**\*args**

\*args haben 2 Anwendungsfälle:

1. Übergabe von Parametern in Objektform

1. Wenn die Parameterwerte für eine Funktion in Objektform vorliegen (einer Liste oder einem Tupel) können sie eben in dieser Form an die Funktion übergeben werden.
2. Die Anzahl der Übergabeparameter an die Funktion soll variabel bleiben.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

4 verbleibt ungenutzt als Tupel innerhalb der Funktion und wird über den args-Parameternamen ansprechbar

Durch den Funktionsaufruf mit \*args verkürzt sich gegenüber einem Aufruf mit Parameternamen auch der Quellcode:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

**\*\*kwargs**

Übergabe der Parameter als Dictionary. Über die Keys werden die Werte den Parametern der Funktion zugeordnet.

# Python Modules and Packages – An Introduction

by [John Sturtz](https://realpython.com/python-modules-packages/#author) Apr 17, 2018 [43 Comments](https://realpython.com/python-modules-packages/#reader-comments) [basics](https://realpython.com/tutorials/basics/) [python](https://realpython.com/tutorials/python/)

[Tweet](https://twitter.com/intent/tweet/?text=Check%20out%20this%20%23Python%20tutorial:%20Python%20Modules%20and%20Packages%20%E2%80%93%20An%20Introduction%20by%20@realpython&url=https%3A//realpython.com/python-modules-packages/) [Share](https://facebook.com/sharer/sharer.php?u=https%3A//realpython.com/python-modules-packages/) [Email](mailto:?subject=Python%20article%20for%20you&body=Check%20out%20this%20Python%20tutorial:%0A%0APython%20Modules%20and%20Packages%20%E2%80%93%20An%20Introduction%0A%0Ahttps%3A//realpython.com/python-modules-packages/)

Table of Contents

* [Python Modules: Overview](https://realpython.com/python-modules-packages/#python-modules-overview)
* [The Module Search Path](https://realpython.com/python-modules-packages/#the-module-search-path)
* [The import Statement](https://realpython.com/python-modules-packages/#the-import-statement)
  + [import <module\_name>](https://realpython.com/python-modules-packages/#import-module_name)
  + [from <module\_name> import <name(s)>](https://realpython.com/python-modules-packages/#from-module_name-import-names)
  + [from <module\_name> import <name> as <alt\_name>](https://realpython.com/python-modules-packages/#from-module_name-import-name-as-alt_name)
  + [import <module\_name> as <alt\_name>](https://realpython.com/python-modules-packages/#import-module_name-as-alt_name)
* [The dir() Function](https://realpython.com/python-modules-packages/#the-dir-function)
* [Executing a Module as a Script](https://realpython.com/python-modules-packages/#executing-a-module-as-a-script)
* [Reloading a Module](https://realpython.com/python-modules-packages/#reloading-a-module)
* [Python Packages](https://realpython.com/python-modules-packages/#python-packages)
* [Package Initialization](https://realpython.com/python-modules-packages/#package-initialization)
* [Importing \* From a Package](https://realpython.com/python-modules-packages/#importing-from-a-package)
* [Subpackages](https://realpython.com/python-modules-packages/#subpackages)
* [Conclusion](https://realpython.com/python-modules-packages/#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 Modules and Packages: An Introduction**](https://realpython.com/courses/python-modules-packages/)

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 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**](https://realpython.com/python-namespaces-scope/), which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the [Zen of Python](https://www.python.org/dev/peps/pep-0020) 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](https://realpython.com/bonus/python-cheat-sheet-short/)

## 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**](https://realpython.com/regex-python/)) module.
3. A **built-in** module is intrinsically contained in the interpreter, like the [itertools module](https://realpython.com/python-itertools/).

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**

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:

>>> 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:

import mod

When the interpreter executes the above import statement, it searches for mod.py in a [list](https://realpython.com/python-lists-tuples/) 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](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH) environment variable, if it is set. (The format for PYTHONPATH is OS-dependent but should mimic the [PATH](https://realpython.com/add-python-to-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:

>>> 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:

>>> 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:

>>> 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.

## 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:

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:

>>> 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:

>>> 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:

>>> 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:

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:

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:

>>> 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:

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

>>> 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:

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:

>>> 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.

### 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:

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:

>>> 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:

import <module\_name> as <alt\_name>

>>> 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](https://realpython.com/defining-your-own-python-function/). In that case, the import does not occur until the function is called:

>>> 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:

>>> def bar():

... from mod import \*

...

SyntaxError: import \* only allowed at module level

Lastly, a [try statement with an except ImportError](https://realpython.com/python-exceptions/) clause can be used to guard against unsuccessful import attempts:

>>> try:

... # Non-existent module

... import baz

... except ImportError:

... print('Module not found')

...

Module not found

>>> 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 alphabetically sorted list of names in the current **local symbol table**:

>>> 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:

>>> 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:

>>> import mod

>>> dir(mod)

['Foo', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_',

'\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'a', 'foo', 's']

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_']

>>> from mod import \*

>>> dir()

['Foo', '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_',

'\_\_package\_\_', '\_\_spec\_\_', 'a', 'foo', 's']

## 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**

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:

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**

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:

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:

>>> 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\_\_](https://realpython.com/python-main-function/) 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**

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:

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:

>>> 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**](https://realpython.com/python-testing/)**.** For example, suppose you have created a module fact.py containing a **factorial** function, as follows:

**fact.py**

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:

>>> 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:

C:\Users\john\Documents>python fact.py 6

720

## 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**

a = [100, 200, 300]

print('a =', a)

>>> 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:

>>> 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:

[](https://files.realpython.com/media/pkg1.9af1c7aea48f.png)

Here, there is a directory named pkg that contains two modules, mod1.py and mod2.py. The contents of the modules are:

**mod1.py**

def foo():

print('[mod1] foo()')

class Foo:

pass

**mod2.py**

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:

import <module\_name>[, <module\_name> ...]

>>> import pkg.mod1, pkg.mod2

>>> pkg.mod1.foo()

[mod1] foo()

>>> x = pkg.mod2.Bar()

>>> x

<pkg.mod2.Bar object at 0x033F7290>

from <module\_name> import <name(s)>

>>> from pkg.mod1 import foo

>>> foo()

[mod1] foo()

from <module\_name> import <name> as <alt\_name>

>>> 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:

from <package\_name> import <modules\_name>[, <module\_name> ...]

from <package\_name> import <module\_name> as <alt\_name>

>>> 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:

>>> 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:

>>> 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.

## 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***

print(f'Invoking \_\_init\_\_.py for {\_\_name\_\_}')

A = ['quux', 'corge', 'grault']

Let’s add this file to the pkg directory from the above example:

[](https://files.realpython.com/media/pkg2.dab97c2f9c58.png)

Now when the package is imported, the global list A is initialized:

>>> 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**

def foo():

from pkg import A

print('[mod1] foo() / A = ', A)

class Foo:

pass

>>> 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***

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:

>>> 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](https://www.python.org/dev/peps/pep-0420) 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:

[](https://files.realpython.com/media/pkg3.d2160908ae77.png)

There are now four modules defined in the pkg directory. Their contents are as shown below:

***mod1.py***

def foo():

print('[mod1] foo()')

class Foo:

pass

***mod2.py***

def bar():

print('[mod2] bar()')

class Bar:

pass

***mod3.py***

def baz():

print('[mod3] baz()')

class Baz:

pass

***mod4.py***

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:

>>> 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:

from <package\_name> import \*

What does that do?

>>> 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***

\_\_all\_\_ = [

'mod1',

'mod2',

'mod3',

'mod4'

]

Now from pkg import \* imports all four modules:

>>> 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***

\_\_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\_\_:

>>> 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).

