BONUS CHAPTER 45

Using Python

IN THIS CHAPTER

- Python on Linux
- The Basics of Python
- Functions
- Object Orientation
- The Standard Library and the Python Package Index
- References

As PHP has come to dominate the world of web scripting, Python is increasingly dominating the domain of command-line scripting. Python's precise and clean syntax makes it one of the easiest languages to learn, and it allows programmers to code more quickly and spend less time maintaining their code. Although PHP is fundamentally similar to Java and Perl, Python is closer to C and Modula-3, and so it might look unfamiliar at first.

Most of the other languages have a group of developers at their cores, but Python has Guido van Rossum—creator, father, and *benevolent dictator for life (BDFL)*. Although Guido spends less time working on Python now, he still essentially has the right to veto changes to the language, which has enabled it to remain consistent over the many years of its development. The end result is that, in Guido's own words, "Even if you are in fact clueless about language design, you can tell that Python is a very simple language."

This chapter constitutes a "quick-start" tutorial to Python designed to give you all the information you need to put together basic scripts and to point you toward resources that can take you further.

Two supported versions of Python are out in the wild right now. Python version 2.x is the backward-compatible status quo, the version that anyone who has already used Python for more than a few years has learned and the version most likely to have been used in most already written code you discover and read. Python 3.x is the newest branch with lots of shiny newness.

Python 3.x is not backward compatible, and most programs written in or for 2.x need some work to run on 3.x.

Python: Do you want to run your program on many machines, perhaps including some that have only the older version installed? Does the newer version have all the functionality you want and need, including software libraries, because not all of them have been rewritten for 3.x (like Twisted and Django)? If porting is a nontrivial task, you might want to stick with 2.x for now.

Both versions will be supported for some time, but if you expect you will be using what you write for longer than a couple months, you probably want to try to do so in 3.x as it has matured and is the version that is being promoted by all, even though 2.x is still supported. The official Python website helps you make an informed decision; it contains clear discussion of the advantages and disadvantages of each version at http://wiki.python.org/moin/Python2orPython3.

This chapter has information about the 2.x version of Python, and we try to note places where significant differences exist in 3.x. If you are learning Python for the first time, start with a look at the 3.x series because it is the future and is very likely to already have everything you need.

NOTE

This chapter is a very elementary introduction to Python. For that reason, nearly everything we cover is identical in both 2.x and 3.x. We mention the differences earlier and in a couple of notes that follow so that readers wanting to learn more will not be surprised later. Unless noted otherwise, you should find what we say in this chapter applies as well to either version.

Python on Linux

Most versions of Linux and UNIX, including Mac OS X, come with Python preinstalled. This is partially for convenience because it is such a popular scripting language—and it saves having to install it later if the user wants to run a script—and partially because many vital or useful programs included in Linux distributions are written in Python. For example, the Ubuntu Software Center is a Python program.

The Python binary is installed into `/usr/bin/python` (or `/usr/bin/python3`); if you run that, you enter the Python interactive interpreter where you can type commands and have them executed immediately. Although PHP also has an interactive mode (use `php -a` to activate it), it is neither as powerful nor as flexible as Python's.

As with Perl, PHP, and other scripting languages, you can also execute Python scripts by adding a shebang line (`#!`) to the start of your scripts that point to `/usr/bin/python` and then setting the file to be executable.

The third way to run Python scripts is through `mod_python`, which is an Apache module that embeds the Python interpreter into the http server, allowing you to use Python to write web applications. You can install this from the Ubuntu repositories.

We use the interactive Python interpreter for this chapter because it provides immediate feedback on commands as you type them, so it is essential that you become comfortable using it. To get started, open a terminal and run the command `python`. You should see something like this, perhaps with different version numbers and dates:

```
matthew@seymour:~$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:13:53)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
The >>> is where you type your input, and you can set and get a variable like this:
>>> python = 'great'
>>> python'great'
>>>
```

On line one, the variable `python` is set to the text great, and on line two that value is read back from the variable simply by typing the name of the variable you want to read. Line three shows Python printing the variable; on line four you are back at the prompt to type more commands. Python remembers all the variables you use while in the interactive interpreter, which means you can set a variable to be the value of another variable.

