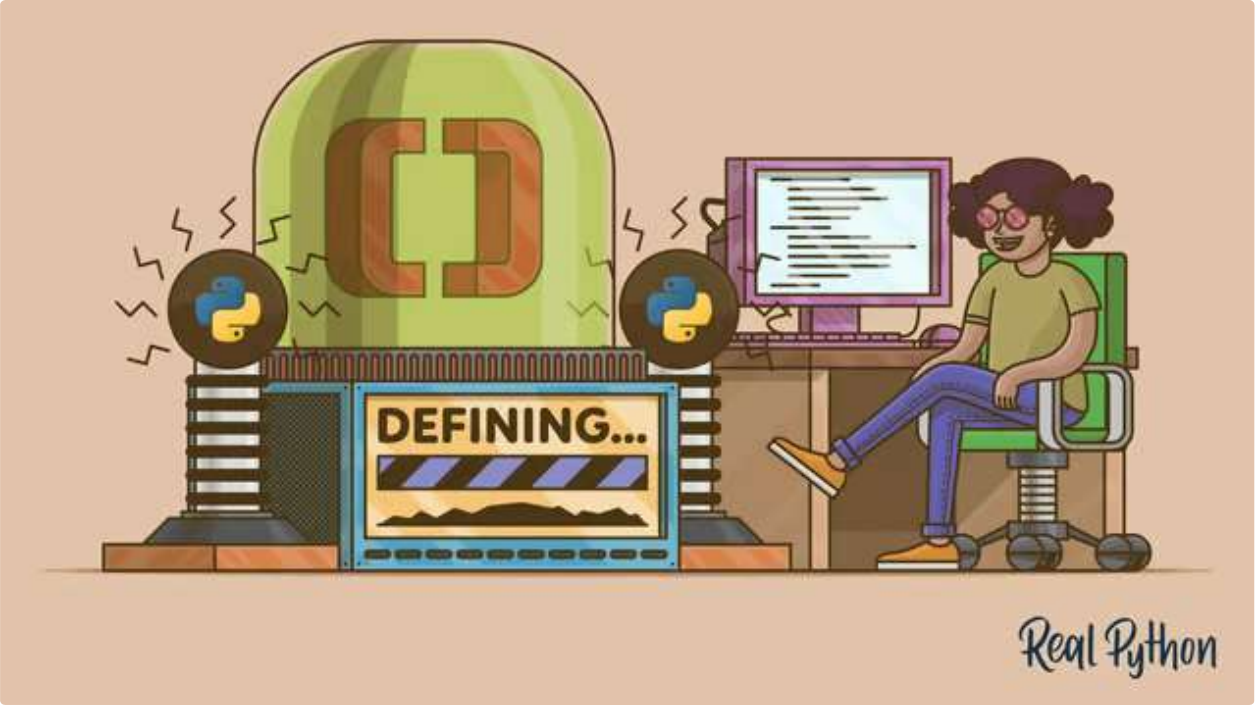


Defining Your Own Python Function

by John Sturtz Mar 09, 2020 13 Comments basics python

Mark as Completed

Share Share Email

Table of Contents

Help

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Defining and Calling Python Functions**

Throughout the previous tutorials in this series, you've seen many examples demonstrating the use of built-in Python functions. In this tutorial, you'll learn how to **define your own Python function**. You'll learn when to divide your program into separate user-defined functions and what tools you'll need to do this.

Here's what you'll learn in this tutorial:

- How **functions** work in Python and why they're beneficial
- How to **define and call** your own Python function
- Mechanisms for **passing arguments** to your function
- How to **return data** from your function back to the calling environment

Functions in Python

You may be familiar with the mathematical concept of a **function**. A function is a relationship or mapping between one or more inputs and a set of outputs. In mathematics, a function is typically represented like this:

$$z = f(x, y)$$

Here, `f` is a function that operates on the inputs `x` and `y`. The output of the function is `z`. However, programming functions are much more generalized and versatile than this mathematical definition. In fact, appropriate function definition and use is so critical to proper software development that virtually all modern programming languages support both built-in and user-defined functions.

In programming, a **function** is a self-contained block of code that encapsulates a specific task or related group of tasks. In previous tutorials in this series, you've been introduced to some of the built-in functions provided by Python. `id()`, for example, takes one argument and returns that object's unique integer identifier:

Python

```
>>> s = 'foobar'
>>> id(s)
56313440
```

`len()` returns the length of the argument passed to it:

Python

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> len(a)
4
```

any() takes an iterable as its argument and returns True if any of the items in the iterable are truthy and False otherwise:

Python

```
>>> any([False, False, False])
False
>>> any([False, True, False])
True

>>> any(['bar' == 'baz', len('foo') == 4, 'qux' in {'foo', 'bar', 'baz'}])
False
>>> any(['bar' == 'baz', len('foo') == 3, 'qux' in {'foo', 'bar', 'baz'}])
True
```

Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but you don't need to know where or even how the code works. All you need to know about is the function's interface:

1. What **arguments** (if any) it takes
2. What **values** (if any) it returns

Then you call the function and pass the appropriate arguments. Program execution goes off to the designated body of code and does its useful thing. When the function is finished, execution returns to your code where it left off. The function may or may not return data for your code to use, as the examples above do.

When you define your own Python function, it works just the same. From somewhere in your code, you'll call your Python function and program execution will transfer to the body of code that makes up the function.

> **Note:** In this case, you will know where the code is and exactly how it works because you wrote it!

When the function is finished, execution returns to the location where the function was called. Depending on how you designed the function's interface, data may be passed in when the function is called, and return values may be passed back when it finishes.

The Importance of Python Functions

Virtually all programming languages used today support a form of user-defined functions, although they aren't always called functions. In other languages, you may see them referred to as one of the following:

- **Subroutines**
- **Procedures**
- **Methods**
- **Subprograms**

So, why bother defining functions? There are several very good reasons. Let's go over a few now.

Abstraction and Reusability

Suppose you write some code that does something useful. As you continue development, you find that the task performed by that code is one you need often, in many different locations within your application. What should you do? Well, you could just replicate the code over and over again, using your editor's copy-and-paste capability.

Later on, you'll probably decide that the code in question needs to be modified. You'll either find something wrong with it that needs to be fixed, or you'll want to enhance it in some way. If copies of the code are scattered all over your application, then you'll need to make the necessary changes in every location.

> **Note:** At first blush, that may seem like a reasonable solution, but in the long term, it's likely to be a maintenance nightmare! While your code editor may help by providing a search-and-replace function, this method is error-prone, and you could easily introduce bugs into your code that will be difficult to find.

A better solution is to **define a Python function that performs the task**. Anywhere in your application that you need to accomplish the task, you simply call the function. Down the line, if you decide to change how it works, then you only need to change the code in one location, which is the place where the function is defined. The changes will automatically be picked up anywhere the function is called.

The **abstraction of functionality** into a function definition is an example of the Don't Repeat Yourself (DRY) Principle of software development. This is arguably the strongest motivation for using functions.

Modularity

Functions allow **complex processes** to be broken up into smaller steps. Imagine, for example, that you have a program that reads in a file, processes the file contents, and then writes an output file. Your code could look like this:

Python

```
# Main program

# Code to read file in
<statement>
<statement>
<statement>
<statement>

# Code to process file
<statement>
<statement>
<statement>
<statement>

# Code to write file out
<statement>
<statement>
<statement>
<statement>
```

In this example, the main program is a bunch of code strung together in a long sequence, with whitespace and comments to help organize it. However, if the code were to get much lengthier and more complex, then you'd have an increasingly difficult time wrapping your head around it.

Alternatively, you could structure the code more like the following:

Python

```
def read_file():
    # Code to read file in
    <statement>
    <statement>
    <statement>
    <statement>

def process_file():
    # Code to process file
    <statement>
    <statement>
    <statement>
    <statement>

def write_file():
    # Code to write file out
    <statement>
    <statement>
    <statement>
    <statement>


# Main program
read_file()
process_file()
write_file()
```

This example is **modularized**. Instead of all the code being strung together, it's broken out into separate functions, each of which focuses on a specific task. Those tasks are *read*, *process*, and *write*. The main program now simply needs to call each of these in turn.

> **Note:** The `def` keyword introduces a new Python function definition. You'll learn all about this very soon.

In life, you do this sort of thing all the time, even if you don't explicitly think of it that way. If you wanted to move some shelves full of stuff from one side of your garage to the other, then you hopefully wouldn't just stand there and aimlessly think, "Oh, geez. I need to move all that stuff over there! How do I do that???" You'd divide the job into manageable steps:

1. **Take** all the stuff off the shelves.
2. **Take** the shelves apart.
3. **Carry** the shelf parts across the garage to the new location.
4. **Re-assemble** the shelves.
5. **Carry** the stuff across the garage.
6. **Put** the stuff back on the shelves.

Breaking a large task into smaller, bite-sized sub-tasks helps make the large task easier to think about and manage. As programs become more complicated, it becomes increasingly beneficial to modularize them in this way.

Namespace Separation

A **namespace** is a region of a program in which **identifiers** have meaning. As you'll see below, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

The practical upshot of this is that variables can be defined and used within a Python function even if they have the same name as variables defined in other functions or in the main program. In these cases, there will be no confusion or interference because they're kept in separate namespaces.

This means that when you write code within a function, you can use variable names and identifiers without worrying about whether they're already used elsewhere outside the function. This helps minimize errors in code considerably.

> **Note:** You'll learn much more about namespaces later in this series.

Hopefully, you're sufficiently convinced of the virtues of functions and eager to create some! Let's see how.

Function Calls and Definition

The usual syntax for defining a Python function is as follows:

Python

