD~3 takes you back three levels.

This works great until we run into merges. Merge commits have two parents, so the ~ just selects the first one. While that works sometimes, there are times when you want to specify the second or later parent. That's why Git has the caret (^) operator.

The ^ operator moves to a specific parent of the specified revision. You use a number to indicate which parent. So HEAD^2 tells Git to select the second parent of the last one committed, **not** the "grandparent." It can be repeated to move back further: HEAD^2^^ takes you back three levels, selecting the second parent on the first step. If you don't give a number, Git assumes 1.

Note: Those of you using Windows will need to escape the ^ character on the DOS command line by using a second ^.

To make life even more fun and less readable, I'll admit, Git allows you to combine these methods, so 25b09b9cc^2~3^3 is a valid way to indicate a revision if you're walking back a tree structure with merges. It takes you to the second parent, then back three revisions from that, and then to the third parent.

Revision Ranges

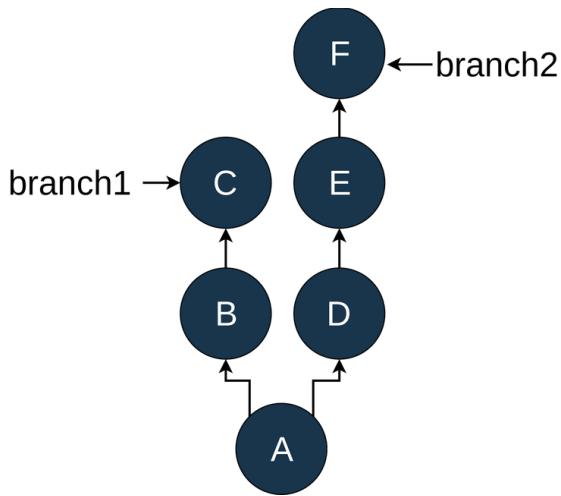
There are a couple of different ways to specify ranges of commits for commands like git log. These don't work exactly like slices in Python, however, so be careful!

Double Dot Notation

The "double dot" method for specifying ranges looks like it sounds: git log b05022238cdf08..60f89368787f0e. It's tempting to think of this as saying "show me all commits after b05022238cdf08 up to and including 60f89368787f0e" and, if b05022238cdf08 is a direct ancestor of 60f89368787f0e, that's exactly what it does.

Note: For the rest of this section, I will be replacing the SHAs of individual commits with capital letters as I think that makes the diagrams a little easier to follow. We'll use this "fake" notation later as well.

It's a bit more powerful than that, however. The double dot notation actually is showing you all commits that are included in the second commit that are not included in the first commit. Let's look at a few diagrams to clarify:



As you can see, we have two branches in our example repo, branch1 and branch2, which diverged after commit A. For starters, let's look at the simple situation. I've modified the log output so that it matches the diagram:

Shell

```
$ git log --oneline D..F
E "Commit message for E"
F "Commit message for F"
```

D..F gives you all of the commits on branch2 **after** commit D.

A more interesting example, and one I learned about while writing this tutorial, is the following:

Shell

```
$ git log --oneline C..F
D "Commit message for D"
E "Commit message for E"
F "Commit message for F"
```

This shows the commits that are part of commit F that are not part of commit c. Because of the structure here, there is not a before/after relationship to these commits because they are on different branches.

Solution: Double Dot Notation

Show/Hide

Triple Dot

Triple dot notation uses, you guessed it, three dots between the revision specifiers. This works in a similar manner to the double dot notation except that it shows all commits that are in **either** revision that are not included in **both** revisions. For our diagram above, using C...F shows you this:

Shell

```
$ git log --oneline C...F
D "Commit message for D"
E "Commit message for E"
F "Commit message for F"
B "Commit message for B"
C "Commit message for C"
```

Double and triple dot notation can be quite powerful when you want to use a range of commits for a command, but they're not as straightforward as many people think.

Branches vs. HEAD vs. SHA

This is probably a good time to review what branches are in Git and how they relate to SHAs and HEAD.

HEAD is the name Git uses to refer to “where your file system is pointing right now.” Most of the time, this will be pointing to a named branch, but it does not have to be. To look at these ideas, let’s walk through an example. Suppose your history looks like this:

At this point, you discover that you accidentally committed a Python logging statement in commit B. Rats. Now, most people would add a new commit, E, push that to master and be done. But you are learning Git and want to fix this the hard way and hide the fact that you made a mistake in the history.

So you move HEAD back to B using `git checkout B`, which looks like this:

You can see that `master` hasn’t changed position, but `HEAD` now points to B. In the Intro to Git tutorial, we talked about the “detached HEAD” state. This is that state again!

Since you want to commit changes, you create a new branch with `git checkout -b temp`:

Now you edit the file and remove the offending log statement. Once that is done, you use `git add` and `git commit --amend` to modify commit B:

Whoa! There's a new commit here called B'. Just like B, it has A as its parent, but C doesn't know anything about it. Now we want master to be based on this new commit, B'.

Because you have a sharp memory, you remember that the `rebase` command does just that. So you get back to the master branch by typing `git checkout master`:

Once you're on master, you can use `git rebase temp` to replay c and d on top of b:

You can see that the rebase created commits c' and d'. c' still has the same changes that c has, and d' has the same changes as d, but they have different SHAs because they are now based on b' instead of b.

As I mentioned earlier, you normally wouldn't go to this much trouble just to fix an errant log statement, but there are times when this approach could be useful, and it does illustrate the differences between HEAD, commits, and branches.

More

Git has even more tricks up its sleeve, but I'll stop here as I've rarely seen the other methods used in the wild. If you'd like to learn about how to do similar operations with more than two branches, checkout the excellent write-up on Revision Selection in the [Pro Git book](#).

Handling Interruptions: `git stash`

One of the Git features I use frequently and find quite handy is the stash. It provides a simple mechanism to save the files you’re working on but are not ready to commit so you can switch to a different task. In this section, you’ll walk through a simple use case first, looking at each of the different commands and options, then you will wrap up with a few other use cases in which git stash really shines.

git stash save and git stash pop

Suppose you’re working on a nasty bug. You’ve got Python logging code in two files, `file1` and `file2`, to help you track it down, and you’ve added `file3` as a possible solution.

In short, the changes to the repo are as follows:

- You’ve edited `file1` and done `git add file1`.
- You’ve edited `file2` but have not added it.
- You’ve created `file3` and have not added it.

You do a `git status` to confirm the condition of the repo:

Shell

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   file1

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   file2

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file3
```

Now a coworker (aren’t they annoying?) walks up and tells you that production is down and it’s “your turn.” You know you can break out your mad `git stash` skills to save you some time and save the day.

You haven’t finished with the work on files 1, 2, and 3, so you really don’t want to commit those changes but you need to get them off of your working directory so you can switch to a different branch to fix that bug. This is the most basic use case for `git stash`.

You can use `git stash save` to “put those changes away” for a little while and return to a clean working directory. The default option for `stash` is `save` so this is usually written as just `git stash`.

When you save something to `stash`, it creates a unique storage spot for those changes and returns your working directory to the state of the last commit. It tells you what it did with a cryptic message:

Shell

```
$ git stash save
Saved working directory and index state WIP on master: 387dcfc adding some files
HEAD is now at 387dcfc adding some files
```

In that output, `master` is the name of the branch, `387dcfc` is the SHA of the last commit, `adding some files` is the commit message for that commit, and `WIP` stands for “work in progress.” The output on your repo will likely be different in those details.

If you do a `status` at this point, it will still show `file3` as an untracked file, but `file1` and `file2` are no longer there:

Shell

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file3

nothing added to commit but untracked files present (use "git add" to track)
```

At this point, as far as Git is concerned, your working directory is “clean,” and you are free to do things like check out a different branch, cherry-pick changes, or anything else you need to.

You go and check out the other branch, fix the bug, earn the admiration of your coworkers, and now are ready to return to this work.

How do you get the last stash back? `git stash pop`!

Using the `pop` command at this point looks like this:

Shell

```
$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   file1
    modified:   file2

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file3

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (71d0f2469db0f1eb9ee7510a9e3e9bd3c1c4211c)
```

Now you can see at the bottom that it has a message about “Dropped refs/stash@{0}”. We’ll talk more about that syntax below, but it’s basically saying that it applied the changes you had stashed and got rid of the stash itself. Before you ask, yes, there is a way to use the stash and **not** get rid of it, but let’s not get ahead of ourselves.

You’ll notice that `file1` used to be in the index but no longer is. By default, `git stash pop` doesn’t maintain the status of changes like that. There is an option to tell it to do so, of course. Add `file1` back to the index and try again:

Shell

```
$ git add file1
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   file1

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   file2

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file3

$ git stash save "another try"
Saved working directory and index state On master: another try
HEAD is now at 387dcfc adding some files
$ git stash pop --index
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   file1

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   file2

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file3

Dropped refs/stash@{0} (aed3a02ae876c1137dd8bab753636a294a3cc43)
```

You can see that the second time we added the `--index` option to the `git pop` command, which tells it to try to maintain the status of whether or not a file is in the index.