When you are done, press Ctrl+D to exit. At this point, all your variables and commands are forgotten by the interpreter, which is why complex Python programs are always saved in scripts.

The Basics of Python

Python is a language wholly unlike most others, and yet it is so logical that most people can pick it up quickly. You have already seen how easily you can assign strings, but in Python nearly everything is that easy—as long as you remember the syntax.

Numbers

The way Python handles numbers is more precise than some other languages. It has all the normal operators—such as `+` for addition, `-` for subtraction, `/` for division, and `*` for multiplication—but it adds `%` for modulus (division remainder), `**` for raise to the power, and `//` for floor division. It is also specific about which type of number is being used, as this example shows:

```
>>> a = 5
>>> b = 10
>>> a * b
50
>>> a / b
0
>>> b = 10.0
>>> a / b
```

```
0.5
>>> a // b
0.0
```

The first division returns `0` because both `a` and `b` are integers (whole numbers), so Python calculates the division as an integer, giving `0`. By converting `b` to `10.0`, Python considers it to be a floating-point number, and so the division is now calculated as a floating-point value, giving `0.5`. Even with `b` being a floating point, using `//`—floor division—rounds it down.

Using `**`, you can easily see how Python works with integers:

```
>>> 2 ** 30
1073741824
>>> 2 ** 31
2147483648L
```

The first statement raises `2` to the power of `30` (that is, 2 times 2 times 2 times 2...), and the second raises `2` to the power of `31`. Notice how the second number has a capital `L` on the end of it—this is Python telling you that it is a long integer. The difference between long integers and normal integers is slight but important: Normal integers can be calculated using simple instructions on the CPU, whereas long integers—because they can be as big as you need them to be—need to be calculated in software and therefore are slower.

When specifying big numbers, you need not put the `L` at the end—Python figures it out for you. Furthermore, if a number starts off as a normal number and then exceeds its boundaries, Python automatically converts it to a long integer. The same is not true the other way around: If you have a long integer and then divide it by another number so that it could be stored as a normal integer, it remains a long integer:

```
>>> num = 999999999999999999999999999999999L
>>> num = num / 10000000000000000000000000000000
>>> num
999L
```

You can convert between number types using typecasting, like this:

```
>>> num = 10
>>> int(num)
10
>>> float(num)
10.0
>>> long(num)
10L
>>> floatnum = 10.0
>>> int(floatnum)
10
>>> float(floatnum)
```

```
10.0
>>> long(floatnum)
10L
```

You need not worry whether you are using integers or long integers; Python handles it all for you, so you can concentrate on getting the code right. In the 3.x series, the Python integer type automatically provides extra precision for large numbers, whereas in 2.x, you use a separate long integer type for numbers larger than the normal integer type is designed to handle.

More on Strings

Python stores strings as an immutable sequence of characters—a jargon-filled way of saying that "it is a collection of characters that, once set, cannot be changed without creating a new string." Sequences are important in Python. There are three primary types, of which strings are one, and they share some properties. Mutability makes a lot of sense when you learn about lists in the next section.

As you saw in the previous example, you can assign a value to strings in Python with just an equal sign, like this:

```
>>> mystring = 'hello';
>>> myotherstring = "goodbye";
>>> mystring'hello'
>>> myotherstring;'goodbye'
>>> test = "Are you really Jayne Cobb?"
>>> test
"Are you really Jayne Cobb?"
```

The first example encapsulates the string in single quotation marks, and the second and third in double quotation marks. However, printing the first and second strings shows them both in single quotation marks because Python does not distinguish between the two. The third example is the exception; it uses double quotation marks because the string itself contains a single quotation mark. Here, Python prints the string with double quotation marks because it knows it contains the single quotation mark.