```
def <function_name>([<parameters>]):
    <statement(s)>
```

The components of the definition are explained in the table below:

Component	Meaning
`def`	The keyword that informs Python that a function is being defined
`<function_name>`	A valid Python identifier that names the function
`<parameters>`	An optional, comma-separated list of parameters that may be passed to the function
`:`	Punctuation that denotes the end of the Python function header (the name and parameter list)
`<statement(s)>`	A block of valid Python statements

The final item, `<statement(s)>`, is called the **body** of the function. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the off-side rule. This is the same as code blocks associated with a control structure, like an `if` or `while` statement.

The syntax for calling a Python function is as follows:

Python

```
<function_name>([<arguments>])
```

`<arguments>` are the values passed into the function. They correspond to the `<parameters>` in the Python function definition. You can define a function that doesn't take any arguments, but the parentheses are still required. Both a function definition and a function call must always include parentheses, even if they're empty.

As usual, you'll start with a small example and add complexity from there. Keeping the time-honored mathematical tradition in mind, you'll call your first Python function `f()`. Here's a script file, `foo.py`, that defines and calls `f()`:

Python

```
1 def f():
2     s = '-- Inside f()'
3     print(s)
4
5 print('Before calling f()')
6 f()
7 print('After calling f()')
```

Here's how this code works:

1. **Line 1** uses the `def` keyword to indicate that a function is being defined. Execution of the `def` statement merely creates the definition of `f()`. All the following lines that are indented (lines 2 to 3) become part of the body of `f()` and are stored as its definition, but they aren't executed yet.

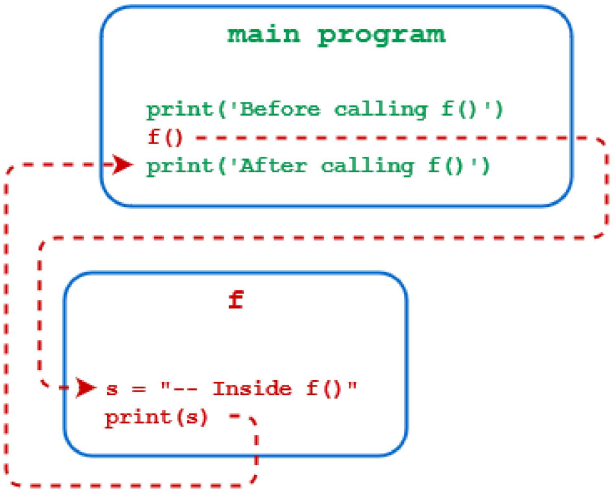
2. **Line 4** is a bit of whitespace between the function definition and the first line of the main program. While it isn't syntactically necessary, it is nice to have. To learn more about whitespace around top-level Python function definitions, check out Writing Beautiful Pythonic Code With PEP 8.

3. **Line 5** is the first statement that isn't indented because it isn't a part of the definition of `f()`. It's the start of the main program. When the main program executes, this statement is executed first.

4. **Line 6** is a call to `f()`. Note that empty parentheses are always required in both a function definition and a function call, even when there are no parameters or arguments. Execution proceeds to `f()` and the statements in the body of `f()` are executed.

5. **Line 7** is the next line to execute once the body of `f()` has finished. Execution returns to this `print()` statement.

The sequence of execution (or **control flow**) for `foo.py` is shown in the following diagram:



When `foo.py` is run from a Windows command prompt, the result is as follows:

Windows Command Prompt

```
C:\Users\john\Documents\Python\doc>python foo.py
Before calling f()
-- Inside f()
After calling f()
```

Occasionally, you may want to define an empty function that does nothing. This is referred to as a **stub**, which is usually a temporary placeholder for a Python function that will be fully implemented at a later time. Just as a block in a control structure can't be empty, neither can the body of a function. To define a stub function, use the `pass`statement:

Python

```
>>> def f():
...     pass
...
>>> f()
```

As you can see above, a call to a stub function is syntactically valid but doesn't do anything.

Argument Passing

So far in this tutorial, the functions you've defined haven't taken any arguments. That can sometimes be useful, and you'll occasionally write such functions. More often, though, you'll want to **pass data into a function** so that its behavior can vary from one invocation to the next. Let's see how to do that.

Positional Arguments

The most straightforward way to pass arguments to a Python function is with **positional arguments** (also called **required arguments**). In the function definition, you specify a comma-separated list of parameters inside the parentheses:

Python

```
>>> def f(qty, item, price):
...     print(f'{qty} {item} cost ${price:.2f}')
...
```

When the function is called, you specify a corresponding list of arguments:

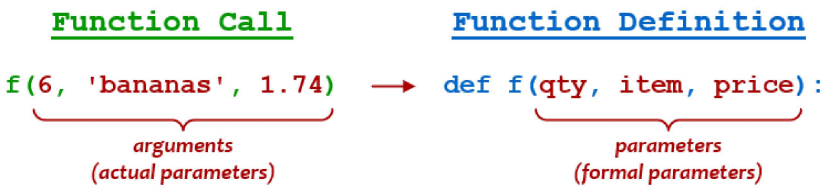
Python

```
>>> f(6, 'bananas', 1.74)
6 bananas cost $1.74
```

The parameters (`qty`, `item`, and `price`) behave like **variables** that are defined locally to the function. When the function is called, the arguments that are passed (`6`, `'bananas'`, and `1.74`) are **bound** to the parameters in order, as though by variable assignment:

Parameter		Argument
`qty`	←	`6`
`item`	←	`bananas`
`price`	←	`1.74`

In some programming texts, the parameters given in the function definition are referred to as **formal parameters**, and the arguments in the function call are referred to as **actual parameters:**



Although positional arguments are the most straightforward way to pass data to a function, they also afford the least flexibility. For starters, the **order** of the arguments in the call must match the order of the parameters in the definition. There's nothing to stop you from specifying positional arguments out of order, of course:

Python

```
>>> f('bananas', 1.74, 6)
bananas 1.74 cost $6.00
```

The function may even still run, as it did in the example above, but it's very unlikely to produce the correct results. It's the responsibility of the programmer who defines the function to document what the **appropriate arguments** should be, and it's the responsibility of the user of the function to be aware of that information and abide by it.

With positional arguments, the arguments in the call and the parameters in the definition must agree not only in order but in **number** as well. That's the reason positional arguments are also referred to as required arguments. You can't leave any out when calling the function:

Python

```
>>> # Too few arguments
>>> f(6, 'bananas')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    f(6, 'bananas')
TypeError: f() missing 1 required positional argument: 'price'
```

Nor can you specify extra ones:

Python

```
>>> # Too many arguments
>>> f(6, 'bananas', 1.74, 'kumquats')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    f(6, 'bananas', 1.74, 'kumquats')
TypeError: f() takes 3 positional arguments but 4 were given
```

Positional arguments are conceptually straightforward to use, but they're not very forgiving. You must specify the same number of arguments in the function call as there are parameters in the definition, and in exactly the same order. In the sections that follow, you'll see some argument-passing techniques that relax these restrictions.

Keyword Arguments

When you're calling a function, you can specify arguments in the form `<keyword>=<value>`. In that case, each `<keyword>` must match a parameter in the Python function definition. For example, the previously defined function `f()` may be called with **keyword arguments** as follows:

Python

```
>>> f(qty=6, item='bananas', price=1.74)
6 bananas cost $1.74
```

Referencing a keyword that doesn't match any of the declared parameters generates an exception:

Python

```
>>> f(qty=6, item='bananas', cost=1.74)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got an unexpected keyword argument 'cost'
```

Using keyword arguments lifts the restriction on argument order. Each keyword argument explicitly designates a specific parameter by name, so you can specify them in any order and Python will still know which argument goes with which parameter:

Python

```
>>> f(item='bananas', price=1.74, qty=6)
6 bananas cost $1.74
```

Like with positional arguments, though, the number of arguments and parameters must still match:

Python

```
>>> # Still too few arguments
>>> f(qty=6, item='bananas')
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    f(qty=6, item='bananas')
TypeError: f() missing 1 required positional argument: 'price'
```

So, keyword arguments allow flexibility in the order that function arguments are specified, but the number of arguments is still rigid.

You can call a function using both positional and keyword arguments:

Python

```
>>> f(6, price=1.74, item='bananas')
6 bananas cost $1.74

>>> f(6, 'bananas', price=1.74)
6 bananas cost $1.74
```

When positional and keyword arguments are both present, all the positional arguments must come first:

Python

