1

A Python Primer

This chapter provides a quick overview of the Python language. The goal in this chapter is not to teach you the Python language — excellent books have been written on that subject, such as *Beginning Python* (Wrox, 2005). This chapter describes Python's lexical structure and programming conventions, so if you are familiar with other scripting languages such as Perl or Ruby, or with compiled programming languages such as Java or C#, you should easily be up to speed in no time.

Getting Started

Of course, the first thing you need to do is install Python, if you don't already have it. Installers are available for Windows, Macintosh, Linux, Unix, and everything from OpenVMS to the Playstation (no, I'm not kidding).

Obtaining Python and Installing It

If you go to `www.python.org/download` you can find links to download the correct version of Python for your operating system. Follow the install instructions for your particular Python distribution — instructions can vary significantly depending on what operating system you're installing to.

What Version Number to Install

Although the examples in this book should work for any Python version above 2.0, it is best to install the latest stable build for your operating system. For Windows (which is the environment I primarily work in), the latest stable version is 2.51. There is an alpha build of Python 3.0 available as of this writing, but other than just looking at it for fun, I'd steer clear of it for the examples in this book — in some cases the syntax is very different, and the examples in this book won't work with Python 3.0.

The Python Interpreter

One of the most useful tools for writing Python code is the *Python interpreter*, an interactive editing and execution environment in which commands are run as soon as you enter them and press Enter. On Unix and Macintosh machines, the Python interpreter can usually be found in the `/usr/local/bin/python` directory, which can be accessed by simply typing the command **python**.

On Windows machines, the Python interpreter is installed to the `c:\python25` directory (for a Python 2.5x installation). To add this directory to your path, type the following at a Windows command prompt: **set path=%path%;C:\python25.**

On a Windows system, such as with Unix/Linux, you simply type **python** to bring up the interpreter (either from the `c:\python25` directory or from any directory if the Python directory has been added to the path).

When you enter the interpreter, you'll see a screen with information like the following:

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Your Editing/Execution Environment

Because the minimum requirements for writing and running Python programs are simply an editor that can save text files and a command prompt where you can run the Python interpreter, you could simply use Notepad on Windows, Vim on Linux/Unix, or TextEdit on Mac, and a command line for running programs.

One nice step up from that is IDLE, Python's integrated development environment (IDE), which is named after Monty Python's Eric Idle and is included with Python. It includes the following useful features:

- ❑ A full-featured text editor
- ❑ Syntax highlighting
- ❑ Code intelligence
- ❑ A class browser
- ❑ A Python path browser
- ❑ A debugger
- ❑ A Python interpreter environment

In addition to IDLE, you do have other options. On Windows, there is a nice IDE called PythonWin, developed by Mark Hammond. It can be installed as a full Python distribution from ActiveState's website (`www.activestate.com`), or you can simply install the win32all package to add PythonWin to a standard Python for Windows install. PythonWin is a great product, very slick and with all the features you'd expect from an IDE.

Other options include an Eclipse distribution for Python called EasyEclipse for Python. For my money, I'd start out with IDLE, and then as your experience with Python grows, explore other options.

Lexical Structure

Following is a simple Python program. It shows the basic structure of many Python scripts, which is as follows:

1. Initialize variables (lines 1–3).
2. Do some processing (lines 4–5).
3. Make decisions and perform actions based on those decisions (lines 6–10).

```
name = "Jim"
age = 42
highschoolGPA = 3.89

enteredName = raw_input("Enter your name: ")

print "\n\n"

if name == "Jim":
    print "Your age is ", age
    print "You had a", highschoolGPA, "GPA in high school"
    if (highschoolGPA > 3):
        print "You had better than a 3.0 GPA...good job!"
```

Keywords

Keywords are words that are "reserved" — they cannot be used as variable names. In the preceding code, the keyword `if` is used multiple times.

The keywords are as follows:

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	
def	finally	in	print	yield

Lines and Indentation

In Python, unlike a compiled language such as C, line breaks are significant, and the end of a program statement is defined by a hard return. Program blocks are defined by a combination of statements (each on a separate line, but with no end-of-statement character visible) and program blocks, delimited visually by the use of indentation.

As shown in the code from the preceding section, lines are indented in Python. This is not simply a stylistic choice — indentation is not just recommended in Python, but enforced by the interpreter. This is probably the most controversial aspect of Python, and it has been the subject of many a flame war online.

Basically, it means that the following code would generate an interpreter error, because the action associated with an `if` statement must be indented:

```
if variable1 == "Jim":
print "variable1 eqiuals Jim"
```

You'll learn more about the actual `if` statement itself later.