In the previous two attempts, you probably noticed that `file3` was not included in your stash. You might want to keep `file3` together with those other changes. Fortunately, there is an option to help you with that: `--include-untracked`.

Assuming we're back to where we were at the end of the last example, we can re-run the command:

Shell

```
$ git stash save --include-untracked "third attempt"
Saved working directory and index state On master: third attempt
HEAD is now at 387dcfc adding some files
$ git status
On branch master
nothing to commit, working directory clean
```

This put the untracked `file3` into the stash with our other changes.

Before we move on, I just want to point out that `save` is the default option for `git stash`. Unless you're specifying a message, which we'll discuss later, you can simply use `git stash`, and it will do a `save`.

`git stash list`

One of the powerful features of `git stash` is that you can have more than one of them. Git stores stashes in a [stack](#), which means that by default it always works with the most recently saved stash. The `git stash list` command will show you the stack of stashes in your local repo. Let's create a couple of stashes so we can see how this works:

Shell

```
$ echo "editing file1" >> file1
$ git stash save "the first save"
Saved working directory and index state On master: the first save
HEAD is now at b3e9b4d adding file1
$ # you can see that stash save cleaned up our working directory
$ # now create a few more stashes by "editing" files and saving them
$ echo "editing file2" >> file2
$ git stash save "the second save"
Saved working directory and index state On master: the second save
HEAD is now at b3e9b4d adding file2
$ echo "editing file3" >> file3
$ git stash save "the third save"
Saved working directory and index state On master: the third save
HEAD is now at b3e9b4d adding file3
$ git status
On branch master
nothing to commit, working directory clean
```

You now have three different stashes saved. Fortunately, Git has a system for dealing with stashes that makes this easy to deal with. The first step of the system is the `git stash list` command:

Shell

```
$ git stash list
stash@{0}: On master: the third save
stash@{1}: On master: the second save
stash@{2}: On master: the first save
```

List shows you the stack of stashes you have in this repo, the newest one first. Notice the `stash@{n}` syntax at the start of each entry? That's the name of that stash. The rest of the `git stash` subcommand will use that name to refer to a specific stash. Generally if you don't give a name, it always assumes you mean the most recent stash, `stash@{0}`. You'll see more of this in a bit.

Another thing I'd like to point out here is that you can see the message we used when we did the `git stash save "message"` command in the listing. This can be quite helpful if you have a number of things stashed.

As we mentioned above, the `save [name]` portion of the `git stash save [name]` command is not required. You can simply type `git stash`, and it defaults to a `save` command, but the auto-generated message doesn't give you much information:

Shell

```
$ echo "more editing file1" >> file1
$ git stash
Saved working directory and index state WIP on master: 387dcfc adding some files
HEAD is now at 387dcfc adding some files
$ git stash list
stash@{0}: WIP on master: 387dcfc adding some files
stash@{1}: On master: the third save
stash@{2}: On master: the second save
stash@{3}: On master: the first save
```

The default message is `WIP on <branch>: <SHA> <commit message>`., which doesn't tell you much. If we had done that for the first three stashes, they all would have had the same message. That's why, for the examples here, I use the full `git stash save <message>` syntax.

git stash show

Okay, so now you have a bunch of stashes, and you might even have meaningful messages describing them, but what if you want to see exactly what's in a particular stash? That's where the `git stash show` command comes in. Using the default options tells you how many files have changed, as well as which files have changed:

Shell

```
$ git stash show stash@{2}
file1 | 1 +
1 file changed, 1 insertion(+)
```

The default options do not tell you what the changes were, however. Fortunately, you can add the `-p`/`--patch` option, and it will show you the diffs in “patch” format:

Shell

```
$ git stash show -p stash@{2}
diff --git a/file1 b/file1
index e212970..04dbd7b 100644
--- a/file1
+++ b/file1
@@ -1 +1,2 @@
 file1
+editing file1
```

Here it shows you that the line “editing file1” was added to `file1`. If you’re not familiar with the patch format for displaying diffs, don’t worry. When you get to the `git difftool` section below, you’ll see how to bring up a visual diff tool on a stash.

git stash pop vs. git stash apply

You saw earlier how to pop the most recent stash back into your working directory by using the `git stash pop` command. You probably guessed that the stash name syntax we saw earlier also applies to the pop command:

Shell

```
$ git stash list
stash@{0}: On master: the third save
stash@{1}: On master: the second save
stash@{2}: On master: the first save
$ git stash pop stash@{1}
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)
  while read line; do echo -n "$line" | wc -c; done<
    modified:   file2

no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{1} (84f7c9890908a1a1bf3c35acfe36a6ecd1f30a2c)
$ git stash list
stash@{0}: On master: the third save
stash@{1}: On master: the first save
```

You can see that the `git stash pop stash@{1}` put “the second save” back into our working directory and collapsed our stack so that only the first and third stashes are there. Notice how “the first save” changed from `stash@{2}` to `stash@{1}` after the pop.

It’s also possible to put a stash onto your working directory but leave it in the stack as well. This is done with `git stash apply`:

Shell

```
$ git stash list
stash@{0}: On master: the third save
stash@{1}: On master: the first save
$ git stash apply stash@{1}
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   file1
    modified:   file2

no changes added to commit (use "git add" and/or "git commit -a")
$ git stash list
stash@{0}: On master: the third save
stash@{1}: On master: the first save
```

This can be handy if you want to apply the same set of changes multiple times. I recently used this while working on prototype hardware. There were changes needed to get the code to work on the particular hardware on my desk, but none of the others. I used `git stash apply` to apply these changes each time I brought down a new copy of master.

git stash drop

The last stash subcommand to look at is `drop`. This is useful when you want to throw away a stash and not apply it to your working directory. It looks like this:

Shell

```
$ git status
On branch master
nothing to commit, working directory clean
$ git stash list
stash@{0}: On master: the third save
stash@{1}: On master: the first save
$ git stash drop stash@{1}
Dropped stash@{1} (9aaa9996bd6aa363e7be723b4712afaae4fc3235)
$ git stash drop
Dropped refs/stash@{0} (194f99db7a8fcc547fd6d9f5fbffe8b896e2267)
$ git stash list
$ git status
On branch master
nothing to commit, working directory clean
```

This dropped the last two stashes, and Git did not change your working directory. There are a couple of things to notice in the above example. First, the `drop` command, like most of the other `git stash` commands, can use the optional `stash@{n}` names. If you don't supply it, Git assumes `stash@{0}`.

The other interesting thing is that the output from the `drop` command gives you a SHA. Like other SHAs in Git, you can make use of this. If, for example, you really meant to do a `pop` and not a `drop` on `stash@{1}` above, you can create a new branch with that SHA it showed you (`9aaa9996`):

Shell

```
$ git branch tmp 9aaa9996
$ git status
On branch master
nothing to commit, working directory clean
$ # use git log <branchname> to see commits on that branch
$ git log tmp
commit 9aaa9996bd6aa363e7be723b4712afaae4fc3235
Merge: b3e9b4d f2d6ecc
Author: Jim Anderson <your_email_here@gmail.com>
Date:   Sat May 12 09:34:29 2018 -0600

  On master: the first save
  [rest of log deleted for brevity]
```

Once you have that branch, you can use the `git merge` or other techniques to get those changes back to your branch. If you didn't save the SHA from the `git drop` command, there are other methods to attempt to recover the changes, but they can get complicated. You can read more about it [here](#).

git stash Example: Pulling Into a Dirty Tree

Let's wrap up this section on `git stash` by looking at one of its uses that wasn't obvious to me at first. Frequently when you're working on a shared branch for a longer period of time, another developer will push changes to the branch that you want to get to your local repo. You'll remember that we use the `git pull` command to do this. However, if you have local changes in files that the `pull` will modify, Git refuses with an error message explaining what went wrong:

Shell

```
error: Your local changes to the following files would be overwritten by merge:  
      <list of files that conflict>  
Please, commit your changes or stash them before you can merge.  
Aborting
```

You could commit this and then do a `pull`, but that would create a merge node, and you might not be ready to commit those files. Now that you know `git stash`, you can use it instead:

Shell

```
$ git stash  
Saved working directory and index state WIP on master: b25fe34 Cleaned up when no TOKEN is present.  
HEAD is now at <SHA> <commit message>  
$ git pull  
Updating <SHA1>..<SHA2>  
Fast-forward  
  <more info here>  
$ git stash pop  
On branch master  
Your branch is up-to-date with 'origin/master'.  
Changes not staged for commit:  
  <rest of stash pop output trimmed>
```

It's entirely possible that doing the `git stash pop` command will produce a merge conflict. If that's the case, you'll need to hand-edit the conflict to resolve it, and then you can proceed. We'll discuss resolving merge conflicts below.

Comparing Revisions: git diff

The `git diff` command is a powerful feature that you'll find yourself using quite frequently. I looked up the list of things it can compare and was surprised by the list. Try typing `git diff --help` if you'd like to see for yourself. I won't cover all of those use cases here, as many of them aren't too common.

