Chapter 9. Functions

The smaller the function, the greater the management.

—C. Northcote Parkinson

So far, all of our Python code examples have been little fragments. These are good for small tasks, but no one wants to retype fragments all the time. We need some way of organizing larger code into manageable pieces.

The first step to code reuse is the *function*: a named piece of code, separate from all others. A function can take any number and type of input *parameters* and return any number and type of output *results*.

You can do two things with a function:

- *Define* it, with zero or more parameters
- *Call* it, and get zero or more results

Define a Function with def

To define a Python function, you type `def`, the function name, parentheses enclosing any input *parameters* to the function, and then finally, a colon (`:`). Function names have the same rules as variable names (they must start with a letter or `_` and contain only letters, numbers, or `_`).

Let’s take things one step at a time, and first define and call a function that has no parameters. Here’s the simplest Python function:

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

would indent code under an `if` statement. Python requires the `pass` statement to show that this function does nothing. It's the equivalent of *This page intentionally left blank* (even though it isn't anymore).

Call a Function with Parentheses

You call this function just by typing its name and parentheses. It works as advertised, doing nothing, but doing it very well:

```
>>> do_nothing()
>>>
```

Now let's define and call another function that has no parameters but prints a single word:

```
>>> def make_a_sound():
...     print('quack')
...
>>> make_a_sound()
quack
```

When you called the `make_a_sound()` function, Python ran the code inside its definition. In this case, it printed a single word and returned to the main program.

Let's try a function that has no parameters but *returns* a value:

```
>>> def agree():
...    return True
...
```

You can call this function and test its returned value by using `if`:

```
>>> if agree():
...     print('Splendid!')
... else:
...     print('That was unexpected.')
...
Splendid!
```

You've just made a big step. The combination of functions with tests such as `if` and loops such as `while` make it possible for you to do things that you could not do before.

Arguments and Parameters

At this point, it's time to put something between those parentheses. Let's define the function `echo()` with one parameter called `anything`. It uses the `return` statement to send the value of `anything` back to its caller twice, with a space between:

```
>>> def echo(anything):
...    return anything + ' ' + anything
...
>>>
```

Now let's call `echo()` with the string `'Rumplestiltskin'`:

```
>>> echo('Rumplestiltskin')
'Rumplestiltskin Rumplestiltskin'
```

The values you pass into the function when you call it are known as *arguments*. When you call a function with arguments, the values of those arguments are copied to their corresponding *parameters* inside the function.

NOTE

Saying it another way: they're called *arguments* outside of the function, but *parameters* inside.

In the previous example, the function `echo()` was called with the argument string `'Rumplestiltskin'`. This value was copied within `echo()` to the parameter `anything`, and then returned (in this case doubled, with a space) to the caller.

These function examples were pretty basic. Let's write a function that takes an input argument and actually does something with it. We'll adapt the earlier code fragment that comments on a color. Call it `commentary` and have it take an input string parameter called `color`. Make it return the string description to its caller, which can decide what to do with it:

```
>>> def commentary(color):
...     if color == 'red':
...         return "It's a tomato."
...     elif color == "green":
...         return "It's a green pepper."
...     elif color == 'bee purple':
...         return "I don't know what it is, but only bees can see it."
...     else:
...         return "I've never heard of the color "  + color +  "."
...
>>>
```

Call the function `commentary()` with the string argument `'blue'`.

```
>>> comment = commentary('blue')
```

The function does the following:

- Assigns the value `'blue'` to the function's internal `color` parameter
- Runs through the `if`-`elif`-`else` logic chain
- Returns a string

The caller then assigns the string to the variable `comment`.

What did we get back?

```
>>> print(comment)
I've never heard of the color blue.
```

A function can take any number of input arguments (including zero) of any type. It can return any number of output results (also including zero) of any type. If a function doesn't call `return` explicitly, the caller gets the result `None`.

```
>>> print(do_nothing())
None
```

None Is Useful

`None` is a special Python value that holds a place when there is nothing to say. It is not the same as the boolean value `False`, although it looks false when evaluated as a boolean. Here's an example:

```
>>> thing = None
>>> if thing:
...     print("It's some thing")
... else:
...     print("It's no thing")
...
It's no thing
```