## Subpackages

Packages can contain nested **subpackages** to arbitrary depth. For example, let’s make one more modification to the example **package** directory as follows:

[](https://files.realpython.com/media/pkg4.a830d6e144bf.png)

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:

>>> 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***

def baz():

print('[mod3] baz()')

class Baz:

pass

from pkg.sub\_pkg1.mod1 import foo

foo()

>>> 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***

def baz():

print('[mod3] baz()')

class Baz:

pass

from .. import sub\_pkg1

print(sub\_pkg1)

from ..sub\_pkg1.mod1 import foo

foo()

>>> 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

# Primer on Python Decorators

by [Geir Arne Hjelle](https://realpython.com/primer-on-python-decorators/#author) Aug 22, 2018 [221 Comments](https://realpython.com/primer-on-python-decorators/#reader-comments) [intermediate](https://realpython.com/tutorials/intermediate/) [python](https://realpython.com/tutorials/python/)

[Tweet](https://twitter.com/intent/tweet/?text=Check%20out%20this%20%23Python%20tutorial:%20Primer%20on%20Python%20Decorators%20by%20@realpython&url=https%3A//realpython.com/primer-on-python-decorators/) [Share](https://facebook.com/sharer/sharer.php?u=https%3A//realpython.com/primer-on-python-decorators/) [Email](mailto:?subject=Python%20article%20for%20you&body=Check%20out%20this%20Python%20tutorial:%0A%0APrimer%20on%20Python%20Decorators%0A%0Ahttps%3A//realpython.com/primer-on-python-decorators/)

Table of Contents

* [Functions](https://realpython.com/primer-on-python-decorators/#functions)
  + [First-Class Objects](https://realpython.com/primer-on-python-decorators/#first-class-objects)
  + [Inner Functions](https://realpython.com/primer-on-python-decorators/#inner-functions)
  + [Returning Functions From Functions](https://realpython.com/primer-on-python-decorators/#returning-functions-from-functions)
* [Simple Decorators](https://realpython.com/primer-on-python-decorators/#simple-decorators)
  + [Syntactic Sugar!](https://realpython.com/primer-on-python-decorators/#syntactic-sugar)
  + [Reusing Decorators](https://realpython.com/primer-on-python-decorators/#reusing-decorators)
  + [Decorating Functions With Arguments](https://realpython.com/primer-on-python-decorators/#decorating-functions-with-arguments)
  + [Returning Values From Decorated Functions](https://realpython.com/primer-on-python-decorators/#returning-values-from-decorated-functions)
  + [Who Are You, Really?](https://realpython.com/primer-on-python-decorators/#who-are-you-really)
* [A Few Real World Examples](https://realpython.com/primer-on-python-decorators/#a-few-real-world-examples)
  + [Timing Functions](https://realpython.com/primer-on-python-decorators/#timing-functions)
  + [Debugging Code](https://realpython.com/primer-on-python-decorators/#debugging-code)
  + [Slowing Down Code](https://realpython.com/primer-on-python-decorators/#slowing-down-code)
  + [Registering Plugins](https://realpython.com/primer-on-python-decorators/#registering-plugins)
  + [Is the User Logged In?](https://realpython.com/primer-on-python-decorators/#is-the-user-logged-in)
* [Fancy Decorators](https://realpython.com/primer-on-python-decorators/#fancy-decorators)
  + [Decorating Classes](https://realpython.com/primer-on-python-decorators/#decorating-classes)
  + [Nesting Decorators](https://realpython.com/primer-on-python-decorators/#nesting-decorators)
  + [Decorators With Arguments](https://realpython.com/primer-on-python-decorators/#decorators-with-arguments)
  + [Both Please, But Never Mind the Bread](https://realpython.com/primer-on-python-decorators/#both-please-but-never-mind-the-bread)
  + [Stateful Decorators](https://realpython.com/primer-on-python-decorators/#stateful-decorators)
  + [Classes as Decorators](https://realpython.com/primer-on-python-decorators/#classes-as-decorators)
* [More Real World Examples](https://realpython.com/primer-on-python-decorators/#more-real-world-examples)
  + [Slowing Down Code, Revisited](https://realpython.com/primer-on-python-decorators/#slowing-down-code-revisited)
  + [Creating Singletons](https://realpython.com/primer-on-python-decorators/#creating-singletons)
  + [Caching Return Values](https://realpython.com/primer-on-python-decorators/#caching-return-values)
  + [Adding Information About Units](https://realpython.com/primer-on-python-decorators/#adding-information-about-units)
  + [Validating JSON](https://realpython.com/primer-on-python-decorators/#validating-json)
* [Conclusion](https://realpython.com/primer-on-python-decorators/#conclusion)
* [Further Reading](https://realpython.com/primer-on-python-decorators/#further-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: [**Python Decorators 101**](https://realpython.com/courses/python-decorators-101/)

In this tutorial on decorators, we’ll look at what they are and how to create and use them. Decorators provide a simple syntax for calling [higher-order functions](http://en.wikipedia.org/wiki/Higher-order_function).

By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

This sounds confusing, but it’s really not, especially after you’ve seen a few examples of how decorators work. You can find all the examples from this article [here](https://github.com/realpython/materials/tree/master/primer-on-python-decorators).

**Free Bonus:** [Click here to get access to a free "The Power of Python Decorators" guide](https://realpython.com/primer-on-python-decorators/) that shows you three advanced decorator patterns and techniques you can use to write cleaner and more Pythonic programs.

**Decorators Cheat Sheet:** [Click here to get access to a free three-page Python decorators cheat sheet](https://realpython.com/bonus/decorators-cheatsheet/) that summarizes the techniques explained in this tutorial.

**Decorators Q&A Transcript:** [Click here to get access to a 25-page chat log from our Python decorators Q&A session](https://realpython.com/bonus/decorators-qa-2019/) in the Real Python Community Slack where we discussed common decorator questions.

**Updates:**

* 08/22/2018: Major update adding more examples and more advanced decorators
* 01/12/2016: Updated examples to Python 3 (v3.5.1) syntax and added a new example
* 11/01/2015: Added a brief explanation on the functools.wraps() decorator

## Functions

Before you can understand decorators, you must first understand how functions work. For our purposes, **a function returns a value based on the given arguments**. Here is a very simple example:

>>> def add\_one(number):

... return number + 1

>>> add\_one(2)

3

In general, functions in Python may also have side effects rather than just turning an input into an output. [The print() function](https://realpython.com/python-print/) is a basic example of this: it [returns](https://realpython.com/python-return-statement/) [None](https://realpython.com/null-in-python/) while having the side effect of outputting something to the console. However, to understand decorators, it is enough to think about functions as something that turns given arguments into a value.

**Note:** In [functional programming](https://realpython.com/python-functional-programming/), you work (almost) only with pure functions without side effects. While not a purely functional language, Python supports many of the functional programming concepts, including functions as first-class objects.

### First-Class Objects

In Python, functions are [first-class objects](https://dbader.org/blog/python-first-class-functions). This means that **functions can be passed around and used as arguments**, just like [any other object (string, int, float, list, and so on)](https://realpython.com/python-data-types/). Consider the following three functions:

def say\_hello(name):

return f"Hello {name}"

def be\_awesome(name):

return f"Yo {name}, together we are the awesomest!"

def greet\_bob(greeter\_func):

return greeter\_func("Bob")

Here, say\_hello() and be\_awesome() are regular functions that expect a name given as a string. The greet\_bob() function however, expects a function as its argument. We can, for instance, pass it the say\_hello() or the be\_awesome() function:

>>> greet\_bob(say\_hello)

'Hello Bob'

>>> greet\_bob(be\_awesome)

'Yo Bob, together we are the awesomest!'

Note that greet\_bob(say\_hello) refers to two functions, but in different ways: greet\_bob() and say\_hello. The say\_hello function is named without parentheses. This means that only a reference to the function is passed. The function is not executed. The greet\_bob() function, on the other hand, is written with parentheses, so it will be called as usual.