Because the characters in a string are stored in sequence, you can index into them by specifying the character you are interested in. Like most other languages, these indexes are zero based, which means you need to ask for character `0` to get the first letter in a string. For example:

```
>>> string = "This is a test string"
>>> string
'This is a test string'
>>> string[0]
'T'
>>> string [0], string[3], string [20]
('T', 's', 'g')
```

The last line shows how, with commas, you can ask for several indexes at the same time. You could print the entire first word using this:

```
>>> string[0], string[1], string[2], string[3]
('T', 'h', 'I', 's')
```

However, for that purpose, you can use a different concept: slicing. A *slice* of a sequence draws a selection of indexes. For example, you can pull out the first word like this:

```
>>> string[0:4]
'This'
```

The syntax there means "take everything from position 0 (including 0) and end at position 4 (excluding it)." So, `[0:4]` copies the items at indexes `0`, `1`, `2`, and `3`. You can omit either side of the indexes, and it will copy either from the start or to the end:

```
>>> string [:4]
'This'
>>> string [5:]
'is a test string'
>>> string [11:]
'est string'
```

You can also omit both numbers, and it gives you the entire sequence:

```
>>> string [:]'This is a test string'
```

Later you learn precisely why you would want to do that, but for now there are a number of other string intrinsics that will make your life easier. For example, you can use the `+` and `*` operators to concatenate (join) and repeat strings, like this:

```
>>> mystring = "Python"
>>> mystring * 4
'PythonPythonPythonPython'
>>> mystring = mystring + " rocks! "
>>> mystring * 2
'Python rocks! Python rocks! '
```

In addition to working with operators, Python strings come with a selection of built-in methods. You can change the case of the letters with `capitalize()` (uppercases the first letter and lowercases the rest), `lower()` (lowercases them all), `title()` (uppercases the first letter in each word), and `upper()` (uppercases them all). You can also check whether strings match certain cases with `islower()`, `istitle()`, and `isupper()`; that also extends to `isalnum()` (returns true if the string is letters and numbers only) and `isdigit()` (returns true if the string is all numbers).

This example demonstrates some of these in action:

```
>>> string
'This is a test string'
>>> string.upper()
'THIS IS A TEST STRING'
>>> string.lower()
'this is a test string'
>>> string.isalnum()
False
>>> string = string.title()
>>> string
'This Is A Test String'
```

Why did `isalnum()` return `false`—our string contains only alphanumeric characters, doesn't it? Well, no. There are spaces in there, which is what is causing the problem. More important, we were calling `upper()` and `lower()` and they were not changing the contents of the string—they just returned the new value. So, to change our string from `This is a test string` to `This Is A Test String`, we have to assign it back to the string variable.

Another really useful and kind of cool thing you can do with strings is triple quoting. This lets you easily create strings that include both double and single quoted strings, as well as strings that span more than one line, without having to escape a bunch of special characters. Here's an example:

```
>>>foo = """
..."foo"
...'bar'
...baz
...blippy
..."""
>>>foo
'\n"foo"\n\'bar\'\nbaz\nblippy\n'
>>>
```

Although this is intended as an introductory guide, the capability to use negative indexes with strings and slices is a pretty neat feature that deserves a mention. Here's an example:

```
>>>bar="news.google.com"
>>>bar[-1]
'm'
>>>bar[-4:]
'.com'
>>>bar[-10:-4]
'google'
>>>
```

One difference between 2.x and 3.x is the introduction of new string types and method calls. That doesn't affect what we share here, but is something to pay attention to as you work with Python.