This section has several use cases with the `diff` command, which displays on the command line. The next section shows how you can set Git up to use a visual diff tool like Meld, Windiff, BeyondCompare, or even extensions in your IDE. The options for `diff` and `difftool` are the same, so most of the discussion in this section will apply there too, but it's easier to show the output on the command line version.

The most common use of `git diff` is to see what you have modified in your working directory:

Shell

```
$ echo "I'm editing file3 now" >> file3  
$ git diff  
diff --git a/file3 b/file3  
index faf2282..c5dd702 100644  
--- a/file3  
+++ b/file3  
@@ -1,3 +1,4 @@  
{other contents of file3}  
+I'm editing file3 now
```

As you can see, `diff` shows you the diffs in a “[patch](#)” format right on the command line. Once you work through the format, you can see that the + characters indicate that a line has been added to the file, and, as you’d expect, the line `I'm editing file3 now` was added to `file3`.

The default options for `git diff` are to show you what changes are in your working directory that are **not** in your index or in HEAD. If you add the above change to the index and then do `diff`, it shows that there are no diffs:

Shell

```
$ git add file3  
$ git diff  
[no output here]
```

I found this confusing for a while, but I’ve grown to like it. To see the changes that are in the index and staged for the next commit, use the `--staged` option:

Shell

```
$ git diff --staged  
diff --git a/file3 b/file3  
index faf2282..c5dd702 100644  
--- a/file3  
+++ b/file3  
@@ -1,3 +1,4 @@  
 file1  
 file2  
 file3  
+I'm editing file3 now
```

The `git diff` command can also be used to compare any two commits in your repo. This can show you the changes between two SHAs:

Shell

```
$ git diff b3e9b4d 387dcfc  
diff --git a/file3 b/file3  
deleted file mode 100644  
index faf2282..0000000  
--- a/file3  
+++ /dev/null  
@@ -1,3 +0,0 @@  
-file1  
-file2  
-file3
```

You can also use branch names to see the full set of changes between one branch and another:

Shell

```
$ git diff master tmp  
diff --git a/file1 b/file1  
index e212970..04dbd7b 100644  
--- a/file1  
+++ b/file1  
@@ -1 +1,2 @@  
 file1  
+editing file1
```

You can even use any mix of the revision naming methods we looked at above:

Shell

```
$ git diff master^ master
diff --git a/file3 b/file3
new file mode 100644
index 0000000..faf2282
--- /dev/null
+++ b/file3
@@ -0,0 +1,3 @@
+file1
+file2
+file3
```

When you compare two branches, it shows you all of the changes between two branches. Frequently, you only want to see the diffs for a single file. You can restrict the output to a file by listing the file after a -- (two minuses) option:

Shell

```
$ git diff HEAD~3 HEAD
diff --git a/file1 b/file1
index e212970..04dbd7b 100644
--- a/file1
+++ b/file1
@@ -1 +1,2 @@
  file1
+editing file1
diff --git a/file2 b/file2
index 89361a0..91c5d97 100644
--- a/file2
+++ b/file2
@@ -1,2 +1,3 @@
  file1
  file2
+editing file2
diff --git a/file3 b/file3
index faf2282..c5dd702 100644
--- a/file3
+++ b/file3
@@ -1,3 +1,4 @@
  file1
  file2
  file3
+I'm editing file3 now
$ git diff HEAD~3 HEAD -- file3
diff --git a/file3 b/file3
index faf2282..c5dd702 100644
--- a/file3
+++ b/file3
@@ -1,3 +1,4 @@
  file1
  file2
  file3
+I'm editing file3 now
```

There are many, many options for `git diff`, and I won't go into them all, but I do want to explore another use case, which I use frequently, showing the files that were changed in a commit.

In your current repo, the most recent commit on `master` added a line of text to `file1`. You can see that by comparing `HEAD` with `HEAD^`:

Shell

```
$ git diff HEAD^ HEAD
diff --git a/file1 b/file1
index e212970..04dbd7b 100644
--- a/file1
+++ b/file1
@@ -1 +1,2 @@
  file1
+editing file1
```

That's fine for this small example, but frequently the diffs for a commit can be several pages long, and it can get quite difficult to pull out the filenames. Of course, Git has an option to help with that:

Shell

```
$ git diff HEAD^ HEAD --name-only  
file1
```

The `--name-only` option will show you the list of filename that were changed between two commits, but not what changed in those files.

As I said above, there are **many** options and use cases covered by the `git diff` command, and you've just scratched the surface here. Once you have the commands listed above figured out, I encourage you to look at `git diff --help` and see what other tricks you can find. I definitely learned new things preparing this tutorial!

git difftool

Git has a mechanism to use a visual diff tool to show diffs instead of just using the command line format we've seen thus far. All of the options and features you looked at with `git diff` still work here, but it will show the diffs in a separate window, which many people, myself included, find easier to read. For this example, I'm going to use `meld` as the diff tool because it's available on Windows, Mac, and Linux.

Difftool is something that is much easier to use if you set it up properly. Git has a set of config options that control the defaults for `difftool`. You can set these from the shell using the `git config` command:

Shell

```
$ git config --global diff.tool meld  
$ git config --global difftool.prompt false
```

The `prompt` option is one I find important. If you do not specify this, Git will prompt you before it launches the external build tool every time it starts. This can be quite annoying as it does it for every file in a diff, one at a time:

Shell

```
$ git difftool HEAD^ HEAD  
Viewing (1/1): 'python-git-intro/new_section.md'  
Launch 'meld' [Y/n]: y
```

Setting `prompt` to `false` forces Git to launch the tool without asking, speeding up your process and making you that much better!

In the `diff` discussion above, you covered most of the features of `difftool`, but I wanted to add one thing I learned while researching for this article. Do you remember above when you were looking at the `git stash show` command? I mentioned that there was a way to see what is in a given stash visually, and `difftool` is that way. All of the syntax we learned for addressing stashes works with `difftool`:

Shell

```
$ git difftool stash@{1}
```

As with all `stash` subcommands, if you just want to see the latest stash, you can use the `stash` shortcut:

Shell

```
$ git difftool stash
```

Many IDEs and editors have tools that can help with viewing diffs. There is a list of editor-specific tutorials at the end of the [Introduction to Git](#) tutorial.

Changing History

One feature of Git that frightens some people is that it has the ability to change commits. While I can understand their concern, this is part of the tool, and, like any powerful tool, you can cause trouble if you use it unwisely.

We'll talk about several ways to modify commits, but before we do, let's discuss when this is appropriate. In previous sections you saw the difference between your local repo and a remote repo. Commits that you have created but have not pushed are in your local repo only. Commits that other developers have pushed but you have not pulled are in the remote repo only. Doing a push or a pull will get these commits into both repos.

The **only** time you should be thinking about modifying a commit is if it exists on your local repo and not the remote. If you modify a commit that has already been pushed from the remote, you are very likely to have a difficult time pushing or pulling from that remote, and your coworkers will be unhappy with you if you succeed.

That caveat aside, let's talk about how you can modify commits and change history!

git commit --amend

What do you do if you just made a commit but then realize that `flake8` has an error when you run it? Or you spot a typo in the commit message you just entered? Git will allow you to "amend" a commit:

Shell

```
$ git commit -m "I am bad at spilling"
[master 63f74b7] I am bad at spilling
 1 file changed, 4 insertions(+)
$ git commit --amend -m "I am bad at spelling"
[master 951bf2f] I am bad at spelling
 Date: Tue May 22 20:41:27 2018 -0600
 1 file changed, 4 insertions(+)
```

Now if you look at the log after the amend, you'll see that there was only one commit, and it has the correct message:

Shell

```
$ git log
commit 951bf2f45957079f305e8a039dea1771e14b503c
Author: Jim Anderson <your_email_here@gmail.com>
Date:   Tue May 22 20:41:27 2018 -0600

  I am bad at spelling

commit c789957055bd81dd57c09f5329c448112c1398d8
Author: Jim Anderson <your_email_here@gmail.com>
Date:   Tue May 22 20:39:17 2018 -0600

  new message
[rest of log deleted]
```

If you had modified and added files before the amend, those would have been included in the single commit as well. You can see that this is a handy tool for fixing mistakes. I'll warn you again that doing a `git commit --amend` modifies the commit. If the original commit was pushed to a remote repo, someone else may already have based changes on it. That would be a mess, so only use this for commits that are local-only.

git rebase

A rebase operation is similar to a merge, but it can produce a much cleaner history. When you rebase, Git will find the common ancestor between your current branch and the specified branch. It will then take all of the changes after that common ancestor from your branch and "replay" them on top of the other branch. The result will look like you did all of your changes **after** the other branch.

This can be a little hard to visualize, so let's look at some actual commits. For this exercise, I'm going to use the `--oneline` option on the `git log` command to cut down on the clutter. Let's start with a feature branch you've been working on called `my_feature_branch`. Here's the state of that branch:

Shell

```
$ git log --oneline  
143ae7f second feature commit  
aef68dc first feature commit  
2512d27 Common Ancestor Commit
```

You can see that the `--oneline` option, as you might expect, shows just the SHA and the commit message for each commit. Your branch has two commits after the one labeled `2512d27 Common Ancestor Commit`.