### Inner Functions

It’s possible to [define functions](https://realpython.com/defining-your-own-python-function/) inside other functions. Such functions are called [inner functions](https://realpython.com/inner-functions-what-are-they-good-for/). Here’s an example of a function with two inner functions:

def parent():

print("Printing from the parent() function")

def first\_child():

print("Printing from the first\_child() function")

def second\_child():

print("Printing from the second\_child() function")

second\_child()

first\_child()

What happens when you call the parent() function? Think about this for a minute. The output will be as follows:

>>> parent()

Printing from the parent() function

Printing from the second\_child() function

Printing from the first\_child() function

Note that the order in which the inner functions are defined does not matter. Like with any other functions, the printing only happens when the inner functions are executed.

Furthermore, the inner functions are not defined until the parent function is called. They are locally scoped to parent(): they only exist inside the parent() function as local [variables](https://realpython.com/python-variables/). Try calling first\_child(). You should get an error:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'first\_child' is not defined

Whenever you call parent(), the inner functions first\_child() and second\_child() are also called. But because of their local scope, they aren’t available outside of the parent() function.

### Returning Functions From Functions

Python also allows you to use functions as return values. The following example returns one of the inner functions from the outer parent() function:

def parent(num):

def first\_child():

return "Hi, I am Emma"

def second\_child():

return "Call me Liam"

if num == 1:

return first\_child

else:

return second\_child

Note that you are returning first\_child without the parentheses. Recall that this means that you are **returning a reference to the function first\_child**. In contrast first\_child() with parentheses refers to the result of evaluating the function. This can be seen in the following example:

>>> first = parent(1)

>>> second = parent(2)

>>> first

<function parent.<locals>.first\_child at 0x7f599f1e2e18>

>>> second

<function parent.<locals>.second\_child at 0x7f599dad5268>

The somewhat cryptic output simply means that the first variable refers to the local first\_child() function inside of parent(), while second points to second\_child().

You can now use first and second as if they are regular functions, even though the functions they point to can’t be accessed directly:

>>> first()

'Hi, I am Emma'

>>> second()

'Call me Liam'

Finally, note that in the earlier example you executed the inner functions within the parent function, for instance first\_child(). However, in this last example, you did not add parentheses to the inner functions—first\_child—upon returning. That way, you got a reference to each function that you could call in the future. Make sense?

## Simple Decorators

Now that you’ve seen that functions are just like any other object in Python, you’re ready to move on and see the magical beast that is the Python decorator. Let’s start with an example:

def my\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.")

return wrapper

def say\_whee():

print("Whee!")

say\_whee = my\_decorator(say\_whee)

Can you guess what happens when you call say\_whee()? Try it:

>>> say\_whee()

Something is happening before the function is called.

Whee!

Something is happening after the function is called.

To understand what’s going on here, look back at the previous examples. We are literally just applying everything you have learned so far.

The so-called decoration happens at the following line:

say\_whee = my\_decorator(say\_whee)

In effect, the name say\_whee now points to the wrapper() inner function. Remember that you return wrapper as a function when you call my\_decorator(say\_whee):

>>> say\_whee

<function my\_decorator.<locals>.wrapper at 0x7f3c5dfd42f0>

However, wrapper() has a reference to the original say\_whee() as func, and calls that function between the two calls to [print()](https://realpython.com/courses/python-print/).

Put simply: **decorators wrap a function, modifying its behavior.**

Before moving on, let’s have a look at a second example. Because wrapper() is a regular Python function, the way a decorator modifies a function can change dynamically. So as not to disturb your neighbors, the following example will only run the decorated code during the day:

from datetime import datetime

def not\_during\_the\_night(func):

def wrapper():

if 7 <= datetime.now().hour < 22:

func()

else:

pass # Hush, the neighbors are asleep

return wrapper

def say\_whee():

print("Whee!")

say\_whee = not\_during\_the\_night(say\_whee)

If you try to call say\_whee() after bedtime, nothing will happen:

>>> say\_whee()

>>>

### Syntactic Sugar!

The way you decorated say\_whee() above is a little clunky. First of all, you end up typing the name say\_whee three times. In addition, the decoration gets a bit hidden away below the definition of the function.

Instead, Python allows you to **use decorators in a simpler way with the @ symbol**, sometimes called the [“pie” syntax](https://www.python.org/dev/peps/pep-0318/#background). The following example does the exact same thing as the first decorator example:

def my\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.")

return wrapper

@my\_decorator

def say\_whee():

print("Whee!")

So, @my\_decorator is just an easier way of saying say\_whee = my\_decorator(say\_whee). It’s how you apply a decorator to a function.

### Reusing Decorators

Recall that a decorator is just a regular Python function. All the usual tools for easy reusability are available. Let’s move the decorator to its own [module](https://realpython.com/python-modules-packages/) that can be used in many other functions.

Create a file called decorators.py with the following content:

def do\_twice(func):

def wrapper\_do\_twice():

func()

func()

return wrapper\_do\_twice

**Note:** You can name your inner function whatever you want, and a generic name like wrapper() is usually okay. You’ll see a lot of decorators in this article. To keep them apart, we’ll name the inner function with the same name as the decorator but with a wrapper\_ prefix.

You can now use this new decorator in other files by doing a regular [import](https://realpython.com/absolute-vs-relative-python-imports/):

from decorators import do\_twice

@do\_twice

def say\_whee():

print("Whee!")

When you run this example, you should see that the original say\_whee() is executed twice:

>>> say\_whee()

Whee!

Whee!

**Free Bonus:** [Click here to get access to a free "The Power of Python Decorators" guide](https://realpython.com/primer-on-python-decorators/) that shows you three advanced decorator patterns and techniques you can use to write cleaner and more Pythonic programs.

### Decorating Functions With Arguments

Say that you have a function that accepts some arguments. Can you still decorate it? Let’s try:

from decorators import do\_twice

@do\_twice

def greet(name):

print(f"Hello {name}")

Unfortunately, running this code raises an error:

>>> greet("World")

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: wrapper\_do\_twice() takes 0 positional arguments but 1 was given

The problem is that the inner function wrapper\_do\_twice() does not take any arguments, but name="World" was passed to it. You could fix this by letting wrapper\_do\_twice() accept one argument, but then it would not work for the say\_whee() function you created earlier.

The solution is to use [\*args and \*\*kwargs](https://realpython.com/python-kwargs-and-args/) in the inner wrapper function. Then it will accept an arbitrary number of positional and keyword arguments. Rewrite decorators.py as follows:

def do\_twice(func):

def wrapper\_do\_twice(\*args, \*\*kwargs):

func(\*args, \*\*kwargs)

func(\*args, \*\*kwargs)

return wrapper\_do\_twice

The wrapper\_do\_twice() inner function now accepts any number of arguments and passes them on to the function it decorates. Now both your say\_whee() and greet() examples works:

>>> say\_whee()

Whee!

Whee!

>>> greet("World")

Hello World

Hello World

### Returning Values From Decorated Functions

What happens to the return value of decorated functions? Well, that’s up to the decorator to decide. Let’s say you decorate a simple function as follows:

from decorators import do\_twice

@do\_twice

def return\_greeting(name):

print("Creating greeting")

return f"Hi {name}"

Try to use it:

>>> hi\_adam = return\_greeting("Adam")

Creating greeting

Creating greeting

>>> print(hi\_adam)

None

Oops, your decorator ate the return value from the function.

Because the do\_twice\_wrapper() doesn’t explicitly return a value, the call return\_greeting("Adam") ended up returning None.

To fix this, you need to **make sure the wrapper function returns the return value of the decorated function**. Change your decorators.py file:

def do\_twice(func):

def wrapper\_do\_twice(\*args, \*\*kwargs):

func(\*args, \*\*kwargs)

return func(\*args, \*\*kwargs)

return wrapper\_do\_twice

The return value from the last execution of the function is returned:

>>> return\_greeting("Adam")

Creating greeting

Creating greeting

'Hi Adam'

### Who Are You, Really?

A great convenience when working with Python, especially in the interactive shell, is its powerful introspection ability. [Introspection](https://en.wikipedia.org/wiki/Type_introspection) is the ability of an object to know about its own attributes at runtime. For instance, a function knows its own name and [documentation](https://realpython.com/documenting-python-code/):

>>> print

<built-in function print>

>>> print.\_\_name\_\_

'print'

>>> help(print)

Help on built-in function print in module builtins:

print(...)