Lists

Python's built-in list data type is a sequence, like strings. However, they are mutable, which means you can change them. Lists are like arrays in that they hold a selection of elements in a given order. You can cycle through them, index into them, and slice them:

```
>>> mylist = ["python", "perl", "php"]
>>> mylist
['python', 'perl', 'php']
>>> mylist + ["java"]
["python", 'perl', 'php', 'java']
>>> mylist * 2
['python', 'perl', 'php', 'python', 'perl', 'php']
>>> mylist[1]
'perl'
>>> mylist[1] = "c++"
>>> mylist[1]
'c++'
>>> mylist[1:3]
['c++', 'php']
```

The brackets notation is important: You cannot use parentheses, (`(` and `)`) or braces ({ and }) for lists. Using `+` for lists is different from using `+` for numbers. Python detects that you are working with a list and appends one list to another. This is known as *operator overloading*, and it is one of the reasons Python is so flexible.

Lists can be *nested*, which means you can put a list inside a list. However, this is where mutability starts to matter, and so this might sound complicated. If you recall, the definition of an immutable string sequence is that it is a collection of characters that, once set, cannot be changed without creating a new string. Lists are mutable, as opposed to immutable, which means you can change your list without creating a new list.

This becomes important because Python, by default, copies only a reference to a variable rather than the full variable. For example:

```
>>> list1 = [1, 2, 3]
>>> list2 = [4, list1, 6]
>>> list1
[1, 2, 3]
>>> list2
[4, [1, 2, 3], 6]
```

Here we have a nested list. `list2` contains `4`, then `list1`, then `6`. When you print the value of `list2`, you can see it also contains `list1`. Now, let's proceed on from that:

```
>>> list1[1] = "Flake"
>>> list2
[4, [1, 'Flake', 3], 6]
```

In line one, we set the second element in `list1` (remember, sequences are zero based!) to be `Flake` rather than `2`; then we print the contents of `list2`. As you can see, when `list1` changed, `list2` was updated, too. The reason for this is that `list2` stores a reference to `list1` as opposed to a copy of `list1`; they share the same value.

We can show that this works both ways by indexing twice into `list2`, like this:

```
>>> list2[1][1] = "Caramello"
>>> list1
[1, 'Caramello', 3]
```

The first line says, "Get the second element in `list2 (list1)` and the second element of that list, and set it to be 'Caramello'." Then `list1`'s value is printed, and you can see it has changed. This is the essence of mutability: We are changing our list without creating a new list. However, editing a string creates a new string, leaving the old one unaltered. For example:

```
>>> mystring = "hello"
>>> list3 = [1, mystring, 3]
>>> list3
[1, 'hello', 3]
>>> mystring = "world"
>>> list3
[1, 'hello', 3]
```

Of course, this raises the question of how you copy without references when references are the default. The answer, for lists, is that you use the `[:]` slice, which are covered earlier in this chapter. This slices from the first element to the last, inclusive, essentially copying it without references. Here is how that looks:

```
>>> list4 = ["a", "b", "c"]
>>> list5 = list4[:]
>>> list4 = list4 + ["d"]
>>> list5
['a', 'b', 'c']
>>> list4
['a', 'b', 'c', 'd']
```

Lists have their own collections of built-in methods, such as `sort()`, `append()`, and `pop()`. The latter two add and remove single elements from the end of the list, with `pop()` also returning the removed element. For example:

```
>>> list5 = ["nick", "paul", "Julian", "graham"]
>>> list5.sort()
```

```
>>> list5
['graham', 'julian', 'nick', 'paul'']
>>> list5.pop()'paul'
>>> list5
['graham', 'julian', 'nick']

>>> list5.append("Rebecca")
```

In addition, one interesting method of strings returns a list: `split()`. This takes a character to split by and then gives you a list in which each element is a chunk from the string. For example:

```
>>> string = "This is a test string";
>>> string.split(" ")
['This', 'is', 'a', 'test', 'string']
```

Lists are used extensively in Python, although this is slowly changing as the language matures. The way lists are compared and sorted has changed in 3.x, so be sure to check the latest documentation when you attempt to perform these tasks.