You need a second branch if you're going to do a rebase and `master` seems like a good choice. Here's the current state of the `master` branch:

Shell

```
$ git log --oneline master  
23a558c third master commit  
5ec06af second master commit  
190d6af first master commit  
2512d27 Common Ancestor Commit
```

There are three commits on `master` after `2512d27 Common Ancestor Commit`. While you still have `my_feature_branch` checked out, you can do a rebase to put the two feature commits **after** the three commits on `master`:

Shell

```
$ git rebase master  
First, rewinding head to replay your work on top of it...  
Applying: first feature commit  
Applying: second feature commit  
$ git log --oneline  
cf16517 second feature commit  
69f61e9 first feature commit  
23a558c third master commit  
5ec06af second master commit  
190d6af first master commit  
2512d27 Common Ancestor Commit
```

There are two things to notice in this log listing:

- 1) As advertised, the two feature commits are after the three master commits.
- 2) The SHAs of those two feature commits have changed.

The SHAs are different because the repo is slightly different. The commits represent the same changes to the files, but since they were added on top of the changes already in `master`, the state of the repo is different, so they have different SHAs.

If you had done a `merge` instead of a `rebase`, there would have been a new commit with the message `Merge branch 'master' into my_feature_branch`, and the SHAs of the two feature commits would be unchanged. Doing a rebase avoids the extra merge commit and makes your revision history cleaner.

git pull -r

Using a rebase can be a handy tool when you're working on a branch with a different developer, too. If there are changes on the remote, and you have local commits to the same branch, you can use the `-r` option on the `git pull` command. Where a normal `git pull` does a `merge` to the remote branch, `git pull -r` will rebase your commits on top of the changes that were on the remote.

git rebase -i

The `rebase` command has another method of operation. There is a `-i` flag you can add to the `rebase` command that will put it into interactive mode. While this seems confusing at first, it is an amazingly powerful feature that lets you have full control over the list of commits before you push them to a remote. Please remember the warning about not changing the history of commits that have been pushed.

These examples show a basic interactive rebase, but be aware that there are more options and more use cases. The `git rebase --help` command will give you the list and actually does a good job of explaining them.

For this example, you're going to imagine you've been working on your Python library, committing several times to your local repo as you implement a solution, test it, discover a problem and fix it. At the end of this process you have a chain of commits on your local repo that all are part of the new feature. Once you've finished the work, you look at your `git log`:

Shell

```
$ git log --oneline
8bb7af8 implemented feedback from code review
504d520 added unit test to cover new bug
56d1c23 more flake8 clean up
d9b1f9e restructuring to clean up
08dc922 another bug fix
7f82500 pylint cleanup
a113f67 found a bug fixing
3b8a6f2 First attempt at solution
af21a53 [older stuff here]
```

There are several commits here that don't add value to other developers or even to you in the future. You can use `rebase -i` to create a "squash commit" and put all of these into a single point in history.

To start the process, you run `git rebase -i af21a53`, which will bring up an editor with a list of commits and some instructions:

Text

```
pick 3b8a6f2 First attempt at solution
pick a113f67 found a bug fixing
pick 7f82500 pylint cleanup
pick 08dc922 another bug fix
pick d9b1f9e restructuring to clean up
pick 56d1c23 more flake8 clean up
pick 504d520 added unit test to cover new bug
pick 8bb7af8 implemented feedback from code review

# Rebase af21a53..8bb7af8 onto af21a53 (8 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

You'll notice that the commits are listed in reverse order, oldest first. This is the order in which Git will replay the commits on top of `af21a53`. If you just save the file at this point, nothing will change. This is also true if you delete all the text and save the file.

Also, there are several lines starting with a `#` reminding you how to edit this file. These comments can be removed but do not need to be.

But you want to squash all of these commits into one so that "future you" will know that this is the commit that completely added the feature. To do that, you can edit the file to look like this:

Text

```
pick 3b8a6f2 First attempt at solution
squash a113f67 found a bug fixing
s 7f82500 pylint cleanup
s 08dc922 another bug fix
s d9b1f9e restructuring to clean up
s 56d1c23 more flake8 clean up
s 504d520 added unit test to cover new bug
s 8bb7af8 implemented feedback from code review
```

You can use either the full word for the commands, or, as you did after the first two lines, use the single character version. The example above selected to “pick” the oldest commit and the “squash” each of the subsequent commits into that one. If you save and exit the editor, Git will proceed to put all of those commits into one and then will bring up the editor again, listing all of the commit messages for the squashed commit:

Text

```
# This is a combination of 8 commits.
# The first commit's message is:
Implemented feature ABC

# This is the 2nd commit message:

found a bug fixing

# This is the 3rd commit message:

pylint cleanup

# This is the 4th commit message:

another bug fix

[the rest trimmed for brevity]
```

By default a squash commit will have a long commit message with all of the messages from each commit. In your case it's better to reword the first message and delete the rest. Doing that and saving the file will finish the process, and your log will now have only a single commit for this feature:

Shell

```
$ git log --oneline
9a325ad Implemented feature ABC
af21a53 [older stuff here]
```

Cool! You just hid any evidence that you had to do more than one commit to solve this issue. Good work! Be warned that deciding **when** to do a squash merge is frequently more difficult than the actual process. There's a great [article](#) that does a nice job of laying out the complexities.

As you probably guessed, `git rebase -i` will allow you to do far more complex operations. Let's look at one more example.

In the course of a week, you've worked on three different issues, committing changes at various times for each. There's also a commit in there that you regret and would like to pretend never happened. Here's your starting log:

Shell

```
$ git log --oneline
2f0a106 feature 3 commit 3
f0e14d2 feature 2 commit 3
b2eec2c feature 1 commit 3
d6afbee really rotten, very bad commit
6219ba3 feature 3 commit 2
70e07b8 feature 2 commit 2
c08bf37 feature 1 commit 2
c9747ae feature 3 commit 1
fdf23fc feature 2 commit 1
0f05458 feature 1 commit 1
3ca2262 older stuff here
```

Your mission is to get this into three clean commits and remove that one bad one. You can follow the same process, `git rebase -i 3ca2262`, and Git presents you with the command file:

Text

```
pick 0f05458 feature 1 commit 1
pick fdf23fc feature 2 commit 1
pick c9747ae feature 3 commit 1
pick c08bf37 feature 1 commit 2
pick 70e07b8 feature 2 commit 2
pick 6219ba3 feature 3 commit 2
pick d6afbee really rotten, very bad commit
pick b2eec2c feature 1 commit 3
pick f0e14d2 feature 2 commit 3
pick 2f0a106 feature 3 commit 3
```

Interactive rebase allows you to not only specify what to do with each commit but also lets you rearrange them. So, to get to your three commits, you edit the file to look like this:

Text

```
pick 0f05458 feature 1 commit 1
s c08bf37 feature 1 commit 2
s b2eec2c feature 1 commit 3
pick fdf23fc feature 2 commit 1
s 70e07b8 feature 2 commit 2
s f0e14d2 feature 2 commit 3
pick c9747ae feature 3 commit 1
s 6219ba3 feature 3 commit 2
s 2f0a106 feature 3 commit 3
# pick d6afbee really rotten, very bad commit
```

The commits for each feature are grouped together with only one of them being “picked” and the rest “squashed.” Commenting out the bad commit will remove it, but you could have just as easily deleted that line from the file to the same effect.

When you save that file, you’ll get a separate editor session to create the commit message for each of the three squashed commits. If you call them `feature 1`, `feature 2`, and `feature 3`, your log will now have only those three commits, one for each feature:

Shell

```
$ git log --oneline
f700f1f feature 3
443272f feature 2
0ff80ca feature 1
3ca2262 older stuff here
```

Just like any rebase or merge, you might run into conflicts in this process, which you will need to resolve by editing the file, getting the changes correct, `git add`ing the file, and running `git rebase --continue`.

I’ll end this section by pointing out a few things about rebase:

- 1) Creating squash commits is a “nice to have” feature, but you can still work successfully with Git without using it.

2) Merge conflicts on large interactive rebases can be confusing. None of the individual steps are difficult, but there

can be a lot of them

3) We've just scratched the surface on what you can do with `git rebase -i`. There are more features here than most people will ever discover.

git revert vs. git reset: Cleaning Up

Unsurprisingly, Git provides you several methods for cleaning up when you've made a mess. These techniques depend on what state your repo is in and whether or not the mess is local to your repo or has been pushed to a remote.

Let's start by looking at the easy case. You've made a commit that you don't want, and it hasn't been pushed to remote. Start by creating that commit so you know what you're looking at:

Shell