To distinguish `None` from a boolean `False` value, use Python's `is` operator:

```
>>> thing = None
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

This seems like a subtle distinction, but it's important in Python. You'll need `None` to distinguish a missing value from an empty value. Remember that zero-valued integers or floats, empty strings (`''`), lists (`[]`), tuples (`(,)`), dictionaries (`{}`), and sets (`set()`) are all `False`, but are not the same as `None`.

Let's write a quick function that prints whether its argument is `None`, `True`, or `False`:

```
>>> def whatis(thing):
...     if thing is None:
...         print(thing, "is None")
...     elif thing:
...         print(thing, "is True")
...     else:
...         print(thing, "is False")
...
```

Let’s run some sanity tests:

```
>>> whatis(None)
None is None
>>> whatis(True)
True is True
>>> whatis(False)
False is False
```

How about some real values?

```
>>> whatis(0)
0 is False
>>> whatis(0.0)
0.0 is False
>>> whatis('')
 is False
>>> whatis("")
 is False
>>> whatis('''''')
 is False
>>> whatis(())
() is False
>>> whatis([])
[] is False
>>> whatis({})
{} is False
>>> whatis(set())
set() is False
```

```
>>> whatis(0.00001)
1e-05 is True
```

```
>>> whatis([0])
[0] is True
>>> whatis([''])
[''] is True
>>> whatis(' ')
  is True
```

Positional Arguments

Python handles function arguments in a manner that's very flexible, when compared to many languages. The most familiar types of arguments are *positional arguments*, whose values are copied to their corresponding parameters in order.

This function builds a dictionary from its positional input arguments and returns it:

```
>>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

Although very common, a downside of positional arguments is that you need to remember the meaning of each position. If we forgot and called `menu()` with wine as the last argument instead of the first, the meal would be very different:

```
>>> menu('beef', 'bagel', 'bordeaux')
{'wine': 'beef', 'entree': 'bagel', 'dessert': 'bordeaux'}
```

Keyword Arguments

To avoid positional argument confusion, you can specify arguments by the names of their corresponding parameters, even in a different order from their definition in the function:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'wine': 'bordeaux', 'entree': 'beef', 'dessert': 'bagel'}
```

You can mix positional and keyword arguments. Let's specify the wine first, but use keyword arguments for the entree and dessert:

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'wine': 'frontenac', 'entree': 'fish', 'dessert': 'flan'}
```

If you call a function with both positional and keyword arguments, the positional arguments need to come first.

Specify Default Parameter Values

You can specify default values for parameters. The default is used if the caller does not provide a corresponding argument. This bland-sounding feature can actually be quite useful. Using the previous example:

```
>>> def menu(wine, entree, dessert='pudding'):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

This time, try calling `menu()` without the `dessert` argument:

```
>>> menu('chardonnay', 'chicken')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'pudding'}
```

If you do provide an argument, it's used instead of the default:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'wine': 'dunkelfelder', 'entree': 'duck', 'dessert': 'doughnut'}
```

NOTE

Default parameter values are calculated when the function is *defined*, not when it is run. A common error with new (and sometimes not-so-new) Python programmers is to use a mutable data type such as a list or dictionary as a default parameter.

In the following test, the `buggy()` function is expected to run each time with a fresh empty `result` list, add the `arg` argument to it, and then print a single-item list. However, there's a bug: it's empty only the first time it's called. The second time, `result` still has one item from the previous call:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a')
['a']
>>> buggy('b')   # expect ['b']
['a', 'b']
```

It would have worked if it had been written like this:

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...
>>> works('a')
['a']
>>> works('b')
['b']
```

The fix is to pass in something else to indicate the first call:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

This is sometimes a Python job interview question. You've been warned.

Explode/Gather Positional Arguments with *

If you've programmed in C or C++, you might assume that an asterisk (`*`) in a Python program has something to do with a *pointer*. Nope, Python doesn't have pointers.