Dictionaries

Unlike lists, dictionaries are collections with no fixed order. Instead, they have a *key* (the name of the element) and a *value* (the content of the element), and Python places them wherever it needs to for maximum performance. When defining dictionaries, you need to use braces (`{ }`) and colons (`:`). You start with an opening brace and then give each element a key and a value, separated by a colon, like this:

```
>>> mydict = { "perl" : "a language", "php" : "another language" }
>>> mydict
{'php': 'another language', 'perl': 'a language'}
```

This example has two elements, with keys `perl` and `php`. However, when the dictionary is printed, we find that `php` comes before `perl`—Python hasn't respected the order in which we entered them. We can index into a dictionary using the normal code:

```
>>> mydict["perl"]'a language'
```

However, because a dictionary has no fixed sequence, we cannot take a slice or index by position.

Like lists, dictionaries are mutable and can also be nested; however, unlike lists, you cannot merge two dictionaries by using `+`. This is because dictionary elements are located using the key. Therefore, having two elements with the same key would cause a clash. Instead, you should use the `update()` method, which merges two arrays by overwriting clashing keys.

You can also use the `keys()` method to return a list of all the keys in a dictionary.

There are subtle differences in how dictionaries are used in 3.x, such as the need to make list calls to produce all values at once so they may be printed. Again, be sure to check the latest documentation as you move ahead, because there are several of these changes in functionality with some tools now behaving differently or even disappearing and other new dictionary tools being added. Some of those features have been backported; for example, Python 2.7 received support for ordered dictionaries, backported from Python 3.1 (see http://docs.python.org/dev/whatsnew/2.7.html#pep-0372).

Conditionals and Looping

So far, we have just been looking at data types, which should show you how powerful Python's data types are. However, you cannot write complex programs without conditional statements and loops.

Python has most of the standard conditional checks, such as `>` (greater than), `<=` (less than or equal to), and `==` (equal), but it also adds some new ones, such as `in`. For example, we can use `in` to check whether a string or a list contains a given character/element:

```
>>> mystring = "J Random Hacker"
>>> "r" in mystring
True
>>> "Hacker" in mystring
True
>>> "hacker" in mystring
False
```


The last example demonstrates how it is case sensitive. We can use the operator for lists, too:

```
>>> mylist = ["soldier", "sailor", "tinker", "spy"]
>>> "tailor" in mylist
False
```

Other comparisons on these complex data types are done item by item:

```
>>> list1 = ["alpha", "beta", "gamma"]
>>> list2 = ["alpha", "beta", "delta"]
>>> list1 > list2
True
```

`list1`'s first element (`alpha`) is compared against `list2`'s first element (`alpha`) and, because they are equal, the next element is checked. That is equal also, so the third element is checked, which is different. The `g` in `gamma` comes after the `d` in `delta` in the alphabet, so `gamma` is considered greater than `delta` and `list1` is considered greater than `list2`.

Loops come in two types, and both are equally flexible. For example, the `for` loop can iterate through letters in a string or elements in a list:

```
>>> string = "Hello, Python!"
>>> for s in string: print s,
...
H e l l o ,   P y t h o n !
```

The `for` loop takes each letter in a string and assigns it to `s`. This then is printed to the screen using the `print` command, but note the comma at the end; this tells Python not to insert a line break after each letter. The ellipsis (`...`) is there because Python allows you to enter more code in the loop; you need to press Enter again here to have the loop execute.

You can use the same construct for lists:

```
>>> mylist = ["andi", "rasmus", "zeev"]
>>> for p in mylist: print p
...
andi
rasmus
zeev
```

Without the comma after the `print` statement, each item is printed on its own line.

The other loop type is the `while` loop, and it looks similar:

```
>> while 1: print "This will loop forever!"
...
This will loop forever!
This will loop forever!
This will loop forever!
This will loop forever!
This will loop forever!
(etc)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyboardInterrupt
>>>
```

That is an infinite loop (it carries on printing that text forever), so you need to press Ctrl+C to interrupt it and regain control.