```
$ ls >> file_i_do_not_want
$ git add file_i_do_not_want
$ git commit -m "bad commit"
[master baebe14] bad commit
 2 files changed, 31 insertions(+)
 create mode 100644 file_i_do_not_want
$ git log --oneline
baebe14 bad commit
443272f feature 2
0ff80ca feature 1
3ca2262 older stuff here
```

The example above created a new file, `file_i_do_not_want`, and committed it to the local repo. It has not been pushed to the remote repo yet. The rest of the examples in this section will use this as a starting point.

To manage commits that are on the local repo only, you can use the `git reset` command. There are two options to explore: `--soft` and `--hard`.

The `git reset --soft <SHA>` command tells Git to move HEAD back to the specified SHA. It doesn't change the local file system, and it doesn't change the index. I'll admit when I read that description, it didn't mean much to me, but looking at the example definitely helps:

Shell

```
$ git reset --soft HEAD^
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   file_i_do_not_want

$ git log --oneline
443272f feature 2
0ff80ca feature 1
3ca2262 older stuff here
```

In the example, we reset HEAD to `HEAD^`. Remember that `^` tells Git to step back one commit. The `--soft` option told Git to **not** change the index or the local file system, so the `file_i_do_not_want` is still in the index in the "Changes to be committed:" state. The `git log` command shows that the `bad commit` was removed from the history, though.

That's what the `--soft` option does. Now let's look at the `--hard` option. Let's go back to your original state so that `bad commit` is in the repo again and try `--hard`:

Shell

```
$ git log --oneline
2e9d704 bad commit
443272f feature 2
0ff80ca feature 1
3ca2262 older stuff here
$ git reset --hard HEAD^
HEAD is now at 443272f feature 2
$ git status
On branch master
nothing to commit, working directory clean
$ git log --oneline
443272f feature 2
0ff80ca feature 1
3ca2262 older stuff here
```

There are several things to notice here. First the reset command actually gives you feedback on the `--hard` option where it does not on the `--soft`. I'm not sure of why this is, quite honestly. Also, when we do the `git status` and `git log` afterwards, you see that not only is the `bad commit` gone, but the changes that were in that commit have also been wiped out. The `--hard` option resets you completely back to the SHA you specified.

Now, if you remember the last section about changing history in Git, it's dawned on you that doing a reset to a branch you've already pushed to a remote might be a bad idea. It changes the history and that can really mess up your co-workers.

Git, of course, has a solution for that. The `git revert` command allows you to easily remove the changes from a given commit but does not change history. It does this by doing the inverse of the commit you specify. If you added a line to a file, `git revert` will remove that line from the file. It does this and automatically creates a new "revert commit" for you.

Once again, reset the repo back to the point that `bad commit` is the most recent commit. First confirm what the changes are in `bad commit`:

Shell

```
$ git diff HEAD^
diff --git a/file_i_do_not_want b/file_i_do_not_want
new file mode 100644
index 0000000..6fe5391
--- /dev/null
+++ b/file_i_do_not_want
@@ -0,0 +1,6 @@
+file1
+file2
+file3
+file4
+file_i_do_not_want
+growing_file
```

You can see that we've simply added the `new file_i_do_not_want` to the repo. The lines below `@@ -0,0 +1,6 @@` are the contents of that new file. Now, assuming that this time you've pushed that `bad commit` to master and you don't want your co-workers to hate you, use `revert` to fix that mistake:

Shell

```
$ git revert HEAD
[master 8a53ee4] Revert "bad commit"
 1 file changed, 6 deletions(-)
 delete mode 100644 file_i_do_not_want
```

When you run that command, Git will pop up an editor window allowing you to modify the commit message for the revert commit:

Text

```
Revert "bad commit"

This reverts commit 1fec3f78f7aea20bf99c124e5b75f8cec319de10.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# deleted:    file_i_do_not_want
#
```

Unlike `commit`, `git revert` does not have an option for specifying the commit message on the command line. You can use `-n` to skip the message editing step and tell Git to simply use the default message.

After we revert the bad commit, our log shows a new commit with that message:

Shell

```
$ git log --oneline
8a53ee4 Revert "bad commit"
1fec3f7 bad commit
443272f feature 2
0ff80ca feature 1
3ca2262 older stuff here
```

The “bad commit” is still there. It needs to be there because you don’t want to change history in this case. There’s a new commit, however, which “undoes” the changes that are in that commit.

git clean

There’s another “clean up” command that I find useful, but I want to present it with a caution.

Caution: Using `git clean` can wipe out files that are not committed to the repo that you will not be able to recover.

`git clean` does what you guess it would: it cleans up your local working directory. I’ve found this quite useful when something large goes wrong and I end up with several files on my file system that I do not want.

In its simple form, `git clean` simply removes files that are not “under version control.” This means that files that show up in the `Untracked files` section when you look at `git status` will be removed from the working tree. There is not a way to recover if you do this accidentally, as those files were not in version control.

That’s handy, but what if you want to remove all of the `.pyc` files created with your Python modules? Those are in your `.gitignore` file, so they don’t show up as Untracked and they don’t get deleted by `git clean`.

The `-x` option tells `git clean` to remove untracked and ignored files, so `git clean -x` will take care of that problem. Almost.

Git is a little conservative with the `clean` command and won’t remove untracked directories unless you tell it to do so. Python 3 likes to create `__pycache__` directories, and it’d be nice to clean these up, too. To solve this, you would add the `-d` option. `git clean -xd` will clean up all of the untracked and ignored files and directories.

Now, if you’ve raced ahead and tested this out, you’ve noticed that it doesn’t actually work. Remember that warning I gave at the beginning of this section? Git tries to be cautious when it comes to deleting files that you can’t recover. So, if you try the above command, you see an error message:

Shell

```
$ git clean -xd
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; refusing to clean
```

While it’s possible to change your `git config` files to not require it, most people I’ve talked to simply get used to using the `-f` option along with the others:

Shell

```
$ git clean -xfd
Removing file_to_delete
```

Again, be warned that `git clean -xfd` will remove files that you will not be able to get back, so please use this with caution!

Resolving Merge Conflicts

When you're new to Git, merge conflicts seem like a scary thing, but with a little practice and a few tricks, they can become much easier to deal with.

Let's start with some of the tricks that can make this easier. The first one changes the format of how conflicts are shown.

diff3 Format

We'll walk through a simple example to see what Git does by default and what options we have to make it easier. To do this, create a new file, `merge.py`, that looks like this:

Python

```
def display():
    print("Welcome to my project!")
```

Add and commit this file to your branch `master`, and this will be your baseline commit. You'll create branches that modify this file in different ways, and then you'll see how to resolve the merge conflict.

You now need to create separate branches that will have conflicting changes. You've seen how this is done before, so I won't describe it in detail:

Shell

```
$ git checkout -b mergebranch
Switched to a new branch 'mergebranch'
$ vi merge.py # edit file to change 'project' to 'program'
$ git add merge.py
$ git commit -m "change project to program"
[mergebranch a775c38] change project to program
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
On branch mergebranch
nothing to commit, working directory clean
$ git checkout master
Switched to branch 'master'
$ vi merge.py # edit file to add 'very cool' before project
$ git add merge.py
$ git commit -m "added description of project"
[master ab41ed2] added description of project
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git show-branch master mergebranch
* [master] added description of project
 ! [mergebranch] change project to program
--
* [master] added description of project
 + [mergebranch] change project to program
*+ [master^] baseline for merging
```

At this point you have conflicting changes on `mergebranch` and `master`. Using the `show-branch` command we learned in our Intro tutorial, you can see this visually on the command line:

Shell

```
$ git show-branch master mergebranch
* [master] added description of project
! [mergebranch] change project to program
--
* [master] added description of project
+ [mergebranch] change project to program
*+ [master^] baseline for merging
```

You're on branch `master`, so let's try to merge in `mergebranch`. Since you've made the changes with the intent of creating a merge conflict, let's hope that happens:

Shell

```
$ git merge mergebranch
Auto-merging merge.py
CONFLICT (content): Merge conflict in merge.py
Automatic merge failed; fix conflicts and then commit the result.
```

As you expected, there's a merge conflict. If you look at status, there's a good deal of useful information there. Not only does it say that you're in the middle of a merge, you have unmerged paths, but it also shows you which files are modified, `merge.py`:

Shell

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:  merge.py

no changes added to commit (use "git add" and/or "git commit -a")
```

You have done all that work to get to the point of having a merge conflict. Now you can start learning about how to resolve it! For this first part, you'll be working with the command line tools and your editor. After that, you'll get fancy and look at using visual diff tools to solve the problem.

When you open `merge.py` in your editor, you can see what Git produced:

Python

```
def display():
<<<<< HEAD
    print("Welcome to my very cool project!")
=====
    print("Welcome to my program!")
>>>> mergebranch
```

Git uses diff syntax from Linux to display the conflict. The top portion, between `<<<<< HEAD` and `=====`, are from `HEAD`, which in your case is `master`. The bottom portion, between `=====` and `>>>> mergebranch` are from, you guessed it, `mergebranch`.

Now, in this very simple example, it's pretty easy to remember which changes came from where and how we should merge this, but there's a setting you can enable which will make this easier.