When used inside the function with a parameter, an asterisk groups a variable number of positional arguments into a single tuple of parameter values. In the following example, `args` is the parameter tuple that resulted from zero or more arguments that were passed to the function `print_args()`:

```
>>> def print_args(*args):
...     print('Positional tuple:', args)
...
```

If you call the function with no arguments, you get nothing in `*args`:

```
>>> print_args()
Positional tuple: ()
```

Whatever you give it will be printed as the `args` tuple:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional tuple: (3, 2, 1, 'wait!', 'uh...')
```

This is useful for writing functions such as `print()` that accept a variable number of arguments. If your function has *required* positional arguments, as well, put them first; `*args` goes at the end and grabs all the rest:

```
>>> def print_more(required1, required2, *args):
...     print('Need this one:', required1)
...     print('Need this one too:', required2)
...     print('All the rest:', args)
...
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
Need this one: cap
Need this one too: gloves
All the rest: ('scarf', 'monocle', 'mustache wax')
```

NOTE

When using `*`, you don’t need to call the tuple argument `*args`, but it’s a common idiom in Python. It’s also common to use `*args` inside the function, as in the preceding example, although technically it’s called a parameter and could be referred to as `*params`.

Summarizing:

- You can pass positional argument to a function, which will match them inside to positional parameters. This is what you’ve seen so far in this book.
- You can pass a tuple argument to a function, and inside it will be a tuple parameter. This is a simple case of the preceding one.
- You can pass positional arguments to a function, and gather them inside as the parameter `*args`, which resolves to the tuple `args`. This was described in this section.
- You can also “explode” a tuple argument called `args` to positional parameters `*args` inside the function, which will be regathered inside into the tuple parameter `args`:

```
>>> print_args(2, 5, 7, 'x')
Positional tuple: (2, 5, 7, 'x')
>>> args = (2,5,7,'x')
>>> print_args(args)
Positional tuple: ((2, 5, 7, 'x'),)
>>> print_args(*args)
Positional tuple: (2, 5, 7, 'x')
```

You can only use the `*` syntax in a function call or definition:

```
>>> *args
  File "<stdin>", line 1
SyntaxError: can't use starred expression here
```

So:

- Outside the function, `*args` explodes the tuple `args` into comma-separated positional parameters.
- Inside the function, `*args` gathers all of the positional arguments into a single `args` tuple. You could use the names `*params` and `params`, but it’s com-

mon practice to use `*args` for both the outside argument and inside parameter.

Readers with synesthesia might also faintly hear `*args` as *puff-args* on the outside and *inhale-args* on the inside, as values are either exploded or gathered.

Explode/Gather Keyword Arguments with **

You can use two asterisks (`**`) to group keyword arguments into a dictionary, where the argument names are the keys, and their values are the corresponding dictionary values. The following example defines the function `print_kwargs()` to print its keyword arguments:

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

Now try calling it with some keyword arguments:

```
>>> print_kwargs()
Keyword arguments: {}
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot',
'entree': 'mutton'}
```

Inside the function, `kwargs` is a dictionary parameter.

Argument order is:

- Required positional arguments
- Optional positional arguments (`*args`)
- Optional keyword arguments (`**kwargs`)

As with `args`, you don't need to call this keyword argument `kwargs`, but it's common usage.[1]

The `**` syntax is valid only in a function call or definition:[2]

```
>>> **kwparams
  File "<stdin>", line 1
    **kwparams
     ^
SyntaxError: invalid syntax
```

Summarizing:

- You can pass keyword arguments to a function, which will match them inside to keyword parameters. This is what you've seen so far.
- You can pass a dictionary argument to a function, and inside it will be dictionary parameters. This is a simple case of the preceding one.
- You can pass one or more keyword arguments (*name=value*) to a function, and gather them inside as `**kwargs`, which resolves to the dictionary parameter called `kwargs`. This was described in this section.
- Outside a function, `**kwargs` *explodes* a dictionary `kwargs` into *name=value* arguments.
- Inside a function, `**kwargs` *gathers* `name=value` arguments into the single dictionary parameter `kwargs`.

If auditory hallucinations help, imagine a puff for each asterisk exploding outside the function, and a little inhaling sound for each one gathering inside.

Keyword-Only Arguments

It's possible to pass in a keyword argument that has the same name as a positional parameter, probably not resulting in what you want. Python 3 lets you specify *keyword-only arguments*. As the name says, they must be provided as *name=value*, not positionally as *value*. The single `*` in the function definition means that the following parameters `start` and `end` must be provided as named arguments if we don't want their default values:

```
>>> def print_data(data, *, start=0, end=100):
...     for value in (data[start:end]):
...         print(value)
...
>>> data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> print_data(data)
a
```

