Defining Functions and Using Built-ins

Functions are the fundamental unit of work in Python. A function in Python performs a task and returns a result. In this chapter, we will start with the basics of functions. Then we look at using the built-in functions. These are the core functions that are always available, meaning they don't require an explicit import into your namespace. Next we will look at some alternative ways of defining functions, such as lambdas and classes. We will also look at more advanced types of functions, namely closures and generator functions.

As you will see, functions are very easy to define and use. Python encourages an incremental style of development that you can leverage when writing functions. So how does this work out in practice? Often when writing a function it may make sense to start with a sequence of statements and just try it out in a console. Or maybe just write a short script in an editor. The idea is to just to prove a path and answer such questions as, "Does this API work in the way I expect?" Because top-level code in a console or script works just like it does in a function, it's easy to later isolate this code in a function body and then package it as a function, maybe in a library, or as a method as part of a class. The ease of doing this style of development is one aspect that makes Python such a joy use. And of course in the Jython implementation, it's easy to use this technique within the context of any Java library.

An important thing to keep in mind is that functions are first-class objects in Python. They can be passed around just like any other variable, resulting in some very powerful solutions. We'll see some examples of using functions in such a way later in this chapter.

Function Syntax and Basics

Functions are usually defined by using the 'def' keyword, the name of the function, its parameters (if any), and the body of code. We will start by looking at this example function:

Listing 4-1.

```
def times2(n):    return n * 2
```

In this example, the function name is times2 and it accepts a parameter n. The body of the function is only one line, but the work being done is the multiplication of the parameter by the number 2. Instead of storing the result in a variable, this function simply returns it to the calling code. An example of using this function would be as follows.

Listing 4-2.

```
>>> times2(8)16>>> x = times2(5)>>> x10
```

Normal usage can treat function definitions as being very simple. But there's subtle power in every piece of the function definition, due to the fact that Python is a dynamic language. We'll

look at these pieces from both a simple (the more typical case) and a more advanced perspective. We will also look at some alternative ways of creating functions in a later section.

The def Keyword

Using ‘def’ for define seems simple enough, and this keyword certainly can be used to declare a function just like you would in a static language. You should write most code that way in fact.

However, a function definition can occur at any level in your code and be introduced at any time. Unlike the case in a language like C or Java, function definitions are not declarations. Instead they are executable statements. You can nest functions, and we’ll describe that more when we talk about nested scopes. And you can do things like conditionally define them.

This means it’s perfectly valid to write code like the following:

Listing 4-3.

```
if variant:
    def f():
        print "One way"
else:
    def f():
        print "or another"
```

Please note, regardless of when and where the definition occurs, including its variants as above, the function definition will be compiled into a function object at the same time as the rest of the module or script that the function is defined in.

Naming the Function

We will describe this more in a later section, but the dir built-in function will tell us about the names defined in a given namespace, defaulting to the module, script, or console environment we are working in. With this new times2 function defined above, we now see the following (at least) in the console namespace:

Listing 4-4.

```
>>> dir()['__doc__', '__name__', 'times2']
```

We can also just look at what is bound to that name:

Listing 4-5.

```
>>> times2<function times2 at 0x1>
```

(This object is further introspectable. Try dir(times2) and go from there.) We can reference the function by supplying the function name such as we did in the example above. However, in order to call the function and make it perform some work, we need to supply the () to the end of the name.

We can also redefine a function at any time:

Listing 4-6.

```
>>> def f(): print "Hello, world"
...
>>> def f(): print "Hi, world"
...
>>> f()
Hi,world
```

This is true not just of running it from the console, but any module or script. The original version of the function object will persist until it's no longer referenced, at which point it will be ultimately be garbage collected. In this case, the only reference was the name f, so it became available for GC immediately upon rebind.

What's important here is that we simply rebound the name. First it pointed to one function object, then another. We can see that in action by simply setting another name (equivalently, a variable) to times2.

Listing 4-7.

```
>>> t2 = times2>>> t2(5)10
```

This makes passing a function as a parameter very easy, for a callback for example. A callback is a function that can be invoked by a function to perform a task and then turn around and invoke the calling function, thus the callback. Let's take a look at function parameters in more detail.

Function Metaprogramming

A given name can only be associated with one function at a time, so can't overload a function with multiple definitions. If you were to define two or more functions with the same name, the last one defined is used, as we saw.