The `diff3` setting modifies the output of merge conflicts to more closely approximate a three-way merge, meaning in this case that it will show you what's in `master`, followed by what it looked like in the common ancestor, followed by what it looks like in `mergebranch`:

Python

```
def display():
<<<<< HEAD
    print("Welcome to my very cool project!")
|||||| merged common ancestors
    print("Welcome to my project!")
=====
    print("Welcome to my program!")
>>>> mergebranch
```

Now that you can see the starting point, “Welcome to my project!”, you can see exactly what change was made on master and what change was made on mergebranch. This might not seem like a big deal on such a simple example, but it can make a huge difference on large conflicts, especially merges where someone else made some of the changes.

You can set this option in Git globally by issuing the following command:

Shell

```
$ git config --global merge.conflictstyle diff3
```

Okay, so you understand how to see the conflict. Let's go through how to fix it. Start by editing the file, removing all of the markers Git added, and making the one conflicting line correct:

Python

```
def display():
    print("Welcome to my very cool program!")
```

You then add your modified file to the index and commit your merge. This will finish the merge process and create the new node:

Shell

```
$ git add merge.py
$ git commit
[master a56a01e] Merge branch 'mergebranch'
$ git log --oneline
a56a01e Merge branch 'mergebranch'
ab41ed2 added description of project
a775c38 change project to program
f29b775 baseline for merging
```

Merge conflicts can happen while you're cherry-picking, too. The process when you are cherry-picking is slightly different. Instead of using the `git commit` command, you use the `git cherry-pick --continue` command. Don't worry, Git will tell you in the status message which command you need to use. You can always go back and check that to be sure.

git mergetool

Similar to `git difftool`, Git will allow you to configure a visual diff tool to deal with three-way merges. It knows about several different tools on different operating systems. You can see the list of tools it knows about on your system by using the command below. On my Linux machine, it shows the following:

Shell

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
  araxis
  gvimdiff
  gvimdiff2
  gvimdiff3
  meld
  vimdiff
  vimdiff2
  vimdiff3

The following tools are valid, but not currently available:
  bc
  bc3
  codecompare
  deltawalker
  diffmerge
  diffuse
  ecmerge
  emerge
  kdiff3
  opendiff
  p4merge
  tkdiff
  tortoiseremerge
  winmerge
  xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

Also similar to `difftool`, you can configure the `mergetool` options globally to make it easier to use:

Shell

```
$ git config --global merge.tool meld
$ git config --global mergetool.prompt false
```

The final option, `mergetool.prompt`, tells Git not to prompt you each time it opens a window. This might not sound annoying, but when your merge involves several files it will prompt you between each of them.

Conclusion

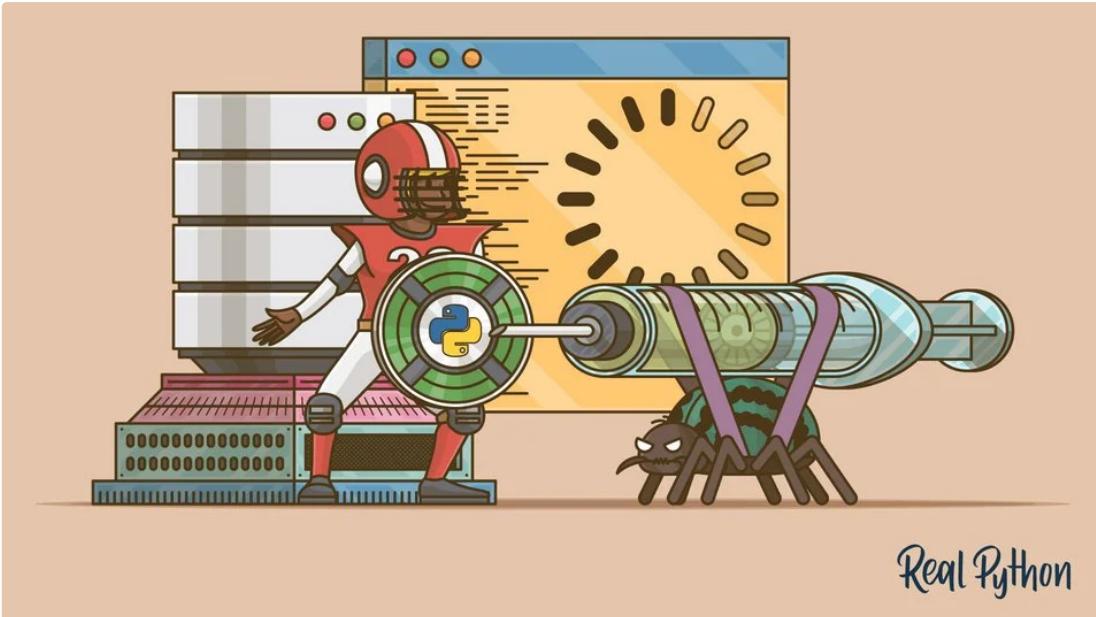
You've covered a lot of ground in these tutorials, but there is so much more to Git. If you'd like to take a deeper dive into Git, I can recommend these resources:

- The free, on-line, [Pro Git](#) is a very handy reference.
- For those of you who like to read on paper, there's a print version of [Pro Git](#), and I found O'Reilly's [Version Control with Git](#) to be useful when I read it.
- `--help` is useful for any of the subcommands you know. `git diff --help` produces almost 1000 lines of information. While portions of these are quite detailed, and some of them assume a deep knowledge of Git, reading the help for commands you use frequently can teach you new tricks on how to use them.

About Jim Anderson

Jim has been programming for a long time in a variety of languages. He has worked on embedded systems, built distributed build systems, done off-shore vendor management, and s in many, many meetings.

[» More about Jim](#)



Real Python

Preventing SQL Injection Attacks With Python

by [Haki Benita](#) ⌚ Sep 30, 2019 💬 4 Comments 🏷️ [best-practices](#) [databases](#) [intermediate](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Understanding Python SQL Injection](#)
- [Setting Up a Database](#)
 - [Creating a Database](#)
 - [Creating a Table With Data](#)
 - [Setting Up a Python Virtual Environment](#)
 - [Connecting to the Database](#)
 - [Executing a Query](#)
- [Using Query Parameters in SQL](#)
- [Exploiting Query Parameters With Python SQL Injection](#)
 - [Crafting Safe Query Parameters](#)
 - [Passing Safe Query Parameters](#)
- [Using SQL Composition](#)
- [Conclusion](#)



Every few years, the Open Web Application Security Project (OWASP) ranks the most critical [web application security risks](#). Since the first report, injection risks have always been on top. Among all injection types, **SQL injection** is one of the most common attack vectors, and arguably the most dangerous. As Python is one of the most popular programming languages in the world, knowing how to protect against Python SQL injection is critical.

In this tutorial, you're going to learn:

- What **Python SQL injection** is and how to prevent it
- How to **compose queries** with both literals and identifiers as parameters
- How to **safely execute queries** in a database

This tutorial is suited for **users of all database engines**. The examples here use PostgreSQL, but the results can be reproduced in other database management systems (such as SQLite, MySQL, Microsoft SQL Server, Oracle, and so on).

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Understanding Python SQL Injection

SQL Injection attacks are such a common security vulnerability that the legendary *xkcd* webcomic devoted a comic to it:



"Exploits of a Mom" (Image: [xkcd](#))

Generating and executing [SQL queries](#) is a common task. However, companies around the world often make horrible mistakes when it comes to composing SQL statements. While the [ORM layer](#) usually composes SQL queries, sometimes you have to write your own.

When you use Python to execute these queries directly into a database, there's a chance you could make mistakes that might compromise your system. In this tutorial, you'll learn how to successfully implement functions that compose dynamic SQL queries *without* putting your system at risk for Python SQL injection.

Setting Up a Database

To get started, you're going to set up a fresh PostgreSQL database and populate it with data. Throughout the tutorial, you'll use this database to witness firsthand how Python SQL injection works.

Creating a Database

First, open your shell and create a new PostgreSQL database owned by the user `postgres`:

Shell

```
$ createdb -O postgres psycopgtest
```

Here you used the command line option `-O` to set the owner of the database to the user `postgres`. You also specified the name of the database, which is `psycopgtest`.

Note: `postgres` is a **special user**, which you would normally reserve for administrative tasks, but for this tutorial, it's fine to use `postgres`. In a real system, however, you should create a separate user to be the owner of the database.

Your new database is ready to go! You can connect to it using `psql`:

Shell

```
$ psql -U postgres -d psycopgtest
psql (11.2, server 10.5)
Type "help" for help.
```

You're now connected to the database `psycopgtest` as the user `postgres`. This user is also the database owner, so you'll have read permissions on every table in the database.