Data Types and Identifiers

Python provides a rich collection of data types to enable programmers to perform virtually any programming task they desire in another language. One nice thing about Python is that it provides many useful and unique data types (such as tuples and dictionaries), and stays away from data types such as the pointers used in C, which have their use but can also make programming much more confusing and difficult for the nonprofessional programmer.

Data Types

Python is known as a *dynamically typed* language, which means that you don't have to explicitly identify the data type when you initialize a variable. In the code example above, the variable `name` is assigned to the string value "Jim". However, you don't specifically identify the variable as a string variable. Python knows, based on the value it has been given, that it should allocate memory for a string. Likewise for the `age` integer variable and the `highschoolGPA` float variable.

The following table shows the most commonly used available data types and their attributes:

Data Type	Attributes	Example
Numeric Types		
Float	Implemented with C doubles.	5.43 9483.123
Integer	Implemented with C longs.	1027 211234
Long Integer	Size is limited only by system resources.	567893L
Sequence Types		
String	A list of characters. Is immutable (not changeable in-place). Can be represented by single quotes or double quotes. Can span multiple lines.	"This is a string" """" This is an example of a DocString """"
List	A mutable (changeable) sequence of data types. List elements do not have to be "like." In other words, you could have a float element and an integer element in a single list.	[1, 2.3, "Jim"] [1, 2, 3] [1.5, 2.7, 3.0] ["Jim", "Joe", "Bob"]
Tuple	An immutable sequence of data types. Other than the fact that it can't be changed, it works just like a list.	(1, 2.3, "Jim") (1, 2, 3) (1.5, 2.7, 3.0) "Jim", "Joe", "Bob"

Dictionary	A list of items indexed by keys.	d = {"first":"Jim",
		"last":"Knowlton"}

Identifiers

An *identifier* is a unique name that enables you to identify something. Identifiers are used to label variables, functions, classes, objects, and modules. They begin with either a letter or an underscore, and they can contain letters, underscores, or digits. They cannot contain punctuation marks.

Operators

If you have programmed in other languages, the operators in Python will be familiar to you. The Python operators are fundamentally similar to those used in other languages. In the code shown earlier, the conditions evaluated in both `if` statements involve comparison operators. The following table describes the operators most commonly used in Python, and the ones used in this book:

Operator	Symbol	Example
Numeric Operators		
Addition	+	x + y
Subtraction	–	x – y
Multiplication	*	x * y
Division	/	x / y
Exponent (Power)	**	x ** y (x to the y power)
Modulo	%	x % y (the remainder of x/y)
Comparison Operators		
Greater than	>	x > y (x is greater than y)
Less than	<	x < y (x is less than y)
Equal to	==	x == y (x equals y)
Greater than or equal to	>=	x >= y (x is greater than or equal to y)
Less than or equal to	<=	x <= y (x is less than or equal to y)
Not equal to	!= or <>	x != y, x <> y (x does not equal y)
Boolean Operators		
and	and	x and y (if both are true, then the expression is true)
or	or	x or y (if either is true, then the expression is true)
not	not	not x (if x is false, then the expression is true)
Assignment Operator		

Assignment	=	X = 15
		name = "Jim"

Expressions and Statements

Expressions and statements are the building blocks of Python programs. They are the equivalent of phrases and sentences in English. To understand Python, it's critical to understand how to put these building blocks together.

Expressions

Expressions consist of combinations of *values*, which can be either constant values, such as a string ("Jim") or a number (12), and *operators*, which are symbols that act on the values in some way.

The following examples are expressions:

```
10 - 4

11 * (4 + 5)

x - 5

a / b
```

Operator Precedence in Expressions

When you have a multiple expression like `5 + 4 * 7`, which operation is done first, the addition or the multiplication? If it isn't too painful to recall your high school algebra class, you might remember learning the rules of *operator precedence*. These kinds of complex expressions require a set of rules defining which expressions are executed first.

The following list describes the basic rules of operator precedence in Python (don't worry if you don't understand all the terms right now; they'll be explained as you need them):

- Expressions are evaluated from left to right.
- Exponents, multiplication, and division are performed before addition and subtraction.
- Expressions in parentheses are performed first.
- Mathematical expressions are performed before Boolean expressions (`AND`, `OR`, `NOT`)

Statements

The *statement* is the basic unit of programming. In essence, it says "do this to this." Statements in Python are not delimited by a visible character, such as the semicolon in C or C#. Every time you press Enter and start a new line, you are entering a new statement.