```
>>> f(6, item='bananas', 1.74)
SyntaxError: positional argument follows keyword argument
```

Once you've specified a keyword argument, there can't be any positional arguments to the right of it.

To learn more about positional and keyword parameters, check out the Real Python course Exploring Special Function Parameters.

Default Parameters

If a parameter specified in a Python function definition has the form `<name>=<value>`, then `<value>` becomes a default value for that parameter. Parameters defined this way are referred to as **default or optional parameters**. An example of a function definition with default parameters is shown below:

Python

```
>>> def f(qty=6, item='bananas', price=1.74):
...     print(f'{qty} {item} cost ${price:.2f}')
...
```

When this version of `f()` is called, any argument that's left out assumes its default value:

Python

```
>>> f(4, 'apples', 2.24)
4 apples cost $2.24
>>> f(4, 'apples')
4 apples cost $1.74

>>> f(4)
4 bananas cost $1.74
>>> f()
6 bananas cost $1.74

>>> f(item='kumquats', qty=9)
9 kumquats cost $1.74
>>> f(price=2.29)
6 bananas cost $2.29
```

In summary:

- **Positional arguments** must agree in order and number with the parameters declared in the function definition.
- **Keyword arguments** must agree with declared parameters in number, but they may be specified in arbitrary order.
- **Default parameters** allow some arguments to be omitted when the function is called.

Mutable Default Parameter Values

Things can get weird if you specify a default parameter value that is a mutable object. Consider this Python function definition:

Python

```
>>> def f(my_list=[]):
...     my_list.append('###')
...     return my_list
...
```

`f()` takes a single list parameter, appends the string `'###'` to the end of the list, and returns the result:

Python

```
>>> f(['foo', 'bar', 'baz'])
['foo', 'bar', 'baz', '###']

>>> f([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5, '###']
```

The default value for parameter `my_list` is the empty list, so if `f()` is called without any arguments, then the return value is a list with the single element `'###'`:

Python

```
>>> f()
['###']
```

Everything makes sense so far. Now, what would you expect to happen if `f()` is called without any parameters a second and a third time? Let's see:

Python

```
>>> f()
['###', '###']
>>> f()
['###', '###', '###']
```

Oops! You might have expected each subsequent call to also return the singleton list `['###']`, just like the first. Instead, the return value keeps growing. What happened?

In Python, default parameter values are **defined only once** when the function is defined (that is, when the `def` statement is executed). The default value isn't re-defined each time the function is called. Thus, each time you call `f()` without a parameter, you're performing `.append()` on the same list.

You can demonstrate this with `id()`:

Python

```
>>> def f(my_list=[]):
...     print(id(my_list))
...     my_list.append('###')
...     return my_list
...
>>> f()
140095566958408
['###']
>>> f()
140095566958408
['###', '###']
>>> f()
140095566958408
['###', '###', '###']
```

The **object identifier** displayed confirms that, when `my_list` is allowed to default, the value is the same object with each call. Since lists are mutable, each subsequent `.append()` call causes the list to get longer. This is a common and pretty well-documented pitfall when you're using a mutable object as a parameter's default value. It potentially leads to confusing code behavior, and is probably best avoided.

As a workaround, consider using a default argument value that signals **no argument has been specified**. Most any value would work, but `None` is a common choice. When the sentinel value indicates no argument is given, create a new empty list inside the function:

Python

```
>>> def f(my_list=None):
...     if my_list is None:
...         my_list = []
...     my_list.append('###')
...     return my_list
...

>>> f()
['###']
>>> f()
['###']
>>> f()
['###']

>>> f(['foo', 'bar', 'baz'])
['foo', 'bar', 'baz', '###']

>>> f([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5, '###']
```

Note how this ensures that `my_list` now truly defaults to an empty list whenever `f()` is called without an argument.

Pass-By-Value vs Pass-By-Reference in Pascal

In programming language design, there are two common paradigms for passing an argument to a function:

1. **Pass-by-value:** A copy of the argument is passed to the function.
2. **Pass-by-reference:** A reference to the argument is passed to the function.

Other mechanisms exist, but they are essentially variations on these two. In this section, you're going to take a short detour from Python and briefly look at Pascal, a programming language that makes a particularly clear distinction between these two.

> **Note:** Don't worry if you aren't familiar with Pascal! The concepts are similar to those of Python, and the examples shown are accompanied by enough detailed explanation that you should get a general idea. Once you've seen how argument passing works in Pascal, we'll circle back around to Python, and you'll see how it compares.

Here's what you need to know about Pascal syntax:

- **Procedures:** A procedure in Pascal is similar to a Python function.
- **Colon-equals:** This operator (`:=`) is used for assignment in Pascal. It's analogous to the equals sign (`=`) in Python.
- **`writeln()`:** This function displays data to the console, similar to Python's `print()`.

With that bit of groundwork in place, here's the first Pascal example:

Pascal

```pascal
// Pascal Example #1

procedure f(fx : integer);
begin
    writeln('Start  f():  fx = ', fx);
    fx := 10;
    writeln('End    f():  fx = ', fx);
end;

// Main program
var
    x : integer;

begin
    x := 5;
    writeln('Before f():  x = ', x);
    f(x);
    writeln('After  f():  x = ', x);
end.
```

Here's what's going on:

- **Line 12:** The main program defines an integer variable `x`.
- **Line 15:** It initially assigns `x` the value `5`.
- **Line 17:** It then calls procedure `f()`, passing `x` as an argument.
- **Line 5:** Inside `f()`, the `writeln()` statement shows that the corresponding parameter `fx` is initially `5`, the value passed in.
- **Line 6:** `fx` is then assigned the value `10`.
- **Line 7:** This value is verified by this `writeln()` statement executed just before `f()` exits.
- **Line 18:** Back in the calling environment of the main program, this `writeln()` statement shows that after `f()` returns, `x` is still `5`, as it was prior to the procedure call.

Running this code generates the following output:

Pascal

```
Before f():  x = 5
Start  f():  fx = 5
End    f():  fx = 10
After  f():  x = 5
```

In this example, `x` is **passed by value**, so `f()` receives only a copy. When the corresponding parameter `fx` is modified, `x` is unaffected.

> **Note:** If you want to see this in action, then you can run the code for yourself using an online Pascal compiler.
>
> Just follow these steps:
>
> 1. Copy the code from the code box above.
> 2. Visit the Online Pascal Compiler.
> 3. In the code box on the left, replace any existing contents with the code you copied in step 1.
> 4. Click *Execute*.
>
> You should see the same output as above.

Now, compare this with the next example:

Pascal

```pascal
// Pascal Example #2

procedure f(var fx : integer);
begin
    writeln('Start  f():  fx = ', fx);
    fx := 10;
    writeln('End    f():  fx = ', fx);
end;

// Main program
var
    x : integer;

begin
    x := 5;
    writeln('Before f():  x = ', x);
    f(x);
    writeln('After  f():  x = ', x);
end.
```

This code is identical to the first example, with one change. It's the presence of the word `var` in front of `fx` in the definition of procedure `f()` on line 3. That indicates that the argument to `f()` is **passed by reference**. Changes made to the corresponding parameter `fx` will also modify the argument in the calling environment.

The output from this code is the same as before, except for the last line:

Pascal

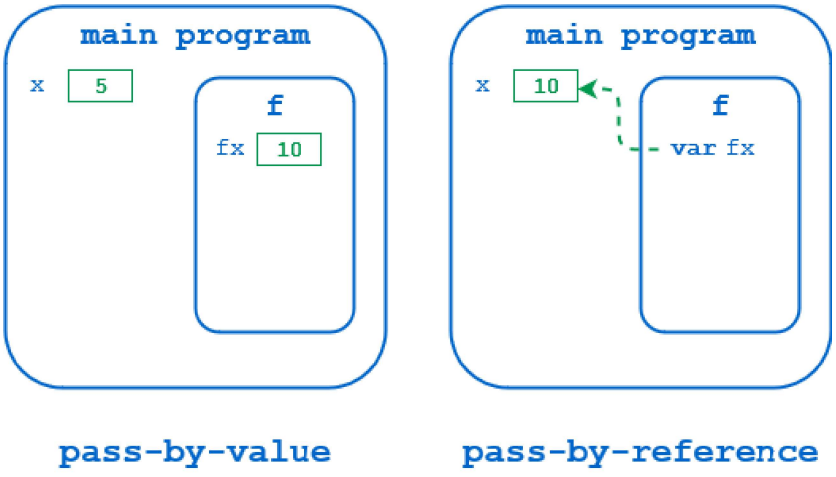
```
Before f():  x = 5
Start  f():  fx = 5
End    f():  fx = 10
After  f():  x = 10
```

Again, `fx` is assigned the value `10` inside `f()` as before. But this time, when `f()` returns, `x` in the main program has also been modified.

In many programming languages, that's essentially the distinction between pass-by-value and pass-by-reference:

- **If a variable is passed by value,** then the function has a copy to work on, but it can't modify the original value in the calling environment.
- **If a variable is passed by reference,** then any changes the function makes to the corresponding parameter will affect the value in the calling environment.

The reason why comes from what a **reference** means in these languages. Variable values are stored in memory. In Pascal and similar languages, a reference is essentially the address of that memory location, as demonstrated below:



In the diagram on the left, `x` has memory allocated in the main program's namespace. When `f()` is called, `x` is **passed by value**, so memory for the corresponding parameter `fx` is allocated in the namespace of `f()`, and the value of `x` is copied there. When `f()` modifies `fx`, it's this local copy that is changed. The value of `x` in the calling environment remains unaffected.

In the diagram on the right, `x` is **passed by reference**. The corresponding parameter `fx` points to the actual address in the main program's namespace where the value of `x` is stored. When `f()` modifies `fx`, it's modifying the value in that location, just the same as if the main program were modifying `x` itself.

Pass-By-Value vs Pass-By-Reference in Python

Are parameters in Python pass-by-value or pass-by-reference? The answer is they're neither, exactly. That's because a reference doesn't mean quite the same thing in Python as it does in Pascal.

Recall that in Python, every piece of data is an **object**. A reference points to an object, not a specific memory location. That means assignment isn't interpreted the same way in Python as it is in Pascal. Consider the following pair of statements in Pascal:

Pascal