However, it is possible to overload a function, or otherwise genericize it. You simply need to create a dispatcher function that then dispatches to your set of corresponding functions. Another way to genericize a function is to make use of the simplegeneric module which lets you define simple single-dispatch generic functions. For more information, please see the simplegeneric package in the Python Package Index.

Function Parameters and Calling Functions

When defining a function, you specify the parameters it takes. Typically you will see something like the following. The syntax is familiar:

```
def tip_calc(amt, pct)
```

As mentioned previously, calling functions is also done by placing parentheses after the function name. For example, for the function x with parameters a,b,c that would be x(a,b,c). Unlike some other dynamic languages like Ruby and Perl, the use of parentheses is required syntax (due the function name being just like any other name).

Objects are strongly typed, as we have seen. But function parameters, like names in general in Python, are not typed. This means that any parameter can refer to any type of object.

We see this play out in the times2 function. The * operator not only means multiply for numbers, it also means repeat for sequences (like strings and lists). So you can use the times2 function as follows:

Listing 4-8.

```
>>> times2(4)
8
>>> times2('abc')
'abcabc'
>>> times2([1,2,3])
[1, 2, 3, 1, 2, 3]
```

All parameters in Python are passed by reference. This is identical to how Java does it with object parameters. However, while Java does support passing unboxed primitive types by value, there are no such entities in Python. Everything is an object in Python. It is important to remember that immutable objects cannot be changed, and therefore, if we pass a string to a function and alter it, a copy of the string is made and the changes are applied to the copy.

Listing 4-9.

```
# The following function changes the text of a string by making a copy

# of the string and then altering it.  The original string is left

# untouched as it is immutable.

>>> def changestr(mystr):

...     mystr = mystr + '_changed'

...     print 'The string inside the function: ', mystr
```

```
...     return

>>> mystr = 'hello'

>>> changestr(mystr)

The string inside the function:  hello_changed

>>> mystr

'hello'
```

Functions are objects too, and they can be passed as parameters:

Listing 4-10.

```
# Define a function that takes two values and a mathematical function
>>> def perform_calc(value1, value2, func):
...     return func(value1, value2)
...
# Define a mathematical function to pass
>>> def mult_values(value1, value2):
...     return value1 * value2...
>>> perform_calc(2, 4, mult_values)
8
# Define another mathematical function to pass
>>> def add_values(value1, value2):
...     return value1 + value2
...
>>> perform_calc(2, 4, add_values)
6
>>>
```

If you have more than two or so arguments, it often makes more sense to call a function by named values, rather than by the positional parameters. This tends to create more robust code. So if you have a function draw_point(x,y), you might want to call it as draw_point(x=10,y=20).

Defaults further simplify calling a function. You use the form of param=default_value when defining the function. For instance, you might take our times2 function and generalize it.

Listing 4-11.

```
def times_by(n, by=2):    return n * by
```

This function is equivalent to times2 when called with just one argument—it uses the default value for the second argument by.

There's one point to remember that often trips up developers. The default value is initialized exactly once, when the function is defined. That's certainly fine for immutable values like

numbers, strings, tuples, frozensets, and similar objects. But you need to ensure that if the default value is mutable, that it's being used correctly. So a dictionary for a shared cache makes sense. But this mechanism won't work for a list where we expect it is initialized to an empty list upon invocation. If you're doing that, you need to write that explicitly in your code. As a best practice, use None as the default value rather than a mutable object, and check at the start of the body of your function for the case value = None and set the variable to your mutable object there.

Lastly, a function can take an unspecified number of ordered arguments, through *args, and keyword args, through **kwargs. These parameter names (args and kwargs) are conventional, so you can use whatever name makes sense for your function. The markers * and * are used to determine that this functionality should be used. The single * argument allows for passing a sequence of values, and a double ** argument allows for passing a dictionary of names and values. If either of these types of arguments is specified, they must follow any single arguments in the function declaration. Furthermore, the double ** must follow the single *.

Definition of a function that takes a sequence of numbers:

Listing 4-12.

```
def sum_args(*nums):
    return sum(nums)
```

Calling the function using a sequence of numbers:

```
>>> seq = [6,5,4,3]
>>> sum_args(*seq)
18
# we can also call the function without using the *
>>> sum_args(1,2,3,4)
10
```

Recursive Function Calls

It is also quite common to see cases in which a function calls itself from inside the function body. This type of function call is known as a recursive function call. Let's take a look at a function that computes the factorial of a given argument. This function calls itself passing in the provided argument decremented by 1 until the argument reaches the value of 0 or 1.