For example, if you type:

```
Print 12 + 15
```

into the Python interpreter, you'll get the following output:

```
>>> print 12 + 15
27
>>>
```

This is because you told the system to "print the result of the expression 12 + 15," which is a complete statement.

However, if you type:

```
print 12 +
```

you'll get a syntax error, as shown here:

```
>>> print 12 +
SyntaxError: invalid syntax
>>>
```

Clearly, the system cannot read this because it isn't a complete statement, so it results in an error.

Multi-line Statements

It *is* possible to have a single statement span multiple lines. You could do this for aesthetic reasons or simply because the line is too long to read on one screen. To do this, simply put a space and a backslash at the end of the line. Here are a few examples:

```
name = "Jim \
   Knowlton"

sum = 12 + \
   13
```

Iteration and Decision-Making

There are two basic ways to control the flow of a program: through iteration (looping) and through decision-making.

Iteration

Iteration in Python is handled through the "usual suspects": the `for` loop and the `while` loop. However, if you've programmed in other languages, these seemingly familiar friends are a little different.

For Loops

Unlike in Java, the `for` loop in Python is more than a simple construct based on a counter. Instead, it is a sequence iterator that will step through the items of any sequenced object (such as a list of names, for instance). Here's a simple example of a `for` loop:

```
>>> names = ["Jim", "Joe"]
>>> for x in names:
  print x


  Jim
  Joe
>>>
```

As you can see, the basic syntax is `for <variable> in <object>:`, followed by the code block to be iterated.

While Loops

A `while` loop is similar to a `for` loop but it's more flexible. It enables you to test for a particular condition and then terminate the loop when the condition is true. This is great for situations when you want to terminate a loop when the program is in a state that you can't predict at runtime (such as when you are processing a file, and you want the loop to be done when you reach the end of the file).

Here's an example of a `while` loop:

```
>>> counter = 5
>>> x = 0
>>> while x < counter:
 print "x=",x
 print "counter = ", counter
 x += 1


x =  0
counter =  5
x =  1
counter =  5
x =  2
counter =  5
x =  3
counter =  5
x =  4
counter =  5
>>>
```

Break and Continue

As with C, in Python you can break out of the innermost `for` or `while` loop by using the `break` statement. Also as with C, you can continue to the next iteration of a loop by using the `continue` statement.

What about switch or case?

Many of you familiar with other programming languages are no doubt wondering about a decision-tree structure similar to C's switch statement or Pascal's case. Unfortunately, you won't find it in Python. However, the conditional `if-elif-else` structure, along with other constructs you'll learn about later, make their absence not such a big deal.

Decision-Making

When writing a program, it is of course critical to be able to evaluate conditions and make decisions. Having an `if` construct is critical for any language, and Python is no exception.

The if Statement

The `if` statement in Python, as in other languages, evaluates an expression. If the expression is true, then the code block is executed. Conversely, if it isn't true, then program execution jumps to the end. Python also supports use of zero or more `elif` statements (short for "else if"), and an optional `else` statement, which appears at the end if you also have `elif` statements, and would be the "default" choice if none of the `if` statements were true.

Here's an example:

```
>>> name = "Jim"
>>> if name == "Jim":
 print "your name is Jim"
elif name == "Joe":
 print "your name is Joe"
else:
 print "I have no idea what your name is"


your name is Jim
>>>
```

Functions

In many ways, the principle behind a function is analogous to turning on a TV. You don't have to understand all the electronics and communications technology behind getting the TV signal to your receiver in order to operate the TV. You do have to know some simple behaviors, however, such as how to turn it on, where the volume switch is, and so on. In a similar fashion, a function gives the program an interface through which it can run program code without knowing the details about the code being run.

Defining a Function

You define a function in Python with the following simple syntax:

```
def functionName(paramenter1, parameter2=default_value):
 <code block>
 return value (optional)
```

Note two elements in the preceding example:

- **Parameters** — As you can see, parameters can simply be a variable name (making them required as part of the function call), or they can have a default value, in which case it is optional to pass them in the function call.
- **The return statement** — This enables the function to return a value to the code that called it. The nice thing about this is that you can run a function and assign its output to a variable.

Here's an example of a function definition:

```
>>> def getname(name):
 return name + " is very hungry"

>>>
```

Calling a Function

To call a function, simply enter the function name with the function signature:

```
functionName(paramenter1, parameter2)
```

If a parameter has a default value in its definition, then you can omit that parameter when you call the function, and the parameter will contain its default value. Alternately, you can override the default value by entering the value yourself when you call the function.