If you want to use multiline loops, you need to get ready to use your Tab key: Python handles loop blocks by recording the level of indent used. Some people find this odious; others admire it for forcing clean coding on users. Most of us, though, just get on with programming.

For example:

```
>>> i = 0
>>> while i < 3:
...     j = 0
```

```
...     while j < 3:
...             print "Pos: " + str(i) + "," + str(j) + ")"
...             j += 1
...     i += 1
...
Pos: (0,0)
Pos: (0,1)
Pos: (0,2)
Pos: (1,0)
Pos: (1,1)
Pos: (1,2)
Pos: (2,0)
Pos: (2,1)
Pos: (2,2)
```

You can control loops using the `break` and `continue` keywords. `break` exits the loop and continues processing immediately afterward, and `continue` jumps to the next loop iteration.

Again, we find many subtle changes in 3.x. Most conditionals and looping work the same, but because we mention it in this section, we should note that `print` is no longer a statement in 3.x and is now a function call. That provides a nice segue to our discussion of functions.

Functions

Other languages—such as PHP—read and process an entire file before executing it, which means you can call a function before it is defined because the compiler reads the definition of the function before it tries to call it. Python is different: If the function definition has not been reached by the time you try to call it, you get an error. The reason behind this behavior is that Python actually creates an object for your function, and that in turns means two things. First, you can define the same function several times in a script and have the script pick the right one at runtime. Second, you can assign the function to another name just by using `=`.

Function definition starts with `def`, then the function name, then parentheses and a list of parameters, and then a colon. The contents of a function need to be indented at least one level beyond the definition. So, using function assignment and dynamic declaration, you can write a script that prints the right greeting in a roundabout manner:

```
>>> def hello_english(Name):
...     print "Hello, " + Name + "!"
...
>>> def hello_hungarian(Name):
...     print "Szia, " + Name + "!"
...
>>> hello = hello_hungarian
```

```
>>> hello("Paul")
Szia, Paul!
>>> hello = hello_english
>>> hello("Paul")
```

Notice that function definitions include no type information. Functions are typeless, as mentioned previously. The upside of this is that you can write one function to do several things:

```
>>> def concat(First, Second):
...     return First + Second
...
>>> concat(["python"], ["perl"])
['python', 'perl']
>>> concat("Hello, ", "world!")'Hello, world!'
```

That demonstrates how the `return` statement sends a value back to the caller, but also how a function can do one thing with lists and another thing with strings. The magic here is being accomplished by the objects. We write a function that tells two objects to add themselves together, and the objects intrinsically know how to do that. If they do not—if, perhaps, the user passes in a string and an integer—Python catches the error for us. However, it is this hands-off, "let the objects sort themselves out" attitude that makes functions so flexible. Our `concat()` function could conceivably concatenate strings, lists, or *zonks*—a data type someone created herself that allows adding. The point is that we do not limit what our function can do. Tacky as it might sound, the only limit is your imagination.

Object Orientation

After having read this far, you should not be surprised to hear that Python's object orientation is flexible and likely to surprise you if you have been using C-like languages for several years.

The best way to learn Python *object-oriented programming (OOP)* is to just do it. So, here is a basic script that defines a class, creates an object of that class, and calls a function:

```
class Dog(object):
    def bark(self):
        print "Woof!"

fluffy = Dog()
fluffy.bark()
```

Defining a class starts, predictably, with the `class` keyword followed by the name of the class you are defining and a colon. The contents of that class need to be indented one level so that Python knows where it stops. Note that the object inside parentheses is there for object inheritance, which is discussed later. For now, the least you need to know is

that if your new class is not based on an existing class, you should put `object` inside parentheses as shown in the previous code.

Functions inside classes work in much the same way as normal functions do (although they are usually called *methods*); the main difference is that they should all take at least one parameter, usually called `self`. This parameter is filled with the name of the object the function was called on, and you need to use it explicitly.