Listing 4-13.

```
def fact(n):
    if n in (0, 1):
        return 1
    else:
        return n * fact(n - 1)
```

It is important to note that Jython is like CPython in that it is ultimately stack based. Stacks are regions of memory where data is added and removed in a last-in first-out manner. If a recursive function calls itself too many times then it is possible to exhaust the stack, which results in an OutOfMemoryError. Therefore, be cautious when developing software using deep recursion.

Function Body

This section will break down the different components that comprise the body of a function. The body of a function is the part that performs the work. Throughout the next couple of sub-sections, you will see that a function body can be comprised of many different parts.

Documenting Functions

First, you should specify a document string for the function. The docstring, if it exists, is a string that occurs as the first value of the function body.

Listing 4-14.

```
def times2(n):
    """Given n, returns n * 2"""
    return n * 2
```

As mentioned in Chapter 1, by convention we use triple-quoted strings, even if your docstring is not multiline. If it is multiline, this is how we recommend you format it. For more information, please take a look at PEP 257 (www.python.org/dev/peps/pep-0257).

Listing 4-15.

```
def fact(n):
    """Returns the factorial of n
       Computes the factorial of n recursively. Does not check its
       arguments if nonnegative integer or if would stack
       overflow. Use with care!
    """
    if n in (0, 1):
        return 1
    else:
        return n * fact(n - 1)
```

Any such docstring, but with leading indentation stripped, becomes the __doc__ attribute of that function object. Incidentally, docstrings are also used for modules and classes, and they work exactly the same way.

You can now use the help built-in function to get the docstring, or see them from various IDEs like PyDev for Eclipse and nbPython for NetBeans as part of the auto-complete.

Listing 4-16.

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

fact(n)
    Returns the factorial of n
>>>
```

Returning Values

All functions return some value. In times2, we use the return statement to exit the function with that value. Functions can easily return multiple values at once by returning a tuple or other structure. The following is a simple example of a function that returns more than one value. In this case, the tip calculator returns the result of a tip based upon two percentage values.

Listing 4-17.

```
>>> def calc_tips(amount):
...     return (amount * .18), (amount * .20)
...
>>> calc_tips(25.25)
(4.545, 5.050000000000001)
```

A function can return at any time, and it can also return any object as its value. So you can have a function that looks like the following:

Listing 4-18.

```
>>> def check_pos_perform_calc(num1, num2, func):
...     if num1 > 0 and num2 > 0:
...         return func(num1, num2)
...     else:
...         return 'Only positive numbers can be used with this function!'
...
>>> def mult_values(value1, value2):
...     return value1 * value2
...
>>> check_pos_perform_calc(3, 4,mult_values)
12
>>> check_pos_perform_calc(3, -44, mult_values)
'Only positive numbers can be used with this function!'
```

If a return statement is not used, the value None is returned. There is no equivalent to a void method in Java, because every function in Python returns a value. However, the Python console will not show the return value when it's None, so you need to explicitly print it to see what is returned.

Listing 4-19.

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

Introducing Variables

A function introduces a scope for new names, such as variables. Any names that are created in the function are only visible within that scope. In the following example, the sq variable is defined within the scope of the function definition itself. If we try to use it outside of the function then we'll receive an error.

Listing 4-20.

```
>>> def square_num(num):
...     """ Return the square of a number"""
...      sq = num * num
...      return sq
...
>>> square_num(35)
1225
>>> sq
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError:  name 'sq' is not defined
```

Global Variables

The global keyword is used to declare that a variable name is from the module scope (or script) containing this function. Using global is rarely necessary in practice, since it is not necessary if the name is called as a function or an attribute is accessed (through dotted notation).

This is a good example of where Python is providing a complex balancing between a complex idea—the lexical scoping of names, and the operations on them—and the fact that in practice it is doing the right thing.

Here is an example of using a global variable in the same square_num() function.

Listing 4-21.