```
x := 5
x := 10
```

These are interpreted this way:

- **The variable** `x` references a specific memory location.
- **The first statement** puts the value `5` in that location.
- **The next statement** overwrites the `5` and puts `10` there instead.

By contrast, in Python, the analogous assignment statements are as follows:

Python

```
x = 5
x = 10
```

These assignment statements have the following meaning:

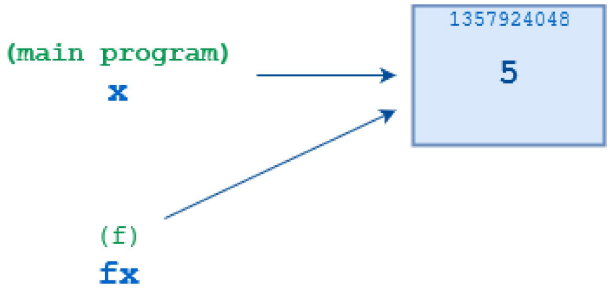
- **The first statement** causes `x` to point to an object whose value is `5`.
- **The next statement** reassigns `x` as a new reference to a different object whose value is `10`. Stated another way, the second assignment rebinds `x` to a different object with value `10`.

In Python, when you pass an argument to a function, a similar **rebinding** occurs. Consider this example:

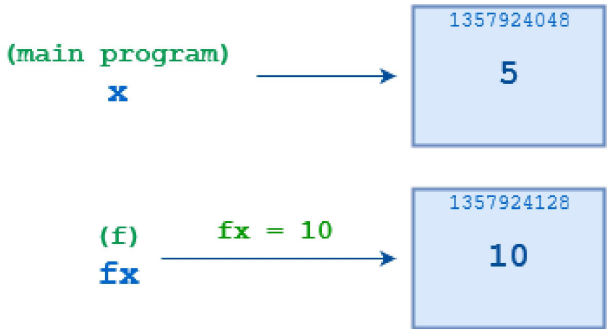
Python

```
1 >>> def f(fx):
2 ...     fx = 10
3 ...
4 >>> x = 5
5 >>> f(x)
6 >>> x
7 5
```

In the main program, the statement `x = 5` on line 4 creates a reference named `x` bound to an object whose value is `5`. `f()` is then called on line 5, with `x` as an argument. When `f()` first starts, a new reference called `fx` is created, which initially points to the same `5` object as `x` does:



However, when the statement `fx = 10` on line 2 is executed, `f()` **rebinds** `fx` to a new object whose value is `10`. The two references, `x` and `fx`, are **uncoupled** from one another. Nothing else that `f()` does will affect `x`, and when `f()` terminates, `x` will still point to the object `5`, as it did prior to the function call:



You can confirm all this using `id()`. Here's a slightly augmented version of the above example that displays the numeric identifiers of the objects involved:

Python

```
>>> def f(fx):
...     print('fx =', fx, '/ id(fx) = ', id(fx))
...     fx = 10
...     print('fx =', fx, '/ id(fx) = ', id(fx))
...

>>> x = 5
>>> print('x =', x, '/ id(x) = ', id(x))
x = 5 / id(x) =  1357924048

>>> f(x)
fx = 5 / id(fx) =  1357924048
fx = 10 / id(fx) =  1357924128

>>> print('x =', x, '/ id(x) = ', id(x))
x = 5 / id(x) =  1357924048
```

When `f()` first starts, `fx` and `x` both point to the same object, whose `id()` is `1357924048`. After `f()` executes the statement `fx = 10` on line 3, `fx` points to a different object whose `id()` is `1357924128`. The connection to the original object in the calling environment is lost.

Argument passing in Python is somewhat of a hybrid between pass-by-value and pass-by-reference. What gets passed to the function is a reference to an object, but the reference is passed by value.

> **Note:** Python's argument-passing mechanism has been called **pass-by-assignment**. This is because parameter names are bound to objects on function entry in Python, and assignment is also the process of binding a name to an object. You may also see the terms pass-by-object, pass-by-object-reference, or pass-by-sharing.

The key takeaway here is that a Python function can't change the value of an argument by reassigning the corresponding parameter to something else. The following example demonstrates this:

Python

```
>>> def f(x):
...     x = 'foo'
...
>>> for i in (
...         40,
...         dict(foo=1, bar=2),
...         {1, 2, 3},
...         'bar',
...         ['foo', 'bar', 'baz']):
...     f(i)
...     print(i)
...
40
{'foo': 1, 'bar': 2}
{1, 2, 3}
bar
['foo', 'bar', 'baz']
```

Here, objects of type `int`, `dict`, `set`, `str`, and `list` are passed to `f()` as arguments. `f()` tries to assign each to the string object `'foo'`, but as you can see, once back in the calling environment, they are all unchanged. As soon as `f()` executes the assignment `x = 'foo'`, the reference is **rebound**, and the connection to the original object is lost.

Does that mean a Python function can never modify its arguments at all? Actually, no, that isn't the case! Watch what happens here:

Python

```
>>> def f(x):
...     x[0] = '---'
...

>>> my_list = ['foo', 'bar', 'baz', 'qux']

>>> f(my_list)
>>> my_list
['---', 'bar', 'baz', 'qux']
```

In this case, the argument to `f()` is a list. When `f()` is called, a reference to `my_list` is passed. You've already seen that `f()` can't reassign `my_list` wholesale. If `x` were assigned to something else, then it would be bound to a different object, and the connection to `my_list` would be lost.

However, `f()` can use the reference to make modifications inside `my_list`. Here, `f()` has modified the first element. You can see that once the function returns, `my_list` has, in fact, been changed in the calling environment. The same concept applies to a dictionary:

Python

```
>>> def f(x):
...     x['bar'] = 22
...

>>> my_dict = {'foo': 1, 'bar': 2, 'baz': 3}

>>> f(my_dict)
>>> my_dict
{'foo': 1, 'bar': 22, 'baz': 3}
```

Here, `f()` uses `x` as a reference to make a change inside `my_dict`. That change is reflected in the calling environment after `f()` returns.

Argument Passing Summary

Argument passing in Python can be summarized as follows. **Passing an immutable object**, like an `int`, `str`, `tuple`, or `frozenset`, to a Python function acts like pass-by-value. The function can't modify the object in the calling environment.

Passing a mutable object such as a `list`, `dict`, or `set` acts somewhat—but not exactly—like pass-by-reference. The function can't reassign the object wholesale, but it can change items in place within the object, and these changes will be reflected in the calling environment.

Side Effects

So, in Python, it's possible for you to modify an argument from within a function so that the change is reflected in the calling environment. But should you do this? This is an example of what's referred to in programming lingo as a **side effect**.

More generally, a Python function is said to cause a side effect if it modifies its calling environment in any way. Changing the value of a function argument is just one of the possibilities.

> **Note:** You're probably familiar with side effects from the field of human health, where the term typically refers to an **unintended consequence** of medication. Often, the consequence is undesirable, like vomiting or sedation. On the other hand, side effects *can* be used intentionally. For example, some medications cause appetite stimulation, which can be used to an advantage, even if that isn't the medication's primary intent.
>
> The concept is similar in programming. If a side effect is a well-documented part of the function specification, and the user of the function is expressly aware of when and how the calling environment might be modified, then it can be okay. But a programmer may not always properly document side effects, or they may not even be aware that side effects are occurring.

When they're hidden or unexpected, side effects can lead to program errors that are very difficult to track down. Generally, it's best to avoid them.

The `return` Statement

What's a Python function to do then? After all, in many cases, if a function doesn't cause some change in the calling environment, then there isn't much point in calling it at all. How should a function affect its caller?

Well, one possibility is to use **function return values**. A `return` statement in a Python function serves two purposes:

1. It immediately terminates the function and passes execution control back to the caller.
2. It provides a mechanism by which the function can pass data back to the caller.

Exiting a Function

Within a function, a `return` statement causes immediate exit from the Python function and transfer of execution back to the caller:

Python

```
>>> def f():
...     print('foo')
...     print('bar')
...     return
...