Creating an instance of a class is done by assignment. You do not need any new keyword, as in some other languages—providing the parentheses makes the creation known. Calling a function of that object is done using a period, the name of the class to call, with any parameters being passed inside parentheses.

Class and Object Variables

Each object has its own set of functions and variables, and you can manipulate those variables independently of objects of the same type. In addition, some class variables are set to a default value for all classes and can be manipulated globally.

This script demonstrates two objects of the `dog` class being created, each with its own name:

```
class Dog(object):
     name = "Lassie"
     def bark(self):
         print self.name + " says 'Woof!'"
     def set_name(self, name):
         self.name = name
fluffy = Dog()
fluffy.bark()
poppy = Dog()
poppy.set_name("Poppy")
poppy.bark()
```

That outputs the following:

```
Lassie says 'Woof!'
Poppy says 'Woof!'
```

There, each dog starts with the name `Lassie`, but it gets customized. Keep in mind that Python assigns by reference by default, meaning each object has a reference to the class's name variable, and as we assign that with the `set_name()` method, that reference is lost. This means that any references we do not change can be manipulated globally. Thus, if we change a class's variable, it also changes in all instances of that class that have not set their own value for that variable. For example:

```
class Dog(object):
```

```
    name = "Lassie"
    color = "brown"
fluffy = Dog()
poppy = Dog()
print fluffy.color
Dog.color = "black"
print poppy.color
fluffy.color = "yellow"
print fluffy.color
print poppy.color
```

So, the default color of dogs is `brown`—both the `fluffy` and `poppy Dog` objects start off as `brown`. Then, using `Dog.color`, we set the default color to be `black`, and because neither of the two objects has set its own color value, they are updated to be `black`. The third-to-last line uses `poppy.color` to set a custom color value for the `poppy` object—`poppy` becomes `yellow`, whereas `fluffy` and the `dog` class in general remain `black`.

Constructors and Destructors

To help you automate the creation and deletion of objects, you can easily override two default methods: `__init__` and `__del__`. These are the methods called by Python when a class is being instantiated and freed, known as the *constructor* and *destructor*, respectively.

Having a custom constructor is great for when you need to accept a set of parameters for each object being created. For example, you might want each dog to have its own name on creation, and you could implement that with this code:

```
class Dog(object):
    def __init__(self, name):
        self.name = name
fluffy = Dog("Fluffy")
print fluffy.name
```

If you do not provide a name parameter when creating the `Dog` object, Python reports an error and stops. You can, of course, ask for as many constructor parameters as you want, although it is usually better to ask only for the ones you need and have other functions to fill in the rest.

On the other side of things is the destructor method, which enables you to have more control over what happens when an object is destroyed. Using the two, we can show the life cycle of an object by printing messages when it is created and deleted:

```
class dog(object):
    def __init__(self, name):
        self.name = name
        print self.name + " is alive!"
    def __del__(self):
```

```
        print self.name + " is no more!"
fluffy = Dog("Fluffy")
```

The destructor is there to give you the chance to free up resources allocated to the object, or perhaps log something to a file. Note that although it is possible to override the `__del__` method, it is a very dangerous idea because it is not guaranteed that `__del__` methods are called for objects that still exist when the interpreter exits. The official Python documentation explains this well at http://docs.python.org/reference/datamodel.html#object.__del__.

Class Inheritance

Python allows you to reuse your code by inheriting one class from one or more others. For example, lions, tigers, and bears are all mammals and so share a number of similar properties. In that scenario, you do not want to have to copy and paste functions between them; it is smarter (and easier) to have a mammal class that defines all the shared functionality and then inherit each animal from that.