```
b
c
d
e
f
>>> print_data(data, start=4)
e
f
>>> print_data(data, end=2)
a
b
```

Mutable and Immutable Arguments

Remember that if you assigned the same list to two variables, you could change it by using either one? And that you could not if the variables both referred to something like an integer or a string? That was because the list was mutable and the integer and string were immutable.

You need to watch for the same behavior when passing arguments to functions. If an argument is mutable, its value can be changed *from inside the function* via its corresponding parameter:[3]

```
>>> outside = ['one', 'fine', 'day']
>>> def mangle(arg):
...     arg[1] = 'terrible!'
...
>>> outside
['one', 'fine', 'day']
>>> mangle(outside)
>>> outside
['one', 'terrible!', 'day']
```

It’s good practice to, uh, not do this.[4] Either document that an argument may be changed, or `return` the new value.

Docstrings

Readability counts, the Zen of Python verily saith. You can attach documentation to a function definition by including a string at the beginning of the function body. This is the function's *docstring*:

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

You can make a docstring quite long, and even add rich formatting if you want:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

To print a function's docstring, call the Python `help()` function. Pass the function's name to get a listing of arguments along with the nicely formatted docstring:

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

echo(anything)
    echo returns its input argument
```

If you want to see just the raw docstring, without the formatting:

```
>>> print(echo.__doc__)
echo returns its input argument
```

That odd-looking `__doc__` is the internal name of the docstring as a variable within the function. Double underscores (aka *dunder* in Python-speak) are used in many places to name Python internal variables, because programmers are unlikely to use them in their own variable names.

Functions Are First-Class Citizens

I've mentioned the Python mantra, *everything is an object*. This includes numbers, strings, tuples, lists, dictionaries—and functions, as well. Functions are first-class citizens in Python. You can assign them to variables, use them as arguments to other functions, and return them from functions. This gives you the capability to do some things in Python that are difficult-to-impossible to carry out in many other languages.

To test this, let's define a simple function called `answer()` that doesn't have any arguments; it just prints the number `42`:

```
>>> def answer():
...     print(42)
```

If you run this function, you know what you'll get:

```
>>> answer()
42
```

Now let's define another function named `run_something`. It has one argument called `func`, a function to run. Once inside, it just calls the function:

```
>>> def run_something(func):
...     func()
```

If we pass `answer` to `run_something()`, we're using a function as data, just as with anything else:

```
>>> run_something(answer)
42
```

Notice that you passed `answer`, not `answer()`. In Python, those parentheses mean *call this function*. With no parentheses, Python just treats the function like any other object. That's because, like everything else in Python, it *is* an object:

```
>>> type(run_something)
<class 'function'>
```

Let's try running a function with arguments. Define a function `add_args()` that prints the sum of its two numeric arguments, `arg1` and `arg2`:

```
>>> def add_args(arg1, arg2):
...     print(arg1 + arg2)
```

And what is `add_args()`?

```
>>> type(add_args)
<class 'function'>
```

At this point, let's define a function called `run_something_with_args()` that takes three arguments:

`func`
: The function to run

`arg1`
: The first argument for `func`

`arg2`
: The second argument for `func`

```
>>> def run_something_with_args(func, arg1, arg2):
...     func(arg1, arg2)
```

When you call `run_something_with_args()`, the function passed by the caller is assigned to the `func` parameter, whereas `arg1` and `arg2` get the values that follow in the argument list. Then, running `func(arg1, arg2)` executes that function with those arguments because the parentheses told Python to do so.

Let's test it by passing the function name `add_args` and the arguments `5` and `9` to `run_something_with_args()`:

```
>>> run_something_with_args(add_args, 5, 9)
14
```

Within the function `run_something_with_args()`, the function name argument `add_args` was assigned to the parameter `func`, `5` to `arg1`, and `9` to `arg2`. This ended up running:

```
add_args(5, 9)
```

You can combine this with the `*args` and `**kwargs` techniques.

Let's define a test function that takes any number of positional arguments, calculates their sum by using the `sum()` function, and then returns that sum:

```
>>> def sum_args(*args):
...     return sum(args)
```

I haven't mentioned `sum()` before. It's a built-in Python function that calculates the sum of the values in its iterable numeric (int or float) argument.