>>> f()
foo
bar
```

In this example, the `return` statement is actually superfluous. A function will return to the caller when it **falls off the end**—that is, after the last statement of the function body is executed. So, this function would behave identically without the `return` statement.

However, `return` statements don't need to be at the end of a function. They can appear anywhere in a function body, and even multiple times. Consider this example:

Python

```
>>> def f(x):
...     if x < 0:
...         return
...     if x > 100:
...         return
...     print(x)
...

>>> f(-3)
>>> f(105)
>>> f(64)
64
```

The first two calls to `f()` don't cause any output, because a `return` statement is executed and the function exits prematurely, before the `print()` statement on line 6 is reached.

This sort of paradigm can be useful for **error checking** in a function. You can check several error conditions at the start of the function, with `return` statements that bail out if there's a problem:

Python

```
def f():
    if error_cond1:
        return
    if error_cond2:
        return
    if error_cond3:
        return

    <normal processing>
```

If none of the error conditions are encountered, then the function can proceed with its normal processing.

Returning Data to the Caller

In addition to exiting a function, the `return` statement is also used to **pass data back to the caller**. If a `return` statement inside a Python function is followed by an expression, then in the calling environment, the function call evaluates to the value of that expression:

Python

```
>>> def f():
...     return 'foo'
...

>>> s = f()
>>> s
'foo'
```

Here, the value of the expression `f()` on line 5 is `'foo'`, which is subsequently assigned to variable `s`.

A function can return any type of **object**. In Python, that means pretty much anything whatsoever. In the calling environment, the function call can be used syntactically in any way that makes sense for the type of object the function returns.

For example, in this code, `f()` returns a dictionary. In the calling environment then, the expression `f()` represents a dictionary, and `f()['baz']` is a valid key reference into that dictionary:

Python

```
>>> def f():
...     return dict(foo=1, bar=2, baz=3)
...

>>> f()
{'foo': 1, 'bar': 2, 'baz': 3}
>>> f()['baz']
3
```

In the next example, `f()` returns a string that you can slice like any other string:

Python

```
>>> def f():
...     return 'foobar'
...

>>> f()[2:4]
'ob'
```

Here, `f()` returns a list that can be indexed or sliced:

Python

```
>>> def f():
...     return ['foo', 'bar', 'baz', 'qux']
...

>>> f()
['foo', 'bar', 'baz', 'qux']
>>> f()[2]
'baz'
>>> f()[::-1]
['qux', 'baz', 'bar', 'foo']
```

If multiple comma-separated expressions are specified in a `return` statement, then they're packed and returned as a tuple:

Python

```
>>> def f():
...     return 'foo', 'bar', 'baz', 'qux'
...

>>> type(f())
<class 'tuple'>
>>> t = f()
>>> t
('foo', 'bar', 'baz', 'qux')

>>> a, b, c, d = f()
>>> print(f'a = {a}, b = {b}, c = {c}, d = {d}')
a = foo, b = bar, c = baz, d = qux
```

When no return value is given, a Python function returns the special Python value `None`:

Python

```
>>> def f():
...     return
...

>>> print(f())
None
```

The same thing happens if the function body doesn't contain a `return` statement at all and the function falls off the end:

Python

```
>>> def g():
...     pass
...

>>> print(g())
None
```

Recall that `None` is falsy when evaluated in a Boolean context.

Since functions that exit through a bare `return` statement or fall off the end return `None`, a call to such a function can be used in a Boolean context:

Python

```
>>> def f():
...     return
...
>>> def g():
...     pass
...

>>> if f() or g():
...     print('yes')
... else:
...     print('no')
...
no
```

Here, calls to both `f()` and `g()` are falsy, so `f() or g()` is as well, and the `else` clause executes.

Revisiting Side Effects

Suppose you want to write a function that takes an integer argument and doubles it. That is, you want to pass an integer variable to the function, and when the function returns, the value of the variable in the calling environment should be twice what it was. In Pascal, you could accomplish this using pass-by-reference:

Pascal

```
1  procedure double(var x : integer);
2  begin
3      x := x * 2;
4  end;
5
6  var
7      x : integer;
8
9  begin
10     x := 5;
11     writeln('Before procedure call: ', x);
12     double(x);
13     writeln('After procedure call:  ', x);
14 end.
```

Executing this code produces the following output, which verifies that `double()` does indeed modify `x` in the calling environment:

Pascal

```
Before procedure call: 5
After procedure call:  10
```

In Python, this won't work. As you now know, Python integers are immutable, so a Python function can't change an integer argument by side effect:

Python

```
>>> def double(x):
...     x *= 2
...

>>> x = 5
>>> double(x)
>>> x
5
```

However, you can use a return value to obtain a similar effect. Simply write `double()` so that it takes an integer argument, doubles it, and returns the doubled value. Then, the caller is responsible for the assignment that modifies the original value:

Python

```
>>> def double(x):
...     return x * 2
...

>>> x = 5
>>> x = double(x)
>>> x
10
```

This is arguably preferable to modifying by side effect. It's very clear that `x` is being modified in the calling environment because the caller is doing so itself. Anyway, it's the only option, because modification by side effect doesn't work in this case.

Still, even in cases where it's possible to modify an argument by side effect, using a return value may still be clearer. Suppose you want to double every item in a list. Because lists are mutable, you could define a Python function that modifies the list in place:

Python

```
>>> def double_list(x):
...     i = 0
...     while i < len(x):
...             x[i] *= 2
...             i += 1
...

>>> a = [1, 2, 3, 4, 5]
>>> double_list(a)
>>> a
[2, 4, 6, 8, 10]
```

Unlike `double()` in the previous example, `double_list()` actually works as intended. If the documentation for the function clearly states that the list argument's contents are changed, then this may be a reasonable implementation.

However, you can also write `double_list()` to pass the desired list back by return value and allow the caller to make the assignment, similar to how `double()` was re-written in the previous example:

Python

```
>>> def double_list(x):
...     r = []
...     for i in x:
...             r.append(i * 2)
...     return r
...

>>> a = [1, 2, 3, 4, 5]
>>> a = double_list(a)
>>> a
[2, 4, 6, 8, 10]
```

Either approach works equally well. As is often the case, this is a matter of style, and personal preferences vary. Side effects aren't necessarily consummate evil, and they have their place, but because virtually anything can be returned from a function, the same thing can usually be accomplished through return values as well.

Variable-Length Argument Lists

In some cases, when you're defining a function, you may not know beforehand how many arguments you'll want it to take. Suppose, for example, that you want to write a Python function that computes the average of several values. You could start with something like this:

Python

```
>>> def avg(a, b, c):
...     return (a + b + c) / 3
...
```

All is well if you want to average three values:

Python

```
>>> avg(1, 2, 3)
2.0
```

However, as you've already seen, when positional arguments are used, the number of arguments passed must agree with the number of parameters declared. Clearly then, all isn't well with this implementation of `avg()` for any number of values other than three:

Python

```
>>> avg(1, 2, 3, 4)
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    avg(1, 2, 3, 4)
TypeError: avg() takes 3 positional arguments but 4 were given
```

You could try to define `avg()` with optional parameters:

Python

```
>>> def avg(a, b=0, c=0, d=0, e=0):
...     .
...     .
...     .
...
```

This allows for a variable number of arguments to be specified. The following calls are at least syntactically correct:

Python

```
avg(1)
avg(1, 2)
avg(1, 2, 3)
avg(1, 2, 3, 4)
avg(1, 2, 3, 4, 5)
```

But this approach still suffers from a couple of problems. For starters, it still only allows up to five arguments, not an arbitrary number. Worse yet, there's no way to distinguish between the arguments that were specified and those that were allowed to default. The function has no way to know how many arguments were actually passed, so it doesn't know what to divide by:

Python

```
>>> def avg(a, b=0, c=0, d=0, e=0):
...     return (a + b + c + d + e) / # Divided by what???
...
```

Evidently, this won't do either.

You could write `avg()` to take a single list argument:

Python

```
>>> def avg(a):
...     total = 0
...     for v in a:
...             total += v
...     return total / len(a)
...

>>> avg([1, 2, 3])
2.0
>>> avg([1, 2, 3, 4, 5])
3.0
```

At least this works. It allows an arbitrary number of values and produces a correct result. As an added bonus, it works when the argument is a tuple as well:

Python

```
>>> t = (1, 2, 3, 4, 5)
>>> avg(t)
3.0
```

The drawback is that the added step of having to group the values into a list or tuple is probably not something the user of the function would expect, and it isn't very elegant. Whenever you find Python code that looks inelegant, there's probably a better option.

In this case, indeed there is! Python provides a way to pass a function a variable number of arguments with argument tuple packing and unpacking using the asterisk (*) operator.

Argument Tuple Packing

When a parameter name in a Python function definition is preceded by an asterisk (*), it indicates **argument tuple packing**. Any corresponding arguments in the function call are packed into a tuple that the function can refer to by the given parameter name. Here's an example:

Python

```
>>> def f(*args):
...     print(args)
...     print(type(args), len(args))
...     for x in args:
...             print(x)
...