<full help message>

The introspection works for functions you define yourself as well:

>>> say\_whee

<function do\_twice.<locals>.wrapper\_do\_twice at 0x7f43700e52f0>

>>> say\_whee.\_\_name\_\_

'wrapper\_do\_twice'

>>> help(say\_whee)

Help on function wrapper\_do\_twice in module decorators:

wrapper\_do\_twice()

However, after being decorated, say\_whee() has gotten very confused about its identity. It now reports being the wrapper\_do\_twice() inner function inside the do\_twice() decorator. Although technically true, this is not very useful information.

To fix this, decorators should use the [@functools.wraps](https://docs.python.org/library/functools.html#functools.wraps) decorator, which will preserve information about the original function. Update decorators.py again:

import functools

def do\_twice(func):

@functools.wraps(func)

def wrapper\_do\_twice(\*args, \*\*kwargs):

func(\*args, \*\*kwargs)

return func(\*args, \*\*kwargs)

return wrapper\_do\_twice

You do not need to change anything about the decorated say\_whee() function:

>>> say\_whee

<function say\_whee at 0x7ff79a60f2f0>

>>> say\_whee.\_\_name\_\_

'say\_whee'

>>> help(say\_whee)

Help on function say\_whee in module whee:

say\_whee()

Much better! Now say\_whee() is still itself after decoration.

**Technical Detail:** The @functools.wraps decorator [uses](https://github.com/python/cpython/blob/5d4cb54800966947db2e86f65fb109c5067076be/Lib/functools.py#L34) the function functools.update\_wrapper() to update special attributes like \_\_name\_\_ and \_\_doc\_\_ that are used in the introspection.

## A Few Real World Examples

Let’s look at a few more useful examples of decorators. You’ll notice that they’ll mainly follow the same pattern that you’ve learned so far:

import functools

def decorator(func):

@functools.wraps(func)

def wrapper\_decorator(\*args, \*\*kwargs):

# Do something before

value = func(\*args, \*\*kwargs)

# Do something after

return value

return wrapper\_decorator

This formula is a good boilerplate template for building more complex decorators.

**Note:** In later examples, we will assume that these decorators are saved in your decorators.py file as well. Recall that you can download [all the examples in this tutorial](https://github.com/realpython/materials/tree/master/primer-on-python-decorators).

### Timing Functions

Let’s start by creating a @timer decorator. It will [measure the time a function takes to execute](https://realpython.com/python-timer/) and print the duration to the console. Here’s the code:

import functools

import time

def timer(func):

"""Print the runtime of the decorated function"""

@functools.wraps(func)

def wrapper\_timer(\*args, \*\*kwargs):

start\_time = time.perf\_counter() # 1

value = func(\*args, \*\*kwargs)

end\_time = time.perf\_counter() # 2

run\_time = end\_time - start\_time # 3

print(f"Finished {func.\_\_name\_\_!r} in {run\_time:.4f} secs")

return value

return wrapper\_timer

@timer

def waste\_some\_time(num\_times):

for \_ in range(num\_times):

sum([i\*\*2 for i in range(10000)])

This decorator works by storing the time just before the function starts running (at the line marked # 1) and just after the function finishes (at # 2). The time the function takes is then the difference between the two (at # 3). We use the [time.perf\_counter()](https://docs.python.org/library/time.html#time.perf_counter) function, which does a good job of measuring time intervals. Here are some examples of timings:

>>> waste\_some\_time(1)

Finished 'waste\_some\_time' in 0.0010 secs

>>> waste\_some\_time(999)

Finished 'waste\_some\_time' in 0.3260 secs

Run it yourself. Work through the code line by line. Make sure you understand how it works. Don’t worry if you don’t get it, though. Decorators are advanced beings. Try to sleep on it or make a drawing of the program flow.

**Note:** The @timer decorator is great if you just want to get an idea about the runtime of your functions. If you want to do more precise measurements of code, you should instead consider the [timeit module](https://docs.python.org/library/timeit.html) in the standard library. It temporarily disables [garbage collection](https://realpython.com/python-memory-management/#garbage-collection) and runs multiple trials to strip out noise from quick function calls.

### Debugging Code

The following @debug decorator will print the arguments a function is called with as well as its return value every time the function is called:

import functools

def debug(func):

"""Print the function signature and return value"""

@functools.wraps(func)

def wrapper\_debug(\*args, \*\*kwargs):

args\_repr = [repr(a) for a in args] # 1

kwargs\_repr = [f"{k}={v!r}" for k, v in kwargs.items()] # 2

signature = ", ".join(args\_repr + kwargs\_repr) # 3

print(f"Calling {func.\_\_name\_\_}({signature})")

value = func(\*args, \*\*kwargs)

print(f"{func.\_\_name\_\_!r} returned {value!r}") # 4

return value

return wrapper\_debug

The signature is created by joining the [string representations](https://dbader.org/blog/python-repr-vs-str) of all the arguments. The numbers in the following list correspond to the numbered comments in the code:

1. Create a list of the positional arguments. Use repr() to get a nice string representing each argument.
2. Create a list of the keyword arguments. The [f-string](https://realpython.com/python-f-strings/) formats each argument as key=value where the !r specifier means that repr() is used to represent the value.
3. The lists of positional and keyword arguments is joined together to one signature string with each argument separated by a comma.
4. The return value is printed after the function is executed.

Let’s see how the decorator works in practice by applying it to a simple function with one position and one keyword argument:

@debug

def make\_greeting(name, age=None):

if age is None:

return f"Howdy {name}!"

else:

return f"Whoa {name}! {age} already, you are growing up!"

Note how the @debug decorator prints the signature and return value of the make\_greeting() function:

>>> make\_greeting("Benjamin")

Calling make\_greeting('Benjamin')

'make\_greeting' returned 'Howdy Benjamin!'

'Howdy Benjamin!'

>>> make\_greeting("Richard", age=112)

Calling make\_greeting('Richard', age=112)

'make\_greeting' returned 'Whoa Richard! 112 already, you are growing up!'

'Whoa Richard! 112 already, you are growing up!'

>>> make\_greeting(name="Dorrisile", age=116)

Calling make\_greeting(name='Dorrisile', age=116)

'make\_greeting' returned 'Whoa Dorrisile! 116 already, you are growing up!'

'Whoa Dorrisile! 116 already, you are growing up!'

This example might not seem immediately useful since the @debug decorator just repeats what you just wrote. It’s more powerful when applied to small convenience functions that you don’t call directly yourself.

The following example calculates an approximation to the [mathematical constant e](https://en.wikipedia.org/wiki/E_(mathematical_constant)):

import math

from decorators import debug

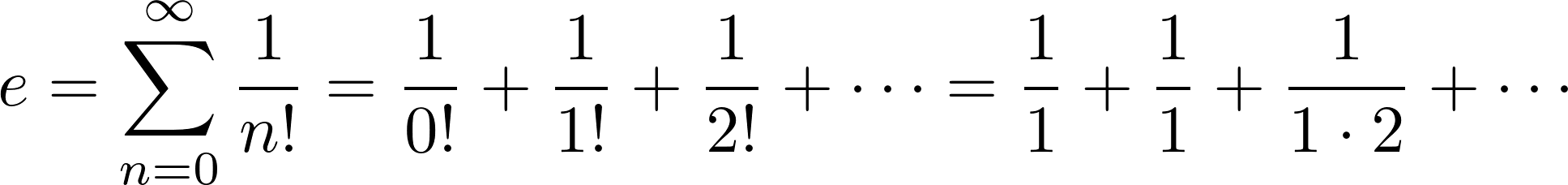
# Apply a decorator to a standard library function

math.factorial = debug(math.factorial)

def approximate\_e(terms=18):

return sum(1 / math.factorial(n) for n in range(terms))

This example also shows how you can apply a decorator to a function that has already been defined. The approximation of e is based on the following [series expansion](https://en.wikipedia.org/wiki/E_(mathematical_constant)):

[](https://files.realpython.com/media/e_series_long.7ce8d6492b4f.png)

When calling the approximate\_e() function, you can see the @debug decorator at work:

>>> approximate\_e(5)

Calling factorial(0)

'factorial' returned 1

Calling factorial(1)

'factorial' returned 1

Calling factorial(2)

'factorial' returned 2

Calling factorial(3)

'factorial' returned 6

Calling factorial(4)

'factorial' returned 24

2.708333333333333

In this example, you get a decent approximation to the true value e = 2.718281828, adding only 5 terms.