Let's define the new function `run_with_positional_args()`, which takes a function and any number of positional arguments to pass to it:

```
>>> def run_with_positional_args(func, *args):
...     return func(*args)
```

Now go ahead and call it:

```
>>> run_with_positional_args(sum_args, 1, 2, 3, 4)
10
```

You can use functions as elements of lists, tuples, sets, and dictionaries. Functions are immutable, so you can also use them as dictionary keys.

Inner Functions

You can define a function within another function:

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

An inner function can be useful when performing some complex task more than once within another function, to avoid loops or code duplication. For a string example, this inner function adds some text to its argument:

```
>>> def knights(saying):
...     def inner(quote):
...         return "We are the knights who say: '%s'" % quote
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

Closures

An inner function can act as a *closure*. This is a function that is dynamically generated by another function and can both change and remember the values of variables that were created outside the function.

The following example builds on the previous `knights()` example. Let's call the new one `knights2()`, because we have no imagination, and turn the `inner()` function into a closure called `inner2()`. Here are the differences:

- `inner2()` uses the outer `saying` parameter directly instead of getting it as an argument.
- `knights2()` returns the `inner2` function name instead of calling it:

```
>>> def knights2(saying):
...     def inner2():
...         return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

The `inner2()` function knows the value of `saying` that was passed in and remembers it. The line `return inner2` returns this specialized copy of the `inner2` function (but doesn't call it). That's a kind of closure: a dynamically created function that remembers where it came from.

Let's call `knights2()` twice, with different arguments:

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

Okay, so what are `a` and `b`?

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

They're functions, but they're also closures:

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

If we call them, they remember the `saying` that was used when they were created by `knights2`:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

Anonymous Functions: lambda

A Python *lambda function* is an anonymous function expressed as a single statement. You can use it instead of a normal tiny function.

To illustrate it, let's first make an example that uses normal functions. To begin, let's define the function `edit_story()`. Its arguments are the following:

- `words`—a list of words
- `func`—a function to apply to each word in `words`

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

Now we need a list of words and a function to apply to each word. For the words, here's a list of (hypothetical) sounds made by my cat if he (hypothetically) missed one of the stairs:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

And for the function, this will capitalize each word and append an exclamation point, perfect for feline tabloid newspaper headlines:

```
>>> def enliven(word):   # give that prose more punch
...     return word.capitalize() + '!'
```

Mixing our ingredients:

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Finally, we get to the lambda. The `enliven()` function was so brief that we could replace it with a lambda:

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
```

A lambda has zero or more comma-separated arguments, followed by a colon (`:`), and then the definition of the function. We're giving this lambda one argument, `word`. You don't use parentheses with `lambda` as you would when calling a function created with `def`.

Often, using real functions such as `enliven()` is much clearer than using lambdas. Lambdas are mostly useful for cases in which you would otherwise need to define many tiny functions and remember what you called them all. In particular, you can use lambdas in graphical user interfaces to define *callback functions*; see Chapter 20 for examples.

Generators

A *generator* is a Python sequence creation object. With it, you can iterate through potentially huge sequences without creating and storing the entire sequence in memory at once. Generators are often the source of data for iterators. If you recall, we already used one of them, `range()`, in earlier code examples to generate a series of integers. In Python 2, `range()` returns a list, which limits it to fit in memory. Python 2 also has the generator `xrange()`, which became the normal `range()` in Python 3. This example adds all the integers from 1 to 100:

```
>>> sum(range(1, 101))
5050
```

Every time you iterate through a generator, it keeps track of where it was the last time it was called and returns the next value. This is different from a normal function, which has no memory of previous calls and always starts at its first line with the same state.

Generator Functions

If you want to create a potentially large sequence, you can write a *generator function*. It's a normal function, but it returns its value with a `yield` statement rather than `return`. Let's write our own version of `range()`:

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

It's a normal function:

```
>>> my_range
<function my_range at 0x10193e268>
```

And it returns a generator object:

```
>>> ranger = my_range(1, 5)
>>> ranger
<generator object my_range at 0x101a0a168>
```

We can iterate over this generator object:

```
>>> for x in ranger:
...     print(x)
...
1
2
3
4
```

NOTE

A generator can be run only once. Lists, sets, strings, and dictionaries exist in memory, but a generator creates its values on the fly and hands them out one at a time through an iterator. It doesn't remember them, so you can't restart or back up a generator.

If you try to iterate this generator again, you'll find that it's tapped out:

```
>>> for try_again in ranger:
...     print(try_again)
...
>>>
```

Generator Comprehensions

You've seen comprehensions for lists, dictionaries, and sets. A *generator comprehension* looks like those, but is surrounded by parentheses instead of square or curly brackets. It's like a shorthand version of a generator function, doing the `yield` invisibly, and also returns a generator object:

```
>>> genobj = (pair for pair in zip(['a', 'b'], ['1', '2']))
>>> genobj
<generator object <genexpr> at 0x10308fde0>
>>> for thing in genobj:
...     print(thing)
...
('a', '1')
('b', '2')
```

Decorators

Sometimes, you want to modify an existing function without changing its source code. A common example is adding a debugging statement to see what arguments were passed in.

A *decorator* is a function that takes one function as input and returns another function. Let's dig into our bag of Python tricks and use the following:

- `*args` and `**kwargs`
- Inner functions
- Functions as arguments

The function `document_it()` defines a decorator that will do the following:

- Print the function's name and the values of its arguments
- Run the function with the arguments
- Print the result
- Return the modified function for use

Here's what the code looks like:

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
```

Whatever `func` you pass to `document_it()`, you get a new function that includes the extra statements that `document_it()` adds. A decorator doesn't actually have to run any code from `func`, but `document_it()` calls `func` partway through so that you get the results of `func` as well as all the extras.

So, how do you use this? You can apply the decorator manually:

```
>>> def add_ints(a, b):
...    return a + b
...
>>> add_ints(3, 5)
8
>>> cooler_add_ints = document_it(add_ints)  # manual decorator assignment
>>> cooler_add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
```

```
Result: 8
8
```

As an alternative to the manual decorator assignment we just looked at, you can add @`decorator_name` before the function that you want to decorate:

```
>>> @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Start function add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

You can have more than one decorator for a function. Let's write another decorator called `square_it()` that squares the result:

```
>>> def square_it(func):
...     def new_function(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result * result
...     return new_function
...
```

The decorator that's used closest to the function (just above the `def`) runs first and then the one above it. Either order gives the same end result, but you can see how the intermediate steps change:

```
>>> @document_it
... @square_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
```

```
Result: 64
64
```

Let’s try reversing the decorator order:

```
>>> @square_it
... @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
64
```

Namespaces and Scope

> *Desiring this man’s art and that man’s scope*
>
> —William Shakespeare

A name can refer to different things, depending on where it’s used. Python programs have various *namespaces*—sections within which a particular name is unique and unrelated to the same name in other namespaces.

Each function defines its own namespace. If you define a variable called `x` in a main program and another variable called `x` in a function, they refer to different things. But the walls can be breached: if you need to, you can access names in other namespaces in various ways.

The main part of a program defines the *global* namespace; thus, the variables in that namespace are *global variables*.

You can get the value of a global variable from within a function:

```
>>> animal = 'fruitbat'
>>> def print_global():
```

```
...     print('inside print_global:', animal)
...
>>> print('at the top level:', animal)
at the top level: fruitbat
>>> print_global()
inside print_global: fruitbat
```

But if you try to get the value of the global variable *and* change it within the function, you get an error:

```
>>> def change_and_print_global():
...     print('inside change_and_print_global:', animal)
...     animal = 'wombat'
...     print('after the change:', animal)
...
>>> change_and_print_global()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in change_and_print_global
UnboundLocalError: local variable 'animal' referenced before assignment
```

If you just change it, it changes a different variable also named `animal`, but this variable is inside the function:

```
>>> def change_local():
...     animal = 'wombat'
...     print('inside change_local:', animal, id(animal))
...
>>> change_local()
inside change_local: wombat 4330406160
>>> animal
'fruitbat'
>>> id(animal)
4330390832
```

What happened here? The first line assigned the string `'fruitbat'` to a global variable named `animal`. The `change_local()` function also has a variable named `animal`, but that's in its local namespace.

I used the Python function `id()` here to print the unique value for each object and prove that the variable `animal` inside `change_local()` is not the same as `animal` at the main level of the program.

To access the global variable rather than the local one within a function, you need to be explicit and use the `global` keyword (you knew this was coming: *explicit is better than implicit*):