```
>>> sq = 0
>>> def square_num(n):
...      global sq
...      sq = n * n
...      return sq
...
>>> square_num(10)
100
>>> sq
100
```

Other Statements

What can go in a function body? Pretty much any statement, including material that we will cover later in this book. So you can define functions or classes or use even import, within the scope of that function.

In particular, performing a potentially expensive operation like import as least as possible, can reduce the startup time of your app. It's even possible it will be never needed too.

There are a couple of exceptions to this rule. In both cases, these statements must go at the beginning of a module, similar to what we see in a static language like Java:

Compiler directives. Python supports a limited set of compiler directives that have the provocative syntax of from __future__ import X; see PEP 236. These are features that will eventually be made available, generally in the next minor revision (such as 2.5 to 2.6). In addition, it's a popular place to put Easter eggs, such as from __future__ import braces. (Try it in the console, which also relaxes what it means to be performed at the beginning.)

Source encoding declaration. Although technically not a statement—it's in a specially parsed comment—this must go in the first or second line.

Empty Functions

It is also possible to define an empty function. Why have a function that does nothing? As in math, it's useful to have an operation that stands for doing nothing, like "add zero" or "multiply by one." These identity functions eliminate special cases. Likewise, as see with empty_callback, we may need to specify a callback function when calling an API, but nothing actually needs to be done. By passing in an empty function—or having this be the default—we can simplify the API. An empty function still needs something in its body. You can use the pass statement.

Listing 4-22.

```
def do_nothing():
    pass # here's how to specify an empty body of code
Or you can just have a docstring for the function body as in the following
example.
def empty_callback(*args, **kwargs):
    """Use this function where we need to supply a callback,
       but have nothing further to do.
    """
```

Miscellaneous Information for the Curious Reader

As you already know, Jython is an interpreted language. That is, the Python code that we write for a Jython application is ultimately compiled down into Java bytecode when our program is run. So oftentimes it is useful for Jython developers to understand what is going on when this

code is interpreted into Java bytecode. What do functions look like from Java? They are instances of an object named PyObject, supporting the __call__ method. Additional introspection is available. If a function object is just a standard function written in Python, it will be of class PyFunction. A built-in function will be of class PyBuiltinFunction. But don't assume that in your code, because many other objects support the function interface (__call__), and these potentially could be proxying, perhaps several layers deep, a given function. You can only assume it's a PyObject. Much more information is available by going to the Jython wiki. You can also send questions to the jython-dev mailing list for more specifics.

Built-in Functions

Built-in functions are those functions that are always in the Python namespace. In other words, these functions—and built-in exceptions, boolean values, and some other objects—are the only truly globally defined names. If you are familiar with Java, they are somewhat like the classes from java.lang. Built-ins are rarely sufficient, however; even a simple command line script generally needs to parse its arguments or read in from its standard input. So for this case you would need to import sys. And in the context of Jython, you will need to import the relevant Java classes you are using, perhaps with import java. But the built-in functions are really the core function that almost all Python code uses. The documentation for covering all of the built-in functions that are available is extensive. However, it has been included in this book as Appendix C. It should be easy to use Appendix C as a reference when using a built-in function, or for choosing which built-in function to use.

Alternative Ways to Define Functions

The 'def' keyword is not the only way to define a function. Here are some alternatives:

- Lambda Functions: 'lambda' functions. The 'lambda' keyword creates an unnamed function. Some people like this because it requires minimal space, especially when used in a callback.
- Classes: In addition, we can also create objects with classes whose instance objects look like ordinary functions. Objects supporting the __call__ protocol. For Java developers, this is familiar. Classes implement such single-method interfaces as Callable or Runnable.
- Bound Methods: Instead of calling x.a(), I can pass x.a as a parameter or bind to another name. Then I can invoke this name. The first parameter of the method will be passed the bound object, which in OO terms is the receiver of the method. This is a simple way of creating callbacks. (In Java you would have just passed the object of course, then having the callback invoke the appropriate method such as call or run.)

Lambda Functions

As stated in the introduction, a lambda function is an anonymous function. In other words, a lambda function is not required to be bound to any name. This can be useful when you are trying to create compact code or when it does not make sense to declare a named function

because it will only be used once. A lambda function is usually written inline with other code, and most often the body of a lambda function is very short in nature. A lambda function is comprised of the following segments: lambda <<argument(s)>> : <<function body>> A lambda function accepts arguments just like any other function, and it uses those arguments within its function body. Also, just like other functions in Python a value is always returned. Let's take a look at a simple lambda function to get a better understanding of how they work.

Listing 4-23. Example of using a lambda function to combine two strings. In this case, a first and last name