### Slowing Down Code

This next example might not seem very useful. Why would you want to slow down your Python code? Probably the most common use case is that you want to rate-limit a function that continuously checks whether a resource—like a web page—has changed. The @slow\_down decorator will sleep one second before it calls the decorated function:

import functools

import time

def slow\_down(func):

"""Sleep 1 second before calling the function"""

@functools.wraps(func)

def wrapper\_slow\_down(\*args, \*\*kwargs):

time.sleep(1)

return func(\*args, \*\*kwargs)

return wrapper\_slow\_down

@slow\_down

def countdown(from\_number):

if from\_number < 1:

print("Liftoff!")

else:

print(from\_number)

countdown(from\_number - 1)

To see the effect of the @slow\_down decorator, you really need to run the example yourself:

>>> countdown(3)

3

2

1

Liftoff!

**Note:** The countdown() function is a recursive function. In other words, it’s a function calling itself. To learn more about recursive functions in Python, see our guide on [Thinking Recursively in Python](https://realpython.com/python-thinking-recursively/).

The @slow\_down decorator always sleeps for one second. [Later](https://realpython.com/primer-on-python-decorators/#slowing-down-code-revisited), you’ll see how to control the rate by passing an argument to the decorator.

### Registering Plugins

Decorators don’t have to wrap the function they’re decorating. They can also simply register that a function exists and return it unwrapped. This can be used, for instance, to create a light-weight plug-in architecture:

import random

PLUGINS = dict()

def register(func):

"""Register a function as a plug-in"""

PLUGINS[func.\_\_name\_\_] = func

return func

@register

def say\_hello(name):

return f"Hello {name}"

@register

def be\_awesome(name):

return f"Yo {name}, together we are the awesomest!"

def randomly\_greet(name):

greeter, greeter\_func = random.choice(list(PLUGINS.items()))

print(f"Using {greeter!r}")

return greeter\_func(name)

The @register decorator simply stores a reference to the decorated function in the global PLUGINS dict. Note that you do not have to write an inner function or use @functools.wraps in this example because you are returning the original function unmodified.

The randomly\_greet() function randomly chooses one of the registered functions to use. Note that the PLUGINS dictionary already contains references to each function object that is registered as a plugin:

>>> PLUGINS

{'say\_hello': <function say\_hello at 0x7f768eae6730>,

'be\_awesome': <function be\_awesome at 0x7f768eae67b8>}

>>> randomly\_greet("Alice")

Using 'say\_hello'

'Hello Alice'

The main benefit of this simple plugin architecture is that you do not need to maintain a list of which plugins exist. That list is created when the plugins register themselves. This makes it trivial to add a new plugin: just define the function and decorate it with @register.

If you are familiar with globals() in Python, you might see some similarities to how the plugin architecture works. globals() gives access to all global variables in the current scope, including your plugins:

>>> globals()

{..., # Lots of variables not shown here.

'say\_hello': <function say\_hello at 0x7f768eae6730>,

'be\_awesome': <function be\_awesome at 0x7f768eae67b8>,

'randomly\_greet': <function randomly\_greet at 0x7f768eae6840>}

Using the @register decorator, you can create your own curated list of interesting variables, effectively hand-picking some functions from globals().

### Is the User Logged In?

The final example before moving on to some fancier decorators is commonly used when working with a web framework. In this example, we are using [Flask](https://realpython.com/tutorials/flask/) to set up a /secret web page that should only be visible to users that are logged in or otherwise authenticated:

from flask import Flask, g, request, redirect, url\_for

import functools

app = Flask(\_\_name\_\_)

def login\_required(func):

"""Make sure user is logged in before proceeding"""

@functools.wraps(func)

def wrapper\_login\_required(\*args, \*\*kwargs):

if g.user is None:

return redirect(url\_for("login", next=request.url))

return func(\*args, \*\*kwargs)

return wrapper\_login\_required

@app.route("/secret")

@login\_required

def secret():

...

While this gives an idea about how to add authentication to your web framework, you should usually not write these types of decorators yourself. For Flask, you can use [the Flask-Login extension](https://flask-login.readthedocs.io/en/latest/#flask_login.login_required) instead, which adds more security and functionality.

## Fancy Decorators

So far, you’ve seen how to create simple decorators. You already have a pretty good understanding of what decorators are and how they work. Feel free to take a break from this article to practice everything you’ve learned.

In the second part of this tutorial, we’ll explore more advanced features, including how to use the following:

* [Decorators on classes](https://realpython.com/primer-on-python-decorators/#decorating-classes)
* [Several decorators on one function](https://realpython.com/primer-on-python-decorators/#nesting-decorators)
* [Decorators with arguments](https://realpython.com/primer-on-python-decorators/#decorators-with-arguments)
* [Decorators that can optionally take arguments](https://realpython.com/primer-on-python-decorators/#both-please-but-never-mind-the-bread)
* [Stateful decorators](https://realpython.com/primer-on-python-decorators/#stateful-decorators)
* [Classes as decorators](https://realpython.com/primer-on-python-decorators/#classes-as-decorators)

### Decorating Classes

There are two different ways you can use decorators on classes. The first one is very close to what you have already done with functions: you can **decorate the methods of a class**. This was [one of the motivations](https://www.python.org/dev/peps/pep-0318/#motivation) for introducing decorators back in the day.

Some commonly used decorators that are even built-ins in Python are [@classmethod, @staticmethod](https://realpython.com/instance-class-and-static-methods-demystified/), and [@property](https://realpython.com/python-property/). The @classmethod and @staticmethod decorators are used to define methods inside a class [namespace](https://realpython.com/python-namespaces-scope/) that are not connected to a particular instance of that class. The @property decorator is used to customize [getters and setters](hhttps://realpython.com/python-getter-setter/) for class attributes. Expand the box below for an example using these decorators.

Let’s define a class where we decorate some of its methods using the [@debug](https://realpython.com/primer-on-python-decorators/#debugging-code) and [@timer](https://realpython.com/primer-on-python-decorators/#timing-functions) decorators from [earlier](https://realpython.com/primer-on-python-decorators/#a-few-real-world-examples):

from decorators import debug, timer

class TimeWaster:

@debug

def \_\_init\_\_(self, max\_num):

self.max\_num = max\_num

@timer

def waste\_time(self, num\_times):

for \_ in range(num\_times):

sum([i\*\*2 for i in range(self.max\_num)])

Using this class, you can see the effect of the decorators:

>>> tw = TimeWaster(1000)

Calling \_\_init\_\_(<time\_waster.TimeWaster object at 0x7efccce03908>, 1000)

'\_\_init\_\_' returned None

>>> tw.waste\_time(999)

Finished 'waste\_time' in 0.3376 secs

The other way to use decorators on classes is to **decorate the whole class**. This is, for example, done in the new [dataclasses module](https://realpython.com/python-data-classes/) in [Python 3.7](https://realpython.com/python37-new-features/):

from dataclasses import dataclass

@dataclass

class PlayingCard:

rank: str

suit: str

The meaning of the syntax is similar to the function decorators. In the example above, you could have done the decoration by writing PlayingCard = dataclass(PlayingCard).