>>> f(1, 2, 3)
(1, 2, 3)
<class 'tuple'> 3
1
2
3

>>> f('foo', 'bar', 'baz', 'qux', 'quux')
('foo', 'bar', 'baz', 'qux', 'quux')
<class 'tuple'> 5
foo
bar
baz
qux
quux
```

In the definition of `f()`, the parameter specification `*args` indicates tuple packing. In each call to `f()`, the arguments are packed into a tuple that the function can refer to by the name `args`. Any name can be used, but `args` is so commonly chosen that it's practically a standard.

Using tuple packing, you can clean up `avg()` like this:

Python

```
>>> def avg(*args):
...     total = 0
...     for i in args:
...         total += i
...     return total / len(args)
...

>>> avg(1, 2, 3)
2.0
>>> avg(1, 2, 3, 4, 5)
3.0
```

Better still, you can tidy it up even further by replacing the `for` loop with the built-in Python function `sum()`, which sums the numeric values in any iterable:

Python

```
>>> def avg(*args):
...     return sum(args) / len(args)
...

>>> avg(1, 2, 3)
2.0
>>> avg(1, 2, 3, 4, 5)
3.0
```

Now, `avg()` is concisely written and works as intended.

Still, depending on how this code will be used, there may still be work to do. As written, `avg()` will produce a `TypeError` exception if any arguments are non-numeric:

Python

```
>>> avg(1, 'foo', 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in avg
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

To be as robust as possible, you should add code to check that the arguments are of the proper type. Later in this tutorial series, you'll learn how to catch exceptions like `TypeError` and handle them appropriately. You can also check out Python Exceptions: An Introduction.

Argument Tuple Unpacking

An analogous operation is available on the other side of the equation in a Python function call. When an argument in a function call is preceded by an asterisk (`*`), it indicates that the argument is a tuple that should be **unpacked** and passed to the function as separate values:

Python

```
>>> def f(x, y, z):
...     print(f'x = {x}')
...     print(f'y = {y}')
...     print(f'z = {z}')
...

>>> f(1, 2, 3)
x = 1
y = 2
z = 3

>>> t = ('foo', 'bar', 'baz')
>>> f(*t)
x = foo
y = bar
z = baz
```

In this example, `*t` in the function call indicates that `t` is a tuple that should be unpacked. The unpacked values `'foo'`, `'bar'`, and `'baz'` are assigned to the parameters `x`, `y`, and `z`, respectively.

Although this type of unpacking is called **tuple** unpacking, it doesn't only work with tuples. The asterisk (`*`) operator can be applied to any iterable in a Python function call. For example, a list or set can be unpacked as well:

Python

```
>>> a = ['foo', 'bar', 'baz']
>>> type(a)
<class 'list'>
>>> f(*a)
x = foo
y = bar
z = baz

>>> s = {1, 2, 3}
>>> type(s)
<class 'set'>
>>> f(*s)
x = 1
y = 2
z = 3
```

You can even use tuple packing and unpacking at the same time:

Python

```
>>> def f(*args):
...     print(type(args), args)
...

>>> a = ['foo', 'bar', 'baz', 'qux']
>>> f(*a)
<class 'tuple'> ('foo', 'bar', 'baz', 'qux')
```

Here, `f(*a)` indicates that list `a` should be unpacked and the items passed to `f()` as individual values. The parameter specification `*args` causes the values to be packed back up into the tuple `args`.

Argument Dictionary Packing

Python has a similar operator, the double asterisk (`**`), which can be used with Python function parameters and arguments to specify **dictionary packing and unpacking**. Preceding a parameter in a Python function definition by a double asterisk (`**`) indicates that the corresponding arguments, which are expected to be `key=value` pairs, should be packed into a dictionary:

Python

```
>>> def f(**kwargs):
...     print(kwargs)
...     print(type(kwargs))
...     for key, val in kwargs.items():
...             print(key, '->', val)
...

>>> f(foo=1, bar=2, baz=3)
{'foo': 1, 'bar': 2, 'baz': 3}
<class 'dict'>
foo -> 1
bar -> 2
baz -> 3
```

In this case, the arguments `foo=1`, `bar=2`, and `baz=3` are packed into a dictionary that the function can reference by the name `kwargs`. Again, any name can be used, but the peculiar `kwargs` (which is short for **keyword args**) is nearly standard. You don't have to adhere to it, but if you do, then anyone familiar with Python coding conventions will know straightaway what you mean.

Argument Dictionary Unpacking

Argument dictionary unpacking is analogous to argument tuple unpacking. When the double asterisk (`**`) precedes an argument in a Python function call, it specifies that the argument is a dictionary that should be unpacked, with the resulting items passed to the function as keyword arguments:

Python

```
>>> def f(a, b, c):
...     print(F'a = {a}')
...     print(F'b = {b}')
...     print(F'c = {c}')
...

>>> d = {'a': 'foo', 'b': 25, 'c': 'qux'}
>>> f(**d)
a = foo
b = 25
c = qux
```

The items in the dictionary `d` are unpacked and passed to `f()` as keyword arguments. So, `f(**d)` is equivalent to `f(a='foo', b=25, c='qux')`:

Python

```
>>> f(a='foo', b=25, c='qux')
a = foo
b = 25
c = qux
```

In fact, check this out:

Python

```
>>> f(**dict(a='foo', b=25, c='qux'))
a = foo
b = 25
c = qux
```

Here, `dict(a='foo', b=25, c='qux')` creates a dictionary from the specified key/value pairs. Then, the double asterisk operator (`**`) unpacks it and passes the keywords to `f()`.

Putting It All Together

Think of `*args` as a variable-length positional argument list, and `**kwargs` as a variable-length keyword argument list.

> **Note:** For another look at `*args` and `**kwargs`, see Python args and kwargs: Demystified.

All three—standard positional parameters, `*args`, and `**kwargs`—can be used in one Python function definition. If so, then they should be specified in that order:

Python

```
>>> def f(a, b, *args, **kwargs):
...     print(F'a = {a}')
...     print(F'b = {b}')
...     print(F'args = {args}')
...     print(F'kwargs = {kwargs}')
...

>>> f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)
a = 1
b = 2
args = ('foo', 'bar', 'baz', 'qux')
kwargs = {'x': 100, 'y': 200, 'z': 300}
```

This provides just about as much flexibility as you could ever need in a function interface!

Multiple Unpackings in a Python Function Call

Python version 3.5 introduced support for additional unpacking generalizations, as outlined in PEP 448. One thing these enhancements allow is **multiple unpackings** in a single Python function call:

Python

```
>>> def f(*args):
...     for i in args:
...             print(i)
...

>>> a = [1, 2, 3]
>>> t = (4, 5, 6)
>>> s = {7, 8, 9}

>>> f(*a, *t, *s)
1
2
3
4
5
6
8
9
7
```

You can specify multiple dictionary unpackings in a Python function call as well:

Python

```
>>> def f(**kwargs):
...     for k, v in kwargs.items():
...             print(k, '->', v)
...

>>> d1 = {'a': 1, 'b': 2}
>>> d2 = {'x': 3, 'y': 4}

>>> f(**d1, **d2)
a -> 1
b -> 2
x -> 3
y -> 4
```

Note: This enhancement is available only in Python version 3.5 or later. If you try this in an earlier version, then you'll get a `SyntaxError` exception.

By the way, the unpacking operators `*` and `**` don't apply only to variables, as in the examples above. You can also use them with literals that are iterable:

Python

```
>>> def f(*args):
...     for i in args:
...             print(i)
...

>>> f(*[1, 2, 3], *[4, 5, 6])
1
2
3
4
5
6

>>> def f(**kwargs):
...     for k, v in kwargs.items():
...             print(k, '->', v)
...

>>> f(**{'a': 1, 'b': 2}, **{'x': 3, 'y': 4})
a -> 1
b -> 2
x -> 3
y -> 4
```

Here, the literal lists `[1, 2, 3]` and `[4, 5, 6]` are specified for tuple unpacking, and the literal dictionaries `{'a': 1, 'b': 2}` and `{'x': 3, 'y': 4}` are specified for dictionary unpacking.

Keyword-Only Arguments

A Python function in version 3.x can be defined so that it takes **keyword-only arguments**. These are function arguments that must be specified by keyword. Let's explore a situation where this might be beneficial.