```
>>> name_combo = lambda first,last: first + ' ' + last
>>> name_combo('Jim','Baker')
'Jim Baker'
```

In the example above, we assigned the function to a name. However, a lambda function can also be defined in-line with other code. Oftentimes a lambda function is used within the context of other functions, namely built-ins.

Generator Functions

Generators are special functions that are an example of iterators, which will be discussed in Chapter 6. Generators advance to the next point by calling the special method next. Usually that's done implicitly, typically through a loop or a consuming function that accepts iterators, including generators. They return values by using the yield statement. Each time a yield statement is encountered then the current iteration halts and a value is returned. Generators have the ability to remember where they left off. Each time next() is called, the generator resumes where it had left off. A StopIteration error will be raised once the generator has been terminated. Over the next couple of sections, we will take a closer look at generators and how they work. Along the way, you will see many examples for creating and using generators.

Defining Generators

A generator function is written so that it consists of one or more yield points, which are marked through the use of the yield statement. As mentioned previously, each time the yield statement is encountered, a value is returned. *Listing 4-24.*

```
def g():
    print "before yield point 1"
    # The generator will return a value once it encounters the yield
statement
    yield 1
    print "after 1, before 2"
    yield 2
    yield 3
```

In the previous example, the generator function g() will halt and return a value once the first yield statement is encountered. In this case, a 1 will be returned. The next time g.next() is called, the generator will continue until it encounters the next yield statement. At that point it will return another value, the 2 in this case. Let's see this generator in action. Note that calling the generator function simply creates your generator, it does not cause any yields. In order to get the value from the first yield, we must call next().

Listing 4-25.

```
# Call the function to create the generator
>>> x = g()
# Call next() to get the value from the yield
>>> x.next()
before the yield point 1
1
>>> x.next()
after 1, before 2
2
>>> x.next()
3
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Let's take a look at another more useful example of a generator. In the following example, the step_to() function is a generator that increments based upon a given factor. The generator starts at zero and increments each time next() is called. It will stop working once it reaches the value that is provided by the stop argument.

Listing 4-26.

```
>>> def step_to(factor, stop):
...     step = factor
...     start = 0
...     while start <= stop:
...         yield start
...         start += step
...
>>> for x in step_to(1, 10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
```

```
10
>>> for x in step_to(2, 10):
...     print x
...
0
2
4
6
8
10
>>>
```

If the yield statement is seen in the scope of a function, then that function is compiled as if it's a generator function. Unlike other functions, you use the return statement only to say, "I'm done," that is, to exit the generator, and not to return any values. You can think of return as acting like a break in a for-loop or while-loop. Let's change the step_to function just a bit to check and ensure that the factor is less than the stopping point. We'll add a return statement to exit the generator if the factor is greater or equal to the stop.

Listing 4-27

```
>>> def step_return(factor, stop):
...     step = factor
...     start = 0
...     if factor >= stop:
...         return
...     while start <= stop:
...         yield start
...         start += step
...
>>> for x in step_return(1,10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
10
>>> for x in step_return(3,10):
...     print x
...
0
3
6
9
>>> for x in step_return(3,3):
...     print x
```

```
...
```

If you attempt to return an argument then a syntax error will be raised.

Listing 4-28.

```
def g():
    yield 1
    yield 2
    return None

for i in g():
    print i

SyntaxError: 'return' with argument inside generator
```

Many useful generators actually will have an infinite loop around their yield expression, instead of ever exiting, explicitly or not. The generator will essentially work each time next() is called throughout the life of the program.

Listing 4-29. Pseudocode for generator using infinite loop

```
while True:
    yield stuff
```

This works because a generator object can be garbage collected as soon as the last reference to the generator is used. The fact that it uses the machinery of function objects to implement itself doesn't matter.

Generator Expressions

Generator expressions are an alternative way to create the generator object. Please note that this is not the same as a generator function! It's the equivalent to what a generator function yields when called. Generator expressions basically create an unnamed generator.

Listing 4-30.

```
>>> x = (2 * x for x in [1,2,3,4])
>>> x
<generator object at 0x1>
>>> x()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not callable
Let's see this generator expression in action:
>>> for v in x:
...     print v
...
```

```
2
4
6
8
>>>
```

Typically generator expressions tend to be more compact but less versatile than generator functions. They are useful for getting things done in a concise manner.

Namespaces, Nested Scopes, and Closures

Note that you can introduce other namespaces into your function definition. It is possible to include import statements directly within the body of a function. This allows such imports to be valid only within the context of the function. For instance, in the following function definition the imports of A and B are only valid within the context of f().