A [common use of class decorators](https://www.python.org/dev/peps/pep-3129/#rationale) is to be a simpler alternative to some use-cases of [metaclasses](https://realpython.com/python-metaclasses/). In both cases, you are changing the definition of a class dynamically.

Writing a class decorator is very similar to writing a function decorator. The only difference is that the decorator will receive a class and not a function as an argument. In fact, all the decorators [you saw above](https://realpython.com/primer-on-python-decorators/#a-few-real-world-examples) will work as class decorators. When you are using them on a class instead of a function, their effect might not be what you want. In the following example, the @timer decorator is applied to a class:

from decorators import timer

@timer

class TimeWaster:

def \_\_init\_\_(self, max\_num):

self.max\_num = max\_num

def waste\_time(self, num\_times):

for \_ in range(num\_times):

sum([i\*\*2 for i in range(self.max\_num)])

Decorating a class does not decorate its methods. Recall that @timer is just shorthand for TimeWaster = timer(TimeWaster).

Here, @timer only measures the time it takes to instantiate the class:

>>> tw = TimeWaster(1000)

Finished 'TimeWaster' in 0.0000 secs

>>> tw.waste\_time(999)

>>>

[Later](https://realpython.com/primer-on-python-decorators/#creating-singletons), you will see an example defining a proper class decorator, namely @singleton, which ensures that there is only one instance of a class.

### Nesting Decorators

You can **apply several decorators** to a function by stacking them on top of each other:

from decorators import debug, do\_twice

@debug

@do\_twice

def greet(name):

print(f"Hello {name}")

Think about this as the decorators being executed in the order they are listed. In other words, @debug calls @do\_twice, which calls greet(), or debug(do\_twice(greet())):

>>> greet("Eva")

Calling greet('Eva')

Hello Eva

Hello Eva

'greet' returned None

Observe the difference if we change the order of @debug and @do\_twice:

from decorators import debug, do\_twice

@do\_twice

@debug

def greet(name):

print(f"Hello {name}")

In this case, @do\_twice will be applied to @debug as well:

>>> greet("Eva")

Calling greet('Eva')

Hello Eva

'greet' returned None

Calling greet('Eva')

Hello Eva

'greet' returned None

### Decorators With Arguments

Sometimes, it’s useful to **pass arguments to your decorators**. For instance, @do\_twice could be extended to a @repeat(num\_times) decorator. The number of times to execute the decorated function could then be given as an argument.

This would allow you to do something like this:

@repeat(num\_times=4)

def greet(name):

print(f"Hello {name}")

>>> greet("World")

Hello World

Hello World

Hello World

Hello World

Think about how you could achieve this.

So far, the name written after the @ has referred to a function object that can be called with another function. To be consistent, you then need repeat(num\_times=4) to return a function object that can act as a decorator. Luckily, you [already know how to return functions](https://realpython.com/primer-on-python-decorators/#returning-functions-from-functions)! In general, you want something like the following:

def repeat(num\_times):

def decorator\_repeat(func):

... # Create and return a wrapper function

return decorator\_repeat

Typically, the decorator creates and returns an inner wrapper function, so writing the example out in full will give you an inner function within an inner function. While this might sound like the programming equivalent of the [Inception movie](https://en.wikipedia.org/wiki/Inception), we’ll untangle it all in a moment:

def repeat(num\_times):

def decorator\_repeat(func):

@functools.wraps(func)

def wrapper\_repeat(\*args, \*\*kwargs):

for \_ in range(num\_times):

value = func(\*args, \*\*kwargs)

return value

return wrapper\_repeat

return decorator\_repeat

It looks a little messy, but we have only put the same decorator pattern you have seen many times by now inside one additional def that handles the arguments to the decorator. Let’s start with the innermost function:

def wrapper\_repeat(\*args, \*\*kwargs):

for \_ in range(num\_times):

value = func(\*args, \*\*kwargs)

return value

This wrapper\_repeat() function takes arbitrary arguments and returns the value of the decorated function, func(). This wrapper function also contains the loop that calls the decorated function num\_times times. This is no different from the earlier wrapper functions you have seen, except that it is using the num\_times parameter that must be supplied from the outside.

One step out, you’ll find the decorator function:

def decorator\_repeat(func):

@functools.wraps(func)

def wrapper\_repeat(\*args, \*\*kwargs):

...

return wrapper\_repeat

Again, decorator\_repeat() looks exactly like the decorator functions you have written earlier, except that it’s named differently. That’s because we reserve the base name—repeat()—for the outermost function, which is the one the user will call.

As you have already seen, the outermost function returns a reference to the decorator function:

def repeat(num\_times):

def decorator\_repeat(func):

...

return decorator\_repeat

There are a few subtle things happening in the repeat() function:

* Defining decorator\_repeat() as an inner function means that repeat() will refer to a function object—decorator\_repeat. Earlier, we used repeat without parentheses to refer to the function object. The added parentheses are necessary when defining decorators that take arguments.
* The num\_times argument is seemingly not used in repeat() itself. But by passing num\_times a [closure](https://realpython.com/inner-functions-what-are-they-good-for/) is created where the value of num\_times is stored until it will be used later by wrapper\_repeat().

With everything set up, let’s see if the results are as expected:

@repeat(num\_times=4)

def greet(name):

print(f"Hello {name}")

>>> greet("World")

Hello World

Hello World

Hello World

Hello World

Just the result we were aiming for.

### Both Please, But Never Mind the Bread

With a little bit of care, you can also define **decorators that can be used both with and without arguments**. Most likely, you don’t need this, but it is nice to have the flexibility.

As you saw in the previous section, when a decorator uses arguments, you need to add an extra outer function. The challenge is for your code to figure out if the decorator has been called with or without arguments.

Since the function to decorate is only passed in directly if the decorator is called without arguments, the function must be an optional argument. This means that the decorator arguments must all be specified by keyword. You can enforce this with the special \* syntax, which means that [all following parameters are keyword-only](https://www.python.org/dev/peps/pep-3102/):

def name(\_func=None, \*, kw1=val1, kw2=val2, ...): # 1

def decorator\_name(func):

... # Create and return a wrapper function.

if \_func is None:

return decorator\_name # 2

else:

return decorator\_name(\_func) # 3

Here, the \_func argument acts as a marker, noting whether the decorator has been called with arguments or not:

1. If name has been called without arguments, the decorated function will be passed in as \_func. If it has been called with arguments, then \_func will be None, and some of the keyword arguments may have been changed from their default values. The \* in the argument list means that the remaining arguments can’t be called as positional arguments.
2. In this case, the decorator was called with arguments. Return a decorator function that can read and return a function.
3. In this case, the decorator was called without arguments. Apply the decorator to the function immediately.

Using this boilerplate on the @repeat decorator in the previous section, you can write the following:

def repeat(\_func=None, \*, num\_times=2):

def decorator\_repeat(func):

@functools.wraps(func)

def wrapper\_repeat(\*args, \*\*kwargs):

for \_ in range(num\_times):

value = func(\*args, \*\*kwargs)

return value

return wrapper\_repeat

if \_func is None:

return decorator\_repeat

else:

return decorator\_repeat(\_func)

Compare this with the original @repeat. The only changes are the added \_func parameter and the if-else at the end.

[Recipe 9.6](https://github.com/dabeaz/python-cookbook/blob/master/src/9/defining_a_decorator_that_takes_an_optional_argument/example.py) of the excellent [Python Cookbook](https://realpython.com/asins/1449340377/) shows an alternative solution using [functools.partial()](https://docs.python.org/library/functools.html#functools.partial).

These examples show that @repeat can now be used with or without arguments:

@repeat

def say\_whee():

print("Whee!")

@repeat(num\_times=3)

def greet(name):

print(f"Hello {name}")

Recall that the default value of num\_times is 2:

>>> say\_whee()

Whee!

Whee!

>>> greet("Penny")

Hello Penny

Hello Penny

Hello Penny

### Stateful Decorators

Sometimes, it’s useful to have **a decorator that can keep track of state**. As a simple example, we will create a decorator that counts the number of times a function is called.