Creating a Table With Data

Next, you need to create a table with some user information and add data to it:

PostgreSQL Console

```
psycopgtest=# CREATE TABLE users (
    username varchar(30),
    admin boolean
);
CREATE TABLE

psycopgtest=# INSERT INTO users
    (username, admin)
VALUES
    ('ran', true),
    ('haki', false);
INSERT 0 2

psycopgtest=# SELECT * FROM users;
 username | admin
-----+-----
 ran     | t
 haki   | f
(2 rows)
```

The table has two columns: `username` and `admin`. The `admin` column indicates whether or not a user has administrative privileges. Your goal is to target the `admin` field and try to abuse it.

Setting Up a Python Virtual Environment

Now that you have a database, it's time to set up your Python environment. For step-by-step instructions on how to do this, check out [Python Virtual Environments: A Primer](#).

Create your virtual environment in a new directory:

Shell

```
(~/src) $ mkdir psycopgtest
(~/src) $ cd psycopgtest
(~/src/psycopgtest) $ python3 -m venv venv
```

After you run this command, a new directory called `venv` will be created. This directory will store all the packages you install inside the virtual environment.

Connecting to the Database

To connect to a database in Python, you need a **database adapter**. Most database adapters follow version 2.0 of the Python Database API Specification [PEP 249](#). Every major database engine has a leading adapter:

Database	Adapter
PostgreSQL	Psycopg
SQLite	sqlite3
Oracle	cx_oracle
MySQL	MySQLdb

To connect to a PostgreSQL database, you'll need to install [Psycopg](#), which is the most popular adapter for PostgreSQL in Python. [Django ORM](#) uses it by default, and it's also supported by [SQLAlchemy](#).

In your terminal, activate the virtual environment and use `pip` to install psycopg:

Shell

```
(~/src/psycopgtest) $ source venv/bin/activate
(~/src/psycopgtest) $ python -m pip install psycopg2>=2.8.0
Collecting psycopg2
  Using cached https://...
    psycopg2-2.8.2.tar.gz
Installing collected packages: psycopg2
  Running setup.py install for psycopg2 ... done
Successfully installed psycopg2-2.8.2
```

Now you're ready to create a connection to your database. Here's the start of your Python script:

Python

```
import psycopg2

connection = psycopg2.connect(
    host="localhost",
    database="psycopgtest",
    user="postgres",
    password=None,
)
connection.set_session(autocommit=True)
```

You used `psycopg2.connect()` to create the connection. This function accepts the following arguments:

- **host** is the [IP address](#) or the DNS of the server where your database is located. In this case, the host is your local machine, or `localhost`.
- **database** is the name of the database to connect to. You want to connect to the database you created earlier, `psycopgtest`.
- **user** is a user with permissions for the database. In this case, you want to connect to the database as the owner, so you pass the user `postgres`.
- **password** is the password for whoever you specified in `user`. In most development environments, users can connect to the local database without a password.

After setting up the connection, you configured the session with `autocommit=True`. Activating `autocommit` means you won't have to manually manage transactions by issuing a `commit` or `rollback`. This is the [default behavior](#) in most ORMs. You use this behavior here as well so that you can focus on composing SQL queries instead of managing transactions.

Note: Django users can get the instance of the connection used by the ORM from `django.db.connection`:

Python

```
from django.db import connection
```

Executing a Query

Now that you have a connection to the database, you're ready to execute a query:

Python

>>>

```
>>> with connection.cursor() as cursor:
...     cursor.execute('SELECT COUNT(*) FROM users')
...     result = cursor.fetchone()
...     print(result)
(2,)
```

You used the `connection` object to create a cursor. Just like a file in Python, `cursor` is implemented as a context manager. When you create the context, a cursor is opened for you to use to send commands to the database. When the context exits, the cursor closes and you can no longer use it.

Note: To learn more about context managers, check out [Python Context Managers](#) and the “with” Statement.

While inside the context, you used `cursor` to execute a query and fetch the results. In this case, you issued a query to count the rows in the `users` table. To fetch the result from the query, you executed `cursor.fetchone()` and received a tuple. Since the query can only return one result, you used `fetchone()`. If the query were to return more than one result, then you’d need to either iterate over `cursor` or use one of the other `fetch*` methods.

Using Query Parameters in SQL

In the previous section, you created a database, established a connection to it, and executed a query. The query you used was **static**. In other words, it had **no parameters**. Now you’ll start to use parameters in your queries.

First, you’re going to implement a function that checks whether or not a user is an admin. `is_admin()` accepts a username and returns that user’s admin status:

Python

```
# BAD EXAMPLE. DON'T DO THIS!
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                admin
            FROM
                users
            WHERE
                username = %s
        """ % username)
        result = cursor.fetchone()
        admin, = result
    return admin
```

This function executes a query to fetch the value of the `admin` column for a given `username`. You used `fetchone()` to return a tuple with a single result. Then, you unpacked this tuple into the variable `admin`. To test your function, check some usernames:

Python

>>>

```
>>> is_admin('haki')
False
>>> is_admin('ran')
True
```

So far so good. The function returned the expected result for both users. But what about non-existing user? Take a look at this [Python traceback](#):

Python

>>>

```
>>> is_admin('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in is_admin
TypeError: cannot unpack non-iterable NoneType object
```

When the user does not exist, a `TypeError` is raised. This is because `.fetchone()` returns `None` when no results are found, and unpacking `None` raises a `TypeError`. The only place you can unpack a tuple is where you populate `admin` from `result`.

To handle non-existing users, create a special case for when `result` is `None`:

Python

```
# BAD EXAMPLE. DON'T DO THIS!
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                admin
            FROM
                users
            WHERE
                username = %s
        """ % username)
        result = cursor.fetchone()

    if result is None:
        # User does not exist
        return False

    admin, = result
    return admin
```

Here, you've added a special case for handling `None`. If `username` does not exist, then the function should return `False`. Once again, test the function on some users:

Python

>>>

```
>>> is_admin('haki')
False
>>> is_admin('ran')
True
>>> is_admin('foo')
False
```

Great! The function can now handle non-existing usernames as well.

Exploiting Query Parameters With Python SQL Injection

In the previous example, you used [string interpolation](#) to generate a query. Then, you executed the query and sent the resulting string directly to the database. However, there's something you may have overlooked during this process.

Think back to the `username` argument you passed to `is_admin()`. What exactly does this variable represent? You might assume that `username` is just a string that represents an actual user's name. As you're about to see, though, an intruder can easily exploit this kind of oversight and cause major harm by performing Python SQL injection.

Try to check if the following user is an admin or not:

Python

>>>

```
>>> is_admin("'; select true; --")
True
```

Wait... What just happened?

Let's take another look at the implementation. Print out the actual query being executed in the database:

Python

>>>

```
>>> print("select admin from users where username = '%s'" % "'; select true; --")
select admin from users where username = ''; select true; --'
```

The resulting text contains three statements. To understand exactly how Python SQL injection works, you need to inspect each part individually. The first statement is as follows:

SQL

```
select admin from users where username = '';
```

This is your intended query. The semicolon (;) terminates the query, so the result of this query does not matter. Next up is the second statement:

SQL

```
select true;
```

This statement was constructed by the intruder. It's designed to always return True.

Lastly, you see this short bit of code:

SQL

```
--'
```

This snippet defuses anything that comes after it. The intruder added the comment symbol (--) to turn everything you might have put after the last placeholder into a comment.

When you execute the function with this argument, *it will always return True*. If, for example, you use this function in your login page, an intruder could log in with the username '`;` `select true;` `--`', and they'll be granted access.

If you think this is bad, it could get worse! Intruders with knowledge of your table structure can use Python SQL injection to cause permanent damage. For example, the intruder can inject an update statement to alter the information in the database:

Python

>>>

```
>>> is_admin('haki')
False
>>> is_admin("'; update users set admin = 'true' where username = 'haki'; select true; --")
True
>>> is_admin('haki')
True
```

Let's break it down again:

SQL

```
';
```

This snippet terminates the query, just like in the previous injection. The next statement is as follows:

SQL

```
update users set admin = 'true' where username = 'haki';
```

This section updates admin to true for user haki.

Finally, there's this code snippet:

SQL

```
select true; --
```

As in the previous example, this piece returns true and comments out everything that follows it.

Why is this worse? Well, if the intruder manages to execute the function with this input, then user haki will become an admin:

PostgreSQL Console

```
psycopgtest=# select * from users;
username | admin
-----+-----
ran     | t
haki   | t
(2 rows)
```

The intruder no longer has to use the hack. They can just log in with the username haki. (If the intruder *really* wanted to cause harm, then they could even issue a `DROP DATABASE` command.)

Before you forget, restore haki back to its original state:

PostgreSQL Console

```
psycopgtest=# update users set admin = false where username = 'haki';
UPDATE 1
```

So, why is this happening? Well, what do you know about the `username` argument? You know it should be a string representing the `username`, but you don't actually check or enforce this assertion. This can be dangerous! It's exactly what attackers are looking for when they try to hack your system.

Crafting Safe Query Parameters

In the previous section, you saw how an intruder can exploit your system and gain admin permissions by using a carefully crafted string. The issue was that you allowed the value passed from the client to be executed directly to the database, without performing any sort of check or validation. [SQL injections](#) rely on this type of vulnerability.