Consider the following code:


```
class Car(object):
    color = "black"
    speed = 0
    def accelerate_to(self, speed):
        self.speed = speed
    def set_color(self, color):
        self.color = color
mycar = Car()
print mycar.color
```

That creates a `Car` class with a default color and provides a `set_color()` function so that people can change their own colors. Now, what do you drive to work? Is it a car? Sure it is, but chances are it is a Ford, or a Dodge, or a Jeep, or some other make; you do not get cars without a make. On the other hand, you do not want to have to define a class `Ford` and give it the methods `accelerate_to()`, `set_color()`, and however many other methods a basic car has and then do the same thing for Ferrari, Nissan, and so on.

The solution is to use inheritance: Define a `car` class that contains all the shared functions and variables of all the makes and then inherit from that. In Python, you do this by putting the name of the class from which to inherit inside parentheses in the `class` declaration, like this:

```
class Car(object):
    color = "black"
    speed = 0
    def accelerate_to(self, speed):
        self.speed = speed
    def set_color(self, color):
```

```
        self.color = color
class Ford(Car): pass
class Nissan(Car): pass
mycar = Ford()
print mycar.color
```

The `pass` directive is an empty statement; it means our class contains nothing new. However, because the `Ford` and `Nissan` classes inherit from the `car` class, they get `color`, `speed`, `accelerate_to()`, and `set_color()` provided by their parent class. Note that we do not need objects after the class names for `Ford` and `Nissan` because they are inherited from an existing class: `car`.

By default, Python gives you all the methods the parent class has, but you can override that by providing new implementations for the classes that need them. For example:

```
class ModelT(car):
    def set_color(self, color):
        print "Sorry, Model Ts only come in black!"

myford Ford()
Ford.set_color("green")
mycar = ModelT()
mycar.set_color("green")
```

The first car is created as a `Ford`, so `set_color()` works fine because it uses the method from the `car` class. However, the second car is created as a `Model T`, which has its own `set_color()` method, so the call fails.

This provides an interesting scenario: What do you do if you have overridden a method and yet want to call the parent's method also? If, for example, changing the color of a `Model T` were allowed but just cost extra, we would want to print a message saying *"You owe $50 more,"* but then change the color. To do this, we need to use the class object from which our current class is inherited (`Car` in this case). Here's an example:

```
class ModelT(Car):
    def set_color(self, color):
        print "You owe $50 more"
        Car.set_color(self, color)

mycar = ModelT()
mycar.set_color("green")
print mycar.color
```

That prints the message and then changes the color of the car.

The Standard Library and the Python Package Index

A default Python install includes many modules (blocks of code) that enable you to interact with the operating system, open and manipulate files, parse command-line options, perform data hashing and encryption, and much more. This is one of the reasons most commonly cited when people are asked why they like Python so much: It comes stocked to the gills with functionality you can take advantage of immediately. In fact, the number of modules included in the Standard Library is so high that entire books have been written about them.

For unofficial scripts and add-ons for Python, the recommended starting place is called the *Python Package Index (PyPI)* at http://pypi.python.org/pypi. There you can find well more than 10,000 public scripts and code examples for everything from mathematics to games.

References

- **www.python.org**—The Python website is packed with information and updated regularly. This should be your first stop, if only to read the latest Python news.
- **www.jython.org**—Python also has an excellent Java-based interpreter to allow it to run on other platforms. If you prefer Microsoft .NET, try www.ironpython.com.
- **http://www.montypython.com/**—Guido van Rossum borrowed the name for his language from *Monty Python's Flying Circus*, and as a result, many Python code examples use oblique *Monty Python* references (spam and eggs are quite common, for example). A visit to the official *Monty Python* site to hone your Python knowledge is highly recommended.
- **www.zope.org**—The home page of the Zope *content management system (CMS)*. It is one of the most popular CMSes around and, more important, written entirely in Python.
- If you want a book, we recommend starting with *Learning Python,* by Mark Lutz (O'Reilly), ISBN: 0-596-15806-8, and for advanced work, move to *Programming Python*, also by Mark Lutz (O'Reilly), ISBN: 0-596-15810-6.