**Note:** In [the beginning of this guide](https://realpython.com/primer-on-python-decorators/#functions), we talked about pure functions returning a value based on given arguments. Stateful decorators are quite the opposite, where the return value will depend on the current state, as well as the given arguments.

In the [next section](https://realpython.com/primer-on-python-decorators/#classes-as-decorators), you will see how to use classes to keep state. But in simple cases, you can also get away with using [function attributes](https://www.python.org/dev/peps/pep-0232/):

import functools

def count\_calls(func):

@functools.wraps(func)

def wrapper\_count\_calls(\*args, \*\*kwargs):

wrapper\_count\_calls.num\_calls += 1

print(f"Call {wrapper\_count\_calls.num\_calls} of {func.\_\_name\_\_!r}")

return func(\*args, \*\*kwargs)

wrapper\_count\_calls.num\_calls = 0

return wrapper\_count\_calls

@count\_calls

def say\_whee():

print("Whee!")

The state—the number of calls to the function—is stored in the function attribute .num\_calls on the wrapper function. Here is the effect of using it:

>>> say\_whee()

Call 1 of 'say\_whee'

Whee!

>>> say\_whee()

Call 2 of 'say\_whee'

Whee!

>>> say\_whee.num\_calls

2

### Classes as Decorators

The typical way to maintain state is by [using classes](https://realpython.com/python3-object-oriented-programming/). In this section, you’ll see how to rewrite the @count\_calls example from the previous section **using a class as a decorator**.

Recall that the decorator syntax @my\_decorator is just an easier way of saying func = my\_decorator(func). Therefore, if my\_decorator is a class, it needs to take func as an argument in its .\_\_init\_\_() method. Furthermore, the class instance needs to be [callable](https://docs.python.org/reference/datamodel.html#emulating-callable-objects) so that it can stand in for the decorated function.

For a class instance to be callable, you implement the special .\_\_call\_\_() method:

class Counter:

def \_\_init\_\_(self, start=0):

self.count = start

def \_\_call\_\_(self):

self.count += 1

print(f"Current count is {self.count}")

The .\_\_call\_\_() method is executed each time you try to call an instance of the class:

>>> counter = Counter()

>>> counter()

Current count is 1

>>> counter()

Current count is 2

>>> counter.count

2

Therefore, a typical implementation of a decorator class needs to implement .\_\_init\_\_() and .\_\_call\_\_():

import functools

class CountCalls:

def \_\_init\_\_(self, func):

functools.update\_wrapper(self, func)

self.func = func

self.num\_calls = 0

def \_\_call\_\_(self, \*args, \*\*kwargs):

self.num\_calls += 1

print(f"Call {self.num\_calls} of {self.func.\_\_name\_\_!r}")

return self.func(\*args, \*\*kwargs)

@CountCalls

def say\_whee():

print("Whee!")

The .\_\_init\_\_() method must store a reference to the function and can do any other necessary initialization. The .\_\_call\_\_() method will be called instead of the decorated function. It does essentially the same thing as the wrapper() function in our earlier examples. Note that you need to use the [functools.update\_wrapper()](https://docs.python.org/library/functools.html#functools.update_wrapper) function instead of @functools.wraps.

This @CountCalls decorator works the same as the one in the previous section:

>>> say\_whee()

Call 1 of 'say\_whee'

Whee!

>>> say\_whee()

Call 2 of 'say\_whee'

Whee!

>>> say\_whee.num\_calls

2

## More Real World Examples

We’ve come a far way now, having figured out how to create all kinds of decorators. Let’s wrap it up, putting our newfound knowledge into creating a few more examples that might actually be useful in the real world.

### Slowing Down Code, Revisited

As noted earlier, our [previous implementation of @slow\_down](https://realpython.com/primer-on-python-decorators/#slowing-down-code) always sleeps for one second. Now you know how to add parameters to decorators, so let’s rewrite @slow\_down using an optional rate argument that controls how long it sleeps:

import functools

import time

def slow\_down(\_func=None, \*, rate=1):

"""Sleep given amount of seconds before calling the function"""

def decorator\_slow\_down(func):

@functools.wraps(func)

def wrapper\_slow\_down(\*args, \*\*kwargs):

time.sleep(rate)

return func(\*args, \*\*kwargs)

return wrapper\_slow\_down

if \_func is None:

return decorator\_slow\_down

else:

return decorator\_slow\_down(\_func)

We’re using the boilerplate introduced in the [Both Please, But Never Mind the Bread](https://realpython.com/primer-on-python-decorators/#both-please-but-never-mind-the-bread) section to make @slow\_down callable both with and without arguments. The same recursive countdown() function [as earlier](https://realpython.com/primer-on-python-decorators/#slowing-down-code) now sleeps two seconds between each count:

@slow\_down(rate=2)

def countdown(from\_number):

if from\_number < 1:

print("Liftoff!")

else:

print(from\_number)

countdown(from\_number - 1)

As before, you must run the example yourself to see the effect of the decorator:

>>> countdown(3)

3

2

1

Liftoff!

### Creating Singletons

A singleton is a class with only one instance. There are several singletons in Python that you use frequently, including None, True, and False. It is the fact that None is a singleton that allows you to compare for None using the [is keyword](https://realpython.com/python-is-identity-vs-equality/), like you saw in the [Both Please](https://realpython.com/primer-on-python-decorators/#both-please-but-never-mind-the-bread) section:

if \_func is None:

return decorator\_name

else:

return decorator\_name(\_func)

Using is returns True only for objects that are the exact same instance. The following @singleton decorator turns a class into a singleton by storing the first instance of the class as an attribute. Later attempts at creating an instance simply return the stored instance:

import functools

def singleton(cls):

"""Make a class a Singleton class (only one instance)"""

@functools.wraps(cls)

def wrapper\_singleton(\*args, \*\*kwargs):

if not wrapper\_singleton.instance:

wrapper\_singleton.instance = cls(\*args, \*\*kwargs)

return wrapper\_singleton.instance

wrapper\_singleton.instance = None

return wrapper\_singleton

@singleton

class TheOne:

pass

As you see, this class decorator follows the same template as our function decorators. The only difference is that we are using cls instead of func as the parameter name to indicate that it is meant to be a class decorator.

Let’s see if it works:

>>> first\_one = TheOne()

>>> another\_one = TheOne()

>>> id(first\_one)

140094218762280

>>> id(another\_one)

140094218762280

>>> first\_one is another\_one

True

It seems clear that first\_one is indeed the exact same instance as another\_one.

**Note:** Singleton classes are not really used as often in Python as in other languages. The effect of a singleton is usually better implemented as a global variable in a module.

### Caching Return Values

Decorators can provide a nice mechanism for [caching](https://en.wikipedia.org/wiki/Cache_%28computing%29) and [memoization](https://en.wikipedia.org/wiki/Memoization). As an example, let’s look at a [recursive](https://realpython.com/python-thinking-recursively/) definition of the [Fibonacci sequence](https://en.wikipedia.org/wiki/Fibonacci_number):

from decorators import count\_calls

@count\_calls

def fibonacci(num):

if num < 2:

return num

return fibonacci(num - 1) + fibonacci(num - 2)

While the implementation is simple, its runtime performance is terrible:

>>> fibonacci(10)

<Lots of output from count\_calls>

55

>>> fibonacci.num\_calls

177

To calculate the tenth Fibonacci number, you should really only need to calculate the preceding Fibonacci numbers, but this implementation somehow needs a whopping 177 calculations. It gets worse quickly: 21891 calculations are needed for fibonacci(20) and almost 2.7 million calculations for the 30th number. This is because the code keeps recalculating Fibonacci numbers that are already known.