```
>>> animal = 'fruitbat'
>>> def change_and_print_global():
...     global animal
...     animal = 'wombat'
...     print('inside change_and_print_global:', animal)
...
>>> animal
'fruitbat'
>>> change_and_print_global()
inside change_and_print_global: wombat
>>> animal
'wombat'
```

If you don't say `global` within a function, Python uses the local namespace and the variable is local. It goes away after the function completes.

Python provides two functions to access the contents of your namespaces:

- `locals()` returns a dictionary of the contents of the local namespace.
- `globals()` returns a dictionary of the contents of the global namespace.

And here they are in use:

```
>>> animal = 'fruitbat'
>>> def change_local():
...     animal = 'wombat'  # local variable
...     print('locals:', locals())
...
>>> animal
'fruitbat'
>>> change_local()
locals: {'animal': 'wombat'}
>>> print('globals:', globals()) # reformatted a little for presentation
globals: {'animal': 'fruitbat',
'__doc__': None,
'change_local': <function change_local at 0x1006c0170>,
'__package__': None,
'__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins'>}
>>> animal
'fruitbat'
```

The local namespace within `change_local()` contained only the local variable `animal`. The global namespace contained the separate global variable `animal` and a number of other things.

Uses of _ and __ in Names

Names that begin and end with two underscores (`__`) are reserved for use within Python, so you should not use them with your own variables. This naming pattern was chosen because it seemed unlikely to be selected by application developers for their own variables.

For instance, the name of a function is in the system variable `*function*.__name__`, and its documentation string is `*function*.__doc__`:

```
>>> def amazing():
...     '''This is the amazing function.
...     Want to see it again?'''
...     print('This function is named:', amazing.__name__)
...     print('And its docstring is:', amazing.__doc__)
...
>>> amazing()
This function is named: amazing
And its docstring is: This is the amazing function.
    Want to see it again?
```

As you saw in the earlier `globals` printout, the main program is assigned the special name `__main__`.

Recursion

So far, we've called functions that do some things directly, and maybe call other functions. But what if a function calls itself?[5] This is called *recursion*. Like an unbroken infinite loop with `while` or `for`, you don't want infinite recursion. Do we still need to worry about cracks in the space-time continuum?

Python saves the universe again by raising an exception if you get too deep:

```
>>> def dive():
...     return dive()
...
>>> dive()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in dive
  File "<stdin>", line 2, in dive
  File "<stdin>", line 2, in dive
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Recursion is useful when you're dealing with "uneven" data, like lists of lists of lists. Suppose that you want to "flatten" all sublists of a list,[6] no matter how deeply nested. A generator function is just the thing:

```
>>> def flatten(lol):
...     for item in lol:
...         if isinstance(item, list):
...             for subitem in flatten(item):
...                 yield subitem
...         else:
...             yield item
...
>>> lol = [1, 2, [3,4,5], [6,[7,8,9], []]]
>>> flatten(lol)
<generator object flatten at 0x10509a750>
>>> list(flatten(lol))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python 3.3 added the `yield from` expression, which lets a generator hand off some work to another generator. We can use it to simplify `flatten()`:

```
>>> def flatten(lol):
...     for item in lol:
...         if isinstance(item, list):
...             yield from flatten(item)
...         else:
...             yield item
...
>>> lol = [1, 2, [3,4,5], [6,[7,8,9], []]]
>>> list(flatten(lol))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Async Functions

The keywords `async` and `await` were added to Python 3.5 to define and run *asynchronous functions*. They're:

- Relatively new
- Different enough to be harder to understand
- Will become more important and better known over time

For these reasons, I've moved discussion of these and other async topics to Appendix C.

For now, you need to know that if you see `async` before the `def` line for a function, it's an asynchronous function. Likewise, if you see `await` before a function call, that function is asynchronous.

The main difference between asynchronous and normal functions is that async ones can "give up control" rather than running to completion.

Exceptions

In some languages, errors are indicated by special function return values. When things go south,[7] Python uses *exceptions*: code that is executed when an associated error occurs.