Suppose you want to write a Python function that takes a variable number of string arguments, concatenates them together separated by a dot (`"."`), and prints them to the console. Something like this will do to start:

Python

```
>>> def concat(*args):
...     print(f'-> {".".join(args)}')
...

>>> concat('a', 'b', 'c')
-> a.b.c
>>> concat('foo', 'bar', 'baz', 'qux')
-> foo.bar.baz.qux
```

As it stands, the output prefix is hard-coded to the string `'-> '`. What if you want to modify the function to accept this as an argument as well, so the user can specify something else? This is one possibility:

Python

```
>>> def concat(prefix, *args):
...     print(f'{prefix}{".".join(args)}')
...

>>> concat('//', 'a', 'b', 'c')
//a.b.c
>>> concat('... ', 'foo', 'bar', 'baz', 'qux')
... foo.bar.baz.qux
```

This works as advertised, but there are a couple of undesirable things about this solution:

1. The `prefix` string is lumped together with the strings to be concatenated. Just from looking at the function call, it isn't clear that the first argument is treated differently from the rest. To know that, you'd have to go back and look at the function definition.

2. `prefix` isn't optional. It always has to be included, and there's no way to assume a default value.

You might think you could overcome the second issue by specifying a parameter with a default value, like this, perhaps:

Python

```
>>> def concat(prefix='-> ', *args):
...     print(f'{prefix}{".".join(args)}')
...
```

Unfortunately, this doesn't work quite right. `prefix` is a **positional parameter**, so the interpreter assumes that the first argument specified in the function call is the intended output prefix. This means there isn't any way to omit it and obtain the default value:

Python

```
>>> concat('a', 'b', 'c')
ab.c
```

What if you try to specify `prefix` as a keyword argument? Well, you can't specify it first:

Python

```
>>> concat(prefix='//', 'a', 'b', 'c')
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

As you've seen previously, when both types of arguments are given, all positional arguments must come before any keyword arguments.

However, you can't specify it last either:

Python

```
>>> concat('a', 'b', 'c', prefix='... ')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: concat() got multiple values for argument 'prefix'
```

Again, `prefix` is a positional parameter, so it's assigned the first argument specified in the call (which is `'a'` in this case). Then, when it's specified again as a keyword argument at the end, Python thinks it's been assigned twice.

Keyword-only parameters help solve this dilemma. In the function definition, specify `*args` to indicate a variable number of positional arguments, and then specify `prefix` after that:

Python

```
>>> def concat(*args, prefix='-> '):
...     print(f'{prefix}{".".join(args)}')
...
```

In that case, `prefix` becomes a keyword-only parameter. Its value will never be filled by a positional argument. It can only be specified by a named keyword argument:

Python

```
>>> concat('a', 'b', 'c', prefix='... ')
... a.b.c
```

Note that this is only possible in Python 3. In versions 2.x of Python, specifying additional parameters after the `*args` variable arguments parameter raises an error.

Keyword-only arguments allow a Python function to take a variable number of arguments, followed by one or more additional **options** as keyword arguments. If you wanted to modify `concat()` so that the separator character can optionally be specified as well, then you could add an additional keyword-only argument:

Python

```
>>> def concat(*args, prefix='-> ', sep='.'):
...     print(f'{prefix}{sep.join(args)}')
...

>>> concat('a', 'b', 'c')
-> a.b.c
>>> concat('a', 'b', 'c', prefix='//')
//a.b.c
>>> concat('a', 'b', 'c', prefix='//', sep='-')
//a-b-c
```

If a keyword-only parameter is given a default value in the function definition (as it is in the example above), and the keyword is omitted when the function is called, then the default value is supplied:

Python

```
>>> concat('a', 'b', 'c')
-> a.b.c
```

If, on the other hand, the parameter isn't given a default value, then it becomes required, and failure to specify it results in an error:

Python

```
>>> def concat(*args, prefix):
...     print(f'{prefix}{".".join(args)}')
...

>>> concat('a', 'b', 'c', prefix='... ')
... a.b.c

>>> concat('a', 'b', 'c')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: concat() missing 1 required keyword-only argument: 'prefix'
```

What if you want to define a Python function that takes a keyword-only argument but doesn't take a variable number of positional arguments? For example, the following function performs the specified operation on two numerical arguments:

Python

```
>>> def oper(x, y, op='+'):
...     if op == '+':
...             return x + y
...     elif op == '-':
...             return x - y
...     elif op == '/':
...             return x / y
...     else:
...             return None
...

>>> oper(3, 4)
7
>>> oper(3, 4, '+')
7
>>> oper(3, 4, '/')
0.75
```

If you wanted to make `op` a keyword-only parameter, then you could add an extraneous dummy variable argument parameter and just ignore it:

Python

```
>>> def oper(x, y, *ignore, op='+'):
...     if op == '+':
...             return x + y
...     elif op == '-':
...             return x - y
...     elif op == '/':
...             return x / y
...     else:
...             return None
...

>>> oper(3, 4, op='+')
7
>>> oper(3, 4, op='/')
0.75
```

The problem with this solution is that `*ignore` absorbs any extraneous positional arguments that might happen to be included:

Python

```
>>> oper(3, 4, "I don't belong here")
7
>>> oper(3, 4, "I don't belong here", op='/')
0.75
```

In this example, the extra argument shouldn't be there (as the argument itself announces). Instead of quietly succeeding, it should really result in an error. The fact that it doesn't is untidy at best. At worst, it may cause a result that appears misleading:

Python

```
>>> oper(3, 4, '/')
7
```

To remedy this, version 3 allows a **variable argument parameter** in a Python function definition to be just a bare asterisk (`*`), with the name omitted:

Python

```
>>> def oper(x, y, *, op='+'):
...     if op == '+':
...             return x + y
...     elif op == '-':
...             return x - y
...     elif op == '/':
...             return x / y
...     else:
...             return None
...

>>> oper(3, 4, op='+')
7
>>> oper(3, 4, op='/')
0.75

>>> oper(3, 4, "I don't belong here")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: oper() takes 2 positional arguments but 3 were given

>>> oper(3, 4, '+')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: oper() takes 2 positional arguments but 3 were given
```

The **bare variable argument parameter** `*` indicates that there aren't any more positional parameters. This behavior generates appropriate error messages if extra ones are specified. It allows keyword-only parameters to follow.

Positional-Only Arguments

As of Python 3.8, function parameters can also be declared **positional-only**, meaning the corresponding arguments must be supplied positionally and can't be specified by keyword.

To designate some parameters as positional-only, you specify a bare slash (`/`) in the parameter list of a function definition. Any parameters to the left of the slash (`/`) must be specified positionally. For example, in the following function definition, `x` and `y` are positional-only parameters, but `z` may be specified by keyword:

Python

```
>>> # This is Python 3.8
>>> def f(x, y, /, z):
...     print(f'x: {x}')
...     print(f'y: {y}')
...     print(f'z: {z}')
...
```

This means that the following calls are valid:

Python

```
>>> f(1, 2, 3)
x: 1
y: 2
z: 3

>>> f(1, 2, z=3)
x: 1
y: 2
z: 3
```

The following call to `f()`, however, is not valid:

Python

```
>>> f(x=1, y=2, z=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got some positional-only arguments passed as keyword arguments:
'x, y'
```

The positional-only and keyword-only designators may both be used in the same function definition:

Python

```
>>> # This is Python 3.8
>>> def f(x, y, /, z, w, *, a, b):
...     print(x, y, z, w, a, b)
...

>>> f(1, 2, z=3, w=4, a=5, b=6)
1 2 3 4 5 6

>>> f(1, 2, 3, w=4, a=5, b=6)
1 2 3 4 5 6
```

In this example:

- **`x` and `y`** are positional-only.
- **`a` and `b`** are keyword-only.
- **`z` and `w`** may be specified positionally or by keyword.

For more information on positional-only parameters, see the Python 3.8 release highlights and the What Are Python Asterisk and Slash Special Parameters For? tutorial.

Docstrings

When the first statement in the body of a Python function is a string literal, it's known as the function's **docstring**. A docstring is used to supply documentation for a function. It can contain the function's purpose, what arguments it takes, information about return values, or any other information you think would be useful.

The following is an example of a function definition with a docstring:

Python

```
>>> def avg(*args):
...     """Returns the average of a list of numeric values."""
...     return sum(args) / len(args)
...
```

Technically, docstrings can use any of Python's quoting mechanisms, but the recommended convention is to **triple-quote** using double-quote characters (`"""`), as shown above. If the docstring fits on one line, then the closing quotes should be on the same line as the opening quotes.

Multi-line docstrings are used for lengthier documentation. A multi-line docstring should consist of a summary line, followed by a blank line, followed by a more detailed description. The closing quotes should be on a line by themselves:

Python

```
>>> def foo(bar=0, baz=1):
...     """Perform a foo transformation.
...
...     Keyword arguments:
...     bar -- magnitude along the bar axis (default=0)
...     baz -- magnitude along the baz axis (default=1)
...     """
...     <function_body>
...
```

Docstring formatting and semantic conventions are detailed in PEP 257.

When a docstring is defined, the Python interpreter assigns it to a special attribute of the function called `__doc__`. This attribute is one of a set of specialized identifiers in Python that are sometimes called **magic attributes** or **magic methods** because they provide special language functionality.

> **Note:** These attributes are also referred to by the colorful nickname dunder attributes and dunder methods. The word **dunder** combines the **d** from *double* and **under** from the underscore character (_). You'll encounter many more dunder attributes and methods in future tutorials in this series.

You can access a function's docstring with the expression `<function_name>.__doc__`. The docstrings for the above examples can be displayed as follows:

Python

```
>>> print(avg.__doc__)
Returns the average of a list of numeric values.