The usual solution is to implement Fibonacci numbers using a [for loop](https://realpython.com/python-for-loop/) and a lookup table. However, simple caching of the calculations will also do the trick:

import functools

from decorators import count\_calls

def cache(func):

"""Keep a cache of previous function calls"""

@functools.wraps(func)

def wrapper\_cache(\*args, \*\*kwargs):

cache\_key = args + tuple(kwargs.items())

if cache\_key not in wrapper\_cache.cache:

wrapper\_cache.cache[cache\_key] = func(\*args, \*\*kwargs)

return wrapper\_cache.cache[cache\_key]

wrapper\_cache.cache = dict()

return wrapper\_cache

@cache

@count\_calls

def fibonacci(num):

if num < 2:

return num

return fibonacci(num - 1) + fibonacci(num - 2)

The cache works as a lookup table, so now fibonacci() only does the necessary calculations once:

>>> fibonacci(10)

Call 1 of 'fibonacci'

...

Call 11 of 'fibonacci'

55

>>> fibonacci(8)

21

Note that in the final call to fibonacci(8), no new calculations were needed, since the eighth Fibonacci number had already been calculated for fibonacci(10).

In the standard library, a [Least Recently Used (LRU) cache](https://realpython.com/lru-cache-python/) is available as [@functools.lru\_cache](https://docs.python.org/library/functools.html#functools.lru_cache).

This decorator has more features than the one you saw above. You should use @functools.lru\_cache instead of writing your own cache decorator:

import functools

@functools.lru\_cache(maxsize=4)

def fibonacci(num):

print(f"Calculating fibonacci({num})")

if num < 2:

return num

return fibonacci(num - 1) + fibonacci(num - 2)

The maxsize parameter specifies how many recent calls are cached. The default value is 128, but you can specify maxsize=None to cache all function calls. However, be aware that this can cause memory problems if you are caching many large objects.

You can use the .cache\_info() method to see how the cache performs, and you can tune it if needed. In our example, we used an artificially small maxsize to see the effect of elements being removed from the cache:

>>> fibonacci(10)

Calculating fibonacci(10)

Calculating fibonacci(9)

Calculating fibonacci(8)

Calculating fibonacci(7)

Calculating fibonacci(6)

Calculating fibonacci(5)

Calculating fibonacci(4)

Calculating fibonacci(3)

Calculating fibonacci(2)

Calculating fibonacci(1)

Calculating fibonacci(0)

55

>>> fibonacci(8)

21

>>> fibonacci(5)

Calculating fibonacci(5)

Calculating fibonacci(4)

Calculating fibonacci(3)

Calculating fibonacci(2)

Calculating fibonacci(1)

Calculating fibonacci(0)

5

>>> fibonacci(8)

Calculating fibonacci(8)

Calculating fibonacci(7)

Calculating fibonacci(6)

21

>>> fibonacci(5)

5

>>> fibonacci.cache\_info()

CacheInfo(hits=17, misses=20, maxsize=4, currsize=4)

### Adding Information About Units

The following example is somewhat similar to the [Registering Plugins](https://realpython.com/primer-on-python-decorators/#registering-plugins) example from earlier, in that it does not really change the behavior of the decorated function. Instead, it simply adds unit as a function attribute:

def set\_unit(unit):

"""Register a unit on a function"""

def decorator\_set\_unit(func):

func.unit = unit

return func

return decorator\_set\_unit

The following example calculates the volume of a cylinder based on its radius and height in centimeters:

import math

@set\_unit("cm^3")

def volume(radius, height):

return math.pi \* radius\*\*2 \* height

This .unit function attribute can later be accessed when needed:

>>> volume(3, 5)

141.3716694115407

>>> volume.unit

'cm^3'

Note that you could have achieved something similar using [function annotations](https://www.python.org/dev/peps/pep-3107/):

import math

def volume(radius, height) -> "cm^3":

return math.pi \* radius\*\*2 \* height

However, since annotations are [used for type hints](https://www.python.org/dev/peps/pep-0484/), it would be hard to combine such units as annotations with [static type checking](https://realpython.com/python-type-checking/#static-type-checking).

Units become even more powerful and fun when connected with a library that can convert between units. One such library is [pint](http://pint.readthedocs.io/). With pint installed ([pip install Pint](https://pypi.org/project/Pint/)), you can for instance convert the volume to cubic inches or gallons:

>>> import pint

>>> ureg = pint.UnitRegistry()

>>> vol = volume(3, 5) \* ureg(volume.unit)

>>> vol

<Quantity(141.3716694115407, 'centimeter \*\* 3')>

>>> vol.to("cubic inches")

<Quantity(8.627028576414954, 'inch \*\* 3')>

>>> vol.to("gallons").m # Magnitude

0.0373464440537444

You could also modify the decorator to return a pint [Quantity](https://pint.readthedocs.io/en/latest/tutorial.html) directly. Such a Quantity is made by multiplying a value with the unit. In pint, units must be looked up in a UnitRegistry. The registry is stored as a function attribute to avoid cluttering the namespace:

def use\_unit(unit):

"""Have a function return a Quantity with given unit"""

use\_unit.ureg = pint.UnitRegistry()

def decorator\_use\_unit(func):

@functools.wraps(func)

def wrapper\_use\_unit(\*args, \*\*kwargs):

value = func(\*args, \*\*kwargs)

return value \* use\_unit.ureg(unit)

return wrapper\_use\_unit

return decorator\_use\_unit

@use\_unit("meters per second")

def average\_speed(distance, duration):

return distance / duration

With the @use\_unit decorator, converting units is practically effortless:

>>> bolt = average\_speed(100, 9.58)

>>> bolt

<Quantity(10.438413361169102, 'meter / second')>

>>> bolt.to("km per hour")

<Quantity(37.578288100208766, 'kilometer / hour')>

>>> bolt.to("mph").m # Magnitude

23.350065679064745

### Validating JSON

Let’s look at one last use case. Take a quick look at the following [Flask](https://realpython.com/tutorials/flask/) route handler:

@app.route("/grade", methods=["POST"])

def update\_grade():

json\_data = request.get\_json()

if "student\_id" not in json\_data:

abort(400)

# Update database

return "success!"

Here we ensure that the key student\_id is part of the request. Although this validation works, it really does not belong in the function itself. Plus, perhaps there are other routes that use the exact same validation. So, let’s keep it [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) and abstract out any unnecessary logic with a decorator. The following @validate\_json decorator will do the job:

from flask import Flask, request, abort

import functools

app = Flask(\_\_name\_\_)

def validate\_json(\*expected\_args): # 1

def decorator\_validate\_json(func):

@functools.wraps(func)

def wrapper\_validate\_json(\*args, \*\*kwargs):

json\_object = request.get\_json()

for expected\_arg in expected\_args: # 2

if expected\_arg not in json\_object:

abort(400)

return func(\*args, \*\*kwargs)

return wrapper\_validate\_json

return decorator\_validate\_json

In the above code, the decorator takes a variable length list as an argument so that we can pass in as many string arguments as necessary, each representing a key used to validate the [JSON](https://realpython.com/python-json/) data:

1. The list of keys that must be present in the JSON is given as arguments to the decorator.
2. The wrapper function validates that each expected key is present in the JSON data.

The route handler can then focus on its real job—updating grades—as it can safely assume that JSON data are valid:

@app.route("/grade", methods=["POST"])

@validate\_json("student\_id")

def update\_grade():

json\_data = request.get\_json()

# Update database.

return "success!"

## Conclusion

This has been quite a journey! You started this tutorial by looking a little closer at functions, particularly how they can be defined inside other functions and passed around just like any other Python object. Then you learned about decorators and how to write them such that:

* They can be reused.
* They can decorate functions with arguments and return values.
* They can use @functools.wraps to look more like the decorated function.

In the second part of the tutorial, you saw more advanced decorators and learned how to:

* Decorate classes
* Nest decorators
* Add arguments to decorators
* Keep state within decorators
* Use classes as decorators

You saw that, to define a decorator, you typically define a function returning a wrapper function. The wrapper function uses \*args and \*\*kwargs to pass on arguments to the decorated function. If you want your decorator to also take arguments, you need to nest the wrapper function inside another function. In this case, you usually end up with three return statements.