Listing 4-31.

```
def f():
    from NS import A, B
```

At first glance, including import statements within your function definitions may seem unnecessary. However, if you think of a function as an object then it makes much more sense. We can pass functions around just like other objects in Python such as variables. As mentioned previously, functions can even be passed to other functions as arguments. Function namespaces provide the ability to treat functions as their own separate piece of code. Oftentimes, functions that are used in several different places throughout an application are stored in a separate module. The module is then imported into the program where needed. Functions can also be nested within each other to create useful solutions. Since functions have their own namespace, any function that is defined within another function is only valid within the parent function. Let's take a look at a simple example of this before we go any further.

Listing 4-32.

```
>>> def parent_function():
...     x = [0]
...     def child_function():
...         x[0] += 1
...         return x[0]
...     return child_function
…
>>> p = parent_function()
>>> p()
1
>>> p()
2
>>> p()
3
```

```
>>> p()
4
```

While this example is not extremely useful, it allows you to understand a few of the concepts for nesting functions. As you can see, the parent_function contains a function named child_function. The parent_function in this example returns the child_function. What we have created in this example is a simple Closure function. Each time the function is called, it executes the inner function and increments the variable x which is only available within the scope of this closure. In the context of Jython, using closures such as the one defined previously can be useful for integrating Java concepts as well. It is possible to import Java classes into the scope of your function just as it is possible to work with other Python modules. It is sometimes useful to import in a function call in order to avoid circular imports, which is the case when function A imports function B, which in turn contains an import to function A. By specifying an import in a function call you are only using the import where it is needed. You will learn more about using Java within Jython in Chapter 10.

Function Decorators

Decorators are a convenient syntax that describes a way to transform a function. They are essentially a metaprogramming technique that enhances the action of the function that they decorate. To program a function decorator, a function that has already been defined can be used to decorate another function, which basically allows the decorated function to be passed into the function that is named in the decorator. Let's look at a simple example.

Listing 4-33.

```
def plus_five(func):
    x = func()
    return x + 5

@plus_five
def add_nums():
    return 1 + 2
```

In this example, the add_nums() function is decorated with the plus_five() function. This has the same effect as passing the add_nums function into the plus_five function. In other words, this decorator is syntactic sugar that makes this technique easier to use. The decorator above has the same functionality as the following code.

Listing 4-34.

```
add_nums = plus_five(add_nums)
```

In actuality, add_nums is now no longer a function, but rather an integer. After decorating with plus_five you can no longer call add_nums(), we can only reference it as if it were an integer. As you can see, add_nums is being passed to plus_five at import time. Normally, we'd want to

have add_nums finish up as a function so that it is still callable. In order to make this example more useful, we'll want to make add_nums callable again and we will also want the ability to change the numbers that are added. To do so, we need to rewrite the decorator function a bit so that it includes an inner function that accepts arguments from the decorated function.

Listing 4-35.

```
def plus_five(func):
    def inner(*args, **kwargs):
        x = func(*args, **kwargs) + 5
        return x
    return inner

@plus_five
def add_nums(num1, num2):
    return num1 + num2
```

Now we can call the add_nums() function once again and we can also pass two arguments to it. Because it is decorated with the plus_five function it will be passed to it and then the two arguments will be added together and the number five will be added to that sum. The result will then be returned.

Listing 4-36.

```
>>> add_nums(2,3)
10
>>> add_nums(2,6)
13
```

Now that we've covered the basics of function decorators it is time to take a look at a more in-depth example of the concept. In the following decorator function example, we are taking a twist on the old tip_calculator function and adding a sales tax calculation. As you see, the original calc_bill function takes a sequence of amounts, namely the amounts for each item on the bill. The calc_bill function then simply sums the amounts and returns the value. In the given example, we apply the sales_tax decorator to the function which then transforms the function so that it not only calculates and returns the sum of all amounts on the bill, but it also applies a standard sales tax to the bill and returns the tax amount and total amounts as well.