>>> print(foo.__doc__)
Perform a foo transformation.

    Keyword arguments:
    bar -- magnitude along the bar axis (default=0)
    baz -- magnitude along the baz axis (default=1)
```

In the interactive Python interpreter, you can type `help(<function_name>)` to display the docstring for `<function_name>`:

Python

```
>>> help(avg)
Help on function avg in module __main__:

avg(*args)
    Returns the average of a list of numeric values.

>>> help(foo)
Help on function foo in module __main__:

foo(bar=0, baz=1)
    Perform a foo transformation.

    Keyword arguments:
    bar -- magnitude along the bar axis (default=0)
    baz -- magnitude along the baz axis (default=1)
```

It's considered good coding practice to specify a docstring for each Python function you define. For more on docstrings, check out Documenting Python Code: A Complete Guide.

Python Function Annotations

As of version 3.0, Python provides an additional feature for documenting a function called a **function annotation**. Annotations provide a way to attach metadata to a function's parameters and return value.

To add an annotation to a Python function parameter, insert a colon (`:`) followed by any expression after the parameter name in the function definition. To add an annotation to the return value, add the characters `->` and any expression between the closing parenthesis of the parameter list and the colon that terminates the function header. Here's an example:

Python

```
>>> def f(a: '<a>', b: '<b>') -> '<ret_value>':
...     pass
...
```

The annotation for parameter `a` is the string `'<a>'`, for `b` the string `'<b>'`, and for the function return value the string `'<ret_value>'`.

The Python interpreter creates a dictionary from the annotations and assigns them to another special dunder attribute of the function called `__annotations__`. The annotations for the Python function `f()` shown above can be displayed as follows:

Python

```
>>> f.__annotations__
{'a': '<a>', 'b': '<b>', 'return': '<ret_value>'}
```

The keys for the parameters are the parameter names. The key for the return value is the string `'return'`:

Python

```
>>> f.__annotations__['a']
'<a>'
>>> f.__annotations__['b']
'<b>'
>>> f.__annotations__['return']
'<ret_value>'
```

Note that annotations aren't restricted to string values. They can be any expression or object. For example, you might annotate with type objects:

Python

```
>>> def f(a: int, b: str) -> float:
...     print(a, b)
...     return(3.5)
...

>>> f(1, 'foo')
1 foo
3.5

>>> f.__annotations__
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

An annotation can even be a composite object like a list or a dictionary, so it's possible to attach multiple items of metadata to the parameters and return value:

Python

```
>>> def area(
...     r: {
...             'desc': 'radius of circle',
...             'type': float
...        }) -> \
...        {
...             'desc': 'area of circle',
...             'type': float
...        }:
...     return 3.14159 * (r ** 2)
...

>>> area(2.5)
19.6349375

>>> area.__annotations__
{'r': {'desc': 'radius of circle', 'type': <class 'float'>},
'return': {'desc': 'area of circle', 'type': <class 'float'>}}

>>> area.__annotations__['r']['desc']
'radius of circle'
>>> area.__annotations__['return']['type']
<class 'float'>
```

In the example above, an annotation is attached to the parameter `r` and to the return value. Each annotation is a dictionary containing a string description and a type object.

If you want to assign a default value to a parameter that has an annotation, then the default value goes after the annotation:

Python

```
>>> def f(a: int = 12, b: str = 'baz') -> float:
...     print(a, b)
...     return(3.5)
...

>>> f.__annotations__
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}

>>> f()
12 baz
3.5
```

What do annotations do? Frankly, they don't do much of anything. They're just kind of there. Let's look at one of the examples from above again, but with a few minor modifications:

Python

```
>>> def f(a: int, b: str) -> float:
...     print(a, b)
...     return 1, 2, 3
...

>>> f('foo', 2.5)
foo 2.5
(1, 2, 3)
```

What's going on here? The annotations for `f()` indicate that the first argument is `int`, the second argument `str`, and the return value `float`. But the subsequent call to `f()` breaks all the rules! The arguments are `str` and `float`, respectively, and the return value is a tuple. Yet the interpreter lets it all slide with no complaint at all.

Annotations don't impose any **semantic restrictions** on the code whatsoever. They're simply bits of metadata attached to the Python function parameters and return value. Python dutifully stashes them in a dictionary, assigns the dictionary to the function's `__annotations__` dunder attribute, and that's it. Annotations are completely optional and don't have any impact on Python function execution at all.

To quote Amahl in *Amahl and the Night Visitors*, "What's the use of having it then?"

For starters, annotations make good **documentation**. You can specify the same information in the docstring, of course, but placing it directly in the function definition adds clarity. The types of the arguments and the return value are obvious on sight for a function header like this:

Python

```
def f(a: int, b: str) -> float:
```

Granted, the interpreter doesn't enforce adherence to the types specified, but at least they're clear to someone reading the function definition.

Deep Dive: Enforcing Type-Checking

If you were inclined to, you could add code to enforce the types specified in the function annotations. Here's a function that checks the actual type of each argument against what's specified in the annotation for the corresponding parameter. It displays `True` if they match ans `False` if they don't:

Python

```
>>> def f(a: int, b: str, c: float):
...     import inspect
...     args = inspect.getfullargspec(f).args
...     annotations = inspect.getfullargspec(f).annotations
...     for x in args:
...         print(x, '->',
...               'arg is', type(locals()[x]), ',',
...               'annotation is', annotations[x],
...               '/', (type(locals()[x])) is annotations[x])
...

>>> f(1, 'foo', 3.3)
a -> arg is <class 'int'> , annotation is <class 'int'> / True
b -> arg is <class 'str'> , annotation is <class 'str'> / True
c -> arg is <class 'float'> , annotation is <class 'float'> / True

>>> f('foo', 4.3, 9)
a -> arg is <class 'str'> , annotation is <class 'int'> / False
b -> arg is <class 'float'> , annotation is <class 'str'> / False
c -> arg is <class 'int'> , annotation is <class 'float'> / False

>>> f(1, 'foo', 'bar')
a -> arg is <class 'int'> , annotation is <class 'int'> / True
b -> arg is <class 'str'> , annotation is <class 'str'> / True
c -> arg is <class 'str'> , annotation is <class 'float'> / False
```

(The `inspect` module contains functions that obtain useful information about live objects—in this case, function `f()`.)

A function defined like the one above could, if desired, take some sort of corrective action when it detects that the passed arguments don't conform to the types specified in the annotations.

In fact, a scheme for using annotations to perform static type checking in Python is described in PEP 484. A free static type checker for Python called mypy is available, which is built on the PEP 484 specification.

There's another benefit to using annotations as well. The standardized format in which annotation information is stored in the `__annotations__` attribute lends itself to the parsing of function signatures by automated tools.

When it comes down to it, annotations aren't anything especially magical. You could even define your own without the special syntax that Python provides. Here's a Python function definition with type object annotations attached to the parameters and return value:

Python

```
>>> def f(a: int, b: str) -> float:
...     return
...

>>> f.__annotations__
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

The following is essentially the same function, with the `__annotations__` dictionary constructed manually:

Python

```
>>> def f(a, b):
...     return
...

>>> f.__annotations__ = {'a': int, 'b': str, 'return': float}

>>> f.__annotations__
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

The effect is identical in both cases, but the first is more visually appealing and readable at first glance.

In fact, the `__annotations__` attribute isn't significantly different from most other attributes of a function. For example, it can be modified dynamically. You could choose to use the return value attribute to count how many times a function is executed:

Python

```
>>> def f() -> 0:
...     f.__annotations__['return'] += 1
...     print(f"f() has been executed {f.__annotations__['return']} time(s)")
...

>>> f()
f() has been executed 1 time(s)
>>> f()
f() has been executed 2 time(s)
>>> f()
f() has been executed 3 time(s)
```

Python function annotations are nothing more than dictionaries of metadata. It just happens that you can create them with convenient syntax that's supported by the interpreter. They're whatever you choose to make of them.

Conclusion

As applications grow larger, it becomes increasingly important to modularize code by breaking it up into smaller functions of manageable size. You now hopefully have all the tools you need to do this.

You've learned:

- How to create a **user-defined function** in Python
- Several different ways you can pass **arguments** to a function
- How you can **return** data from a function to its caller
- How to add documentation to functions with **docstrings** and **annotations**

Next up in this series are two tutorials that cover **searching** and **pattern matching**. You will get an in-depth look at a Python module called **re**, which contains functionality for searching and matching using a versatile pattern syntax called a **regular expression**.

« Python String Formatting Techniques

Defining Your Own Python Function

Regular Expressions: Regexes in Python (Part 1) »

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Defining and Calling Python Functions**