For example, if a function were defined as follows:

```
def jimsFunc(age, name = "Jim"):
```

Then you could call the function in any of the following three ways:

```
jimsFunc(23)

jimsFunc(42, "James")
jimsFunc(42, firstName="Joe")
```

In the first example, I simply took the default value for the first parameter; in the second, I replaced it with "James."

Modules

A *module* is the highest-level programming unit in Python. A module usually corresponds to a program file in Python. Unlike in Ruby, modules are not declared — the name of the `*.py` file is the name of the module. In other words, basically each file is a module, and modules import other modules to perform various programming tasks.

Importing Modules

Importing modules is done with either the `import` or `reload` command.

Import

To use a module, you `import` it. Usually import statements occur at the beginning of the Python module. Importing modules is a fairly simple operation, but it requires a little explanation. Consider the following examples:

```
1.      import os
2.      import os, sys
3.      from os import getcwd
4.      import os as operatingSystem
```

These examples highlight some variations in how you can import modules:

1. This first example is the simplest and easiest to understand. It is merely the keyword `import` followed by the module name (in this case, `os`).
2. Multiple modules can be imported with the same `import` command, with the modules separated by a comma.

3. You can import specific names only within a module, without importing the whole module, by using the `from <module> import <name>` statement. This can be useful for performance reasons if you only need one function from a large module.
4. If a module has a name that's difficult to work with or remember, and you want to use a name to represent it that is meaningful to you, simply use the `as` keyword and `import <module> as <identifier>`.

Reload

`Reload` is another very useful command, especially when entering code within the Python interactive interpreter. It enables you to reload a particular module without reloading Python. For example, if you wanted to reload the `os` module, you would simply enter `reload os`.

If you're wondering why you would ever want to do that, one scenario would be if you have a Python script that runs all the time and it accesses a module on another machine. Assuming you always want to ensure that you're running the most current version of the remote module you're accessing, you'd use the `reload` command.

How Python Finds Modules to Load

When you use an `import` statement, you don't tell Python where the module that needs to be loaded is located. How, then, does it know where to find the file? The answer to that question is the *module search path*.

The Module Search Path

Python has a predefined priority specifying where it should look for modules, known as the module search path. When you enter an `import` command and the name of the module, Python checks the following locations in the order shown here:

1. **The home directory** — This is either the directory from which you launched the Python interactive interpreter or the directory where the main Python program is located.
2. `PYTHONPATH` — This is an environment variable set in the system. Its value is a list of directories, which Python will search for modules.
3. **Standard library directories** — The directory in which the standard libraries are located are searched next.

Exploring sys.path

If you ever want to see your system's Python search path, all you have to do is bring up the interactive interpreter, import the `sys` module, and type **sys.path**. The full Python module search path will be returned, as shown in the following example:

```
>>> import sys
>>> sys.path
['C:\\Python25', 'C:\\Python25\\Lib\\idlelib', 'C:\\Program Files\\PythonNet',
'c:\\scripts\\python', 'c:\\python25', 'C:\\Python25\\pyunit-1.4.1',
'c:\\python25\\pamie', 'C:\\WINDOWS\\system32\\python25.zip', 'C:\\Python25\\DLLs',
'C:\\Python25\\lib', 'C:\\Python25\\lib\\plat-win', 'C:\\Python25\\lib\\lib-tk',
'C:\\Python25\\lib\\site-packages', 'C:\\Python25\\lib\\site-packages\\win32',
'C:\\Python25\\lib\\site-packages\\win32\\lib', 'C:\\Python25\\lib\\site-
packages\\
Pythonwin', 'C:\\Python25\\lib\\site-packages\\wx-2.8-msw-ansi']
>>>
```

Classes

Python is a language that can support both procedural programming and object-oriented programming. Here is an example of a Python class:

```
>>> class name1():
 def setmyname(self, myname):
         self.name = myname


>>> jimname = name1()
>>> jimname.setmyname("Jim")
>>> print jimname.name
Jim
>>>
```

Note some points about Python's implementation of class programming as demonstrated in the preceding example:

- ❑ If we were inheriting from other classes, those class names would have been inside the parentheses of the `class name1():` definition.
- ❑ In this case, there is one class method, `setmyname`. If we wanted to create a constructor for the class, it would be named `__init__` .
- ❑ To create an instance of a class, you simply assign a variable to the class definition, as in `jimname = name1()` .
- ❑ Attributes are accessed with familiar dot notation (instance `variable.attribute`) such as `jimname.name` .

Summary

This chapter provided a brief tour of the Python language, including the following highlights:

- How to get up and running with Python
- Python's lexical structure
- Operators, expressions, and statements
- Iteration and decision-