Any time user input is used in a database query, there's a possible vulnerability for SQL injection. The key to preventing Python SQL injection is to make sure the value is being used as the developer intended. In the previous example, you intended for `username` to be used as a string. In reality, it was used as a raw SQL statement.

To make sure values are used as they're intended, you need to **escape** the value. For example, to prevent intruders from injecting raw SQL in the place of a string argument, you can escape quotation marks:

Python

>>>

```
>>> # BAD EXAMPLE. DON'T DO THIS!
>>> username = username.replace('"', "'")
```

This is just one example. There are a lot of special characters and scenarios to think about when trying to prevent Python SQL injection. Lucky for you, modern database adapters, come with built-in tools for preventing Python SQL injection by using **query parameters**. These are used instead of plain string interpolation to compose a query with parameters.

Note: Different adapters, databases, and programming languages refer to query parameters by different names. Common names include **bind variables**, **replacement variables**, and **substitution variables**.

Now that you have a better understanding of the vulnerability, you're ready to rewrite the function using query parameters instead of string interpolation:

Python

```
1 def is_admin(username: str) -> bool:
2     with connection.cursor() as cursor:
3         cursor.execute("""
4             SELECT
5                 admin
6             FROM
7                 users
8             WHERE
9                 username = %(username)s
10            """, {
11                'username': username
12            })
13            result = cursor.fetchone()
14
15        if result is None:
16            # User does not exist
17            return False
18
19        admin, = result
20        return admin
```

Here's what's different in this example:

- In line 9, you used a named parameter `username` to indicate where the `username` should go. Notice how the parameter `username` is no longer surrounded by single quotation marks.
- In line 11, you passed the value of `username` as the second argument to `cursor.execute()`. The connection will use the type and value of `username` when executing the query in the database.

To test this function, try some valid and invalid values, including the dangerous string from before:

```
Python >>>
>>> is_admin('haki')
False
>>> is_admin('ran')
True
>>> is_admin('foo')
False
>>> is_admin("'; select true; --")
False
```

Amazing! The function returned the expected result for all values. What's more, the dangerous string no longer works. To understand why, you can inspect the query generated by `execute()`:

```
Python >>>
>>> with connection.cursor() as cursor:
...     cursor.execute("""
...         SELECT
...             admin
...         FROM
...             users
...         WHERE
...             username = %(username)s
...     """, {
...         'username': "'; select true; --"
...     })
...     print(cursor.query.decode('utf-8'))
SELECT
    admin
FROM
    users
WHERE
    username = '''; select true; --'
```

The connection treated the value of `username` as a string and escaped any characters that might terminate the string and introduce Python SQL injection.

Passing Safe Query Parameters

Database adapters usually offer several ways to pass query parameters. **Named placeholders** are usually the best for readability, but some implementations might benefit from using other options.

Let's take a quick look at some of the right and wrong ways to use query parameters. The following code block shows the types of queries you'll want to avoid:

```
Python
# BAD EXAMPLES. DON'T DO THIS!
cursor.execute("SELECT admin FROM users WHERE username = '" + username + "'");
cursor.execute("SELECT admin FROM users WHERE username = '%s' % username");
cursor.execute("SELECT admin FROM users WHERE username = '{}'".format(username));
cursor.execute(F"SELECT admin FROM users WHERE username = '{username}'");
```

Each of these statements passes `username` from the client directly to the database, without performing any sort of check or validation. This sort of code is ripe for inviting Python SQL injection.

In contrast, these types of queries should be safe for you to execute:

Python

```
# SAFE EXAMPLES. DO THIS!
cursor.execute("SELECT admin FROM users WHERE username = %s", (username,));
cursor.execute("SELECT admin FROM users WHERE username = %(username)s", {'username': username});
```

In these statements, `username` is passed as a named parameter. Now, the database will use the specified type and value of `username` when executing the query, offering protection from Python SQL injection.

Using SQL Composition

So far you've used parameters for literals. **Literals** are values such as numbers, strings, and dates. But what if you have a use case that requires composing a different query—one where the parameter is something else, like a table or column name?

Inspired by the previous example, let's implement a function that accepts the name of a table and returns the number of rows in that table:

Python

```
# BAD EXAMPLE. DON'T DO THIS!
def count_rows(table_name: str) -> int:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                count(*)
            FROM
                %(table_name)s
        """, {
            'table_name': table_name,
        })
        result = cursor.fetchone()

    rowcount, = result
    return rowcount
```

Try to execute the function on your `users` table:

Python

>>>

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in count_rows
psycopg2.errors.SyNTAXERROR: syntax error at or near "'users'"
LINE 5:         'users'
                  ^
```

The command failed to generate the SQL. As you've seen already, the database adapter treats the variable as a string or a literal. A table name, however, is not a plain string. This is where [SQL composition](#) comes in.

You already know it's not safe to use string interpolation to compose SQL. Luckily, Psycopg provides a module called `psycopg.sql` to help you safely compose SQL queries. Let's rewrite the function using `psycopg.sql.SQL()`:

Python

```
from psycopg2 import sql

def count_rows(table_name: str) -> int:
    with connection.cursor() as cursor:
        stmt = sql.SQL("""
            SELECT
                count(*)
            FROM
                {table_name}
        """).format(
            table_name = sql.Identifier(table_name),
        )
        cursor.execute(stmt)
        result = cursor.fetchone()

    rowcount, = result
    return rowcount
```

There are two differences in this implementation. First, you used `sql.SQL()` to compose the query. Then, you used `sql.Identifier()` to annotate the argument value `table_name`. (An **identifier** is a column or table name.)

Note: Users of the popular package [django-debug-toolbar](#) might get an error in the SQL panel for queries composed with `psycopg.sql.SQL()`. A fix is expected for release in [version 2.0](#).

Now, try executing the function on the `users` table:

Python

```
>>> count_rows('users')
2
```

>>>

Great! Next, let's see what happens when the table does not exist:

Python

```
>>> count_rows('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in count_rows
psycopg2.errors.UndefinedTable: relation "foo" does not exist
LINE 5:         "foo"
                  ^
```

>>>

The function throws the `UndefinedTable` exception. In the following steps, you'll use this exception as an indication that your function is safe from a Python SQL injection attack.

Note: The exception `UndefinedTable` was added in [psycopg2 version 2.8](#). If you're working with an earlier version of Psycopg, then you'll get a different exception.

To put it all together, add an option to count rows in the table up to a certain limit. This feature might be useful for very large tables. To implement this, add a `LIMIT` clause to the query, along with query parameters for the limit's value:

Python

```
from psycopg2 import sql

def count_rows(table_name: str, limit: int) -> int:
    with connection.cursor() as cursor:
        stmt = sql.SQL("""
            SELECT
                COUNT(*)
            FROM (
                SELECT
                    1
                FROM
                    {table_name}
                LIMIT
                    {limit}
            ) AS limit_query
        """).format(
            table_name = sql.Identifier(table_name),
            limit = sql.Literal(limit),
        )
        cursor.execute(stmt)
        result = cursor.fetchone()

    rowcount, = result
    return rowcount
```

In this code block, you annotated `limit` using `sql.Literal()`. As in the previous example, `psycopg` will bind all query parameters as literals when using the simple approach. However, when using `sql.SQL()`, you need to explicitly annotate each parameter using either `sql.Identifier()` or `sql.Literal()`.

Note: Unfortunately, the Python API specification does not address the binding of identifiers, only literals. `Psycopg` is the only popular adapter that added the ability to safely compose SQL with both literals and identifiers. This fact makes it even more important to pay close attention when binding identifiers.

Execute the function to make sure that it works:

Python

>>>

```
>>> count_rows('users', 1)
1
>>> count_rows('users', 10)
2
```

Now that you see the function is working, make sure it's also safe:

Python

>>>

```
>>> count_rows("(select 1) as foo; update users set admin = true where name = 'haki'; --", 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 18, in count_rows
psycopg2.errors.UndefinedTable: relation "(select 1) as foo; update users set admin = true where nam
LINE 8:                                     "(select 1) as foo; update users set adm...
                                         ^

```

This traceback shows that `psycopg` escaped the value, and the database treated it as a table name. Since a table with this name doesn't exist, an `UndefinedTable` exception was raised and you were not hacked!

Conclusion

You've successfully implemented a function that composes dynamic SQL *without* putting your system at risk for Python SQL injection! You've used both literals and identifiers in your query without compromising security.

You've learned:

- What **Python SQL injection** is and how it can be exploited
- How to **prevent Python SQL injection** using query parameters

- How to **safely compose SQL statements** that use literals and identifiers as parameters

You're now able to create programs that can withstand attacks from the outside. Go forth and thwart the hackers!

About Haki Benita

Haki is an avid Pythonista and writes for Real Python.

[» More about Haki](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

[Aldren](#)

[Brad](#)

[Geir Arne](#)

[Jaya](#)

[Joanna](#)

Keep Learning

Related Tutorial Categories: [best-practices](#) [databases](#) [intermediate](#)