Listing 4-37.

```
def sales_tax(func):
    ''' Applies a sales tax to a given bill calculator '''
    def calc_tax(*args, **kwargs):
        f = func(*args, **kwargs)
        tax = f * .18
        print "Total before tax: $ %.2f" % (f)
        print "Tax Amount: $ %.2f" % (tax)
        print "Total bill: $ %.2f" % (f + tax)
```

```
    return calc_tax

@sales_tax
def calc_bill(amounts):
    ''' Takes a sequence of amounts and returns sum '''
    return sum(amounts)
```

The decorator function contains an inner function that accepts two arguments, a sequence of arguments and a dictionary of keyword args. We must pass these arguments to our original function when calling from the decorator to ensure that the arguments that we passed to the original function are applied within the decorator function as well. In this case, we want to pass a sequence of amounts to calc_bill, so passing the *args, and **kwargs arguments to the function ensures that our amounts sequence is passed within the decorator. The decorator function then performs simple calculations for the tax and total dollar amounts and prints the results. Let's see this in action:

Listing 4-38.

```
>>> amounts = [12.95,14.57,9.96]
>>> calc_bill(amounts)
Total before tax: $ 37.48
Tax Amount: $ 6.75
Total bill: $ 44.23
```

It is also possible to pass arguments to decorator functions when doing the decorating. In order to do so, we must nest another function within our decorator function. The outer function will accept the arguments to be passed into the decorator function, the inner function will accept the decorated function, and the inner most function will perform the work. We'll take another spin on the tip calculator example and create a decorator that will apply the tip calculation to the calc_bill function.

Listing 4-39.

```
def tip_amount(tip_pct):
    def calc_tip_wrapper(func):
        def calc_tip_impl(*args, **kwargs):
            f = func(*args, **kwargs)
            print "Total bill before tip: $ %.2f" % (f)
            print "Tip amount: $ %.2f" % (f * tip_pct)
            print "Total with tip: $ %.2f" % (f + (f * tip_pct))
        return calc_tip_impl
    return calc_tip_wrapper
```

Now let's see this decorator function in action. As you'll notice, we pass a percentage amount to the decorator itself and it is applied to the decorator function.

Listing 4-40.

```
>>> @tip_amount(.18)
... def calc_bill(amounts):
...     ''' Takes a sequence of amounts and returns sum '''
...     return sum(amounts)
...
>>> amounts = [20.95, 3.25, 10.75]
>>> calc_bill(amounts)
Total bill before tip: $ 34.95
Tip amount: $ 6.29
Total with tip: $ 41.24
```

As you can see, we have a similar result as was produced with the sales tax calculator, except that with this decorator solution we can now vary the tip percentage. All of the amounts in the sequence of amounts are summed up and then the tip is applied. Let's take a quick look at what is actually going on if we do not use the decorator @ syntax.

Listing 4-41.

```
calc_bill = tip_amount(.18)(calc_bill)
```

At import time, the tip_amount() function takes both the tip percentage and the calc_bill function as arguments, and the result becomes the new calc_bill function. By including the decorator, we're actually decorating calc_bill with the function which is returned by tip_amount(.18). In the larger scale of the things, if we applied this decorator solution to a complete application then we could accept the tip percentage from the keyboard and pass it into the decorator as we've shown in the example. The tip amount would then become a variable that can fluctuate based upon a different situation. Lastly, if we were dealing with a more complex decorator function, we have the ability to change the inner-working of the function without adjusting the original decorated function at all. Decorators are an easy way to make our code more versatile and manageable.

Coroutines

Coroutines are often compared to generator functions in that they also make use of the yield statement. However, a coroutine is exactly the opposite of a generator in terms of functionality. A coroutine actually treats a yield statement as an expression, and it accepts data instead of returning it. Coroutines are oftentimes overlooked as they may at first seem like a daunting topic. However, once it is understood that coroutines and generators are not the same thing then the concept of how they work is a bit easier to grasp. A coroutine is a function that receives data and does something with it. We will take a look at a simple coroutine example and then break it down to study the functionality.

Listing 4-42.

```
def co_example(name):
    print 'Entering coroutine %s' % (name)
    my_text = []
```

```
    while True:
        txt = (yield)
        my_text.append(txt)
        print my_text
```

Here we have a very simplistic coroutine example. It accepts a value as the “name” of the coroutine. It then accepts strings of text, and each time a string of text is sent to the coroutine, it is appended to a list. The yield statement is the point where text is being entered by the user. It is assigned to the txt variable and then processing continues. It is important to note that the my_text list is held in memory throughout the life of the coroutine. This allows us to append values to the list with each yield. Let’s take a look at how to actually use the coroutine.

Listing 4-43.

```
>>> ex = co_example("example1")
>>> ex.next()
Entering coroutine example1
```

In this code, we assign the name “example1” to this coroutine. We could actually accept any type of argument for the coroutine and do whatever we want with it. We’ll see a better example after we understand how this works. Moreover, we could assign this coroutine to multiple variables of different names and each would then be its own coroutine object that would function independently of the others. The next line of code calls next() on the function. The next() must be called once to initialize the coroutine. Once this has been done, the function is ready to accept values.