You've seen some of these already, such as accessing a list or tuple with an out-of-range position, or a dictionary with a nonexistent key. When you run code that might fail under some circumstances, you also need appropriate *exception handlers* to intercept any potential errors.

It's good practice to add exception handling anywhere an exception might occur to let the user know what is happening. You might not be able to fix the problem, but at least you can note the circumstances and shut your program down gracefully. If an exception occurs in some function and is not caught there, it *bubbles up* until it is caught by a matching handler in some calling function. If you don't provide your own exception handler, Python prints an error message and some information about where the error occurred and then terminates the program, as demonstrated in the following snippet:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Handle Errors with try and except

Do, or do not. There is no try.

—Yoda

Rather than leaving things to chance, use `try` to wrap your code, and `except` to provide the error handling:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...     print('Need a position between 0 and', len(short_list)-1, ' but got',
...           position)
...
Need a position between 0 and 2 but got 5
```

The code inside the `try` block is run. If there is an error, an exception is raised and the code inside the `except` block runs. If there are no errors, the `except` block is skipped.

Specifying a plain `except` with no arguments, as we did here, is a catchall for any exception type. If more than one type of exception could occur, it's best to provide a separate exception handler for each. No one forces you to do this; you can use a bare `except` to catch all exceptions, but your treatment of them would probably be generic (something akin to printing *Some error occurred*). You can use any number of specific exception handlers.

Sometimes, you want exception details beyond the type. You get the full exception object in the variable *name* if you use the form:

```
except exceptiontype as name
```

The example that follows looks for an `IndexError` first, because that's the exception type raised when you provide an illegal position to a sequence. It saves an `IndexError` exception in the variable `err`, and any other exception in the variable `other`. The example prints everything stored in `other` to show what you get in that object:

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError as err:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
1
Position [q to quit]? 2
3
```

```
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? 2
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? q
```

Inputting position `3` raised an `IndexError` as expected. Entering `two` annoyed the `int()` function, which we handled in our second, catchall `except` code.

Make Your Own Exceptions

The previous section discussed handling exceptions, but all of the exceptions (such as `IndexError`) were predefined in Python or its standard library. You can use any of these for your own purposes. You can also define your own exception types to handle special situations that might arise in your own programs.

NOTE

This requires defining a new object type with a *class*—something we don't get into until Chapter 10. So, if you're unfamiliar with classes, you might want to return to this section later.

An exception is a class. It is a child of the class `Exception`. Let's make an exception called `UppercaseException` and raise it when we encounter an uppercase word in a string:

```
>>> class UppercaseException(Exception):
...     pass
...
>>> words = ['eenie', 'meenie', 'miny', 'MO']
>>> for word in words:
...     if word.isupper():
...         raise UppercaseException(word)
...
Traceback (most recent call last):
    File "<stdin>", line 3, in <module>
  __main__.UppercaseException: MO
```

We didn't even define any behavior for `UppercaseException` (notice we just used `pass`), letting its parent class `Exception` figure out what to print when the exception was raised.

You can access the exception object itself and print it:

```
>>> try:
...     raise OopsException('panic')
... except OopsException as exc:
...     print(exc)
...
panic
```

Coming Up

Objects! We had to get to them sometime in a book about an object-oriented language.

Things to Do

9.1 Define a function called `good()` that returns the following list: `['Harry', 'Ron', 'Hermione']`.

9.2 Define a generator function called `get_odds()` that returns the odd numbers from `range(10)`. Use a `for` loop to find and print the third value returned.

9.3 Define a decorator called `test` that prints `'start'` when a function is called, and `'end'` when it finishes.

9.4 Define an exception called `OopsException`. Raise this exception to see what happens. Then, write the code to catch this exception and print `'Caught an oops'`.

1 Although *Args* and *Kwargs* sound like the names of pirate parrots.

2 Or, as of Python 3.5, a dictionary merge of the form `{**a, **b}`, as you saw in Chapter 8.

3 Like the teens-in-peril movies where they learn “The call’s coming from inside the house!”

4 Like the old doctor joke: “It hurts when I do this.” “Well, then don’t do that.”

5 It’s like saying, “I wish I had a dollar for every time I wished I had a dollar.”

6 Another Python interview question. Collect the whole set!

7 Is this northern hemispherism? Do Aussies and Kiwis say that things go “north” when they mess up?