Listing 4-44.

```
>>> ex.send("test1")
['test1']
>>> ex.send("test2")
['test1', 'test2']
>>> ex.send("test3")
['test1', 'test2', 'test3']
```

As you can see, we use the send() method to actually send data values into the coroutine. In the function itself, the text we send is inserted where the (yield) expression is placed. We can really continue to use the coroutine forever, or until our JVM is out of memory. However, it is a best practice to close() the coroutine once it is no longer needed. The close() call will cause the coroutine to be garbage collected.

Listing 4-45.

```
>>> ex.close()
>>> ex.send("test1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
StopIteration
```

If we try to send more data to the function once it has been closed then a StopIteration error is raised. Coroutines can be very helpful in a number of situations. While the previous example doesn't do much, there are a number of great applications to which we can apply the use of coroutines and we will see a more useful example in a later section.

Decorators in Coroutines

While the initialization of a coroutine by calling the next() method is not difficult to do, we can eliminate this step to help make things even easier. By applying a decorator function to our coroutine, we can automatically initialize it so it is ready to receive data. Let's define a decorator that we can apply to the coroutine in order to make the call to next().

Listing 4-46.

```
def coroutine_next(f):
    def initialize(*args,**kwargs):
        coroutine = f(*args,**kwargs)
        coroutine.next()
        return coroutine
    return initialize
```

Now we will apply our decorator to the coroutine function and then make use of it.

```
>>> @coroutine_next
... def co_example(name):
...     print 'Entering coroutine %s' % (name)
...     my_text = []
...     while True:
...         txt = (yield)
...         my_text.append(txt)
...         print my_text
...
>>> ex2 = co_example("example2")
Entering coroutine example2
>>> ex2.send("one")
['one']
>>> ex2.send("two")
['one', 'two']
>>> ex2.close()
```

As you can see, while it is not necessary to use a decorator for performing such tasks, it definitely makes things easier to use. If we chose not to use the syntactic sugar of the @ syntax, we could do the following to initialize our coroutine with the coroutine_next() function.

Listing 4-47.

```
co_example = coroutine_next(co_example)
```

Coroutine Example

Now that we understand how coroutines are used, let's take a look at a more in-depth example. Hopefully after reviewing this example you will understand how useful such functionality can be. In this example, we will pass the name of a file to the coroutine on initialization. After that, we will send strings of text to the function and it will open the text file that we sent to it (given that the file resides in the correct location), and search for the number of matches per a given word. The numeric result for the number of matches will be returned to the user.

Listing 4-48.

```
def search_file(filename):
    print 'Searching file %s' % (filename)
    my_file = open(filename, 'r')
    file_content = my_file.read()
    my_file.close()
    while True:
        search_text = (yield)
        search_result = file_content.count(search_text)
        print 'Number of matches: %d' % (search_result)
```

The coroutine above opens the given file, reads its content, and then searches and returns the number of matches for any given send call.

Listing 4-49.

```
>>> search = search_file("example4_3.txt")
>>> search.next()
Searching file example4_3.txt
>>> search.send('python')
Number of matches: 0
>>> search.send('Jython')
Number of matches: 1
>>> search.send('the')
Number of matches: 4
>>> search.send('This')
Number of matches: 2
>>> search.close();
```

Summary

In this chapter, we have covered the use of functions in the Python language. There are many different use-cases for functions and we have learned techniques that will allow us to apply the functions to many situations. Functions are first-class objects in Python, and they can be treated as any other object. We started this chapter by learning the basics of how to define a function. After learning about the basics, we began to evolve our knowledge of functions by learning how

to use parameters and make recursive function calls. There are a wide variety of built-in functions available for use. If you take a look at Appendix C of this book you can see a listing of these built-ins. It is a good idea to become familiar with what built-ins are available. After all, it doesn't make much sense to rewrite something that has already been written. This chapter also discussed some alternative ways to define functions including the lambda notation, as well as some alternative types of functions including decorators, generators and coroutines. Wrapping up this chapter, you should now be familiar with Python functions and how to create and use them. You should also be familiar with some of the advanced techniques that can be applied to functions.

Source: http://www.jython.org/jythonbook/en/1.0/DefiningFunctionsandUsingBuilt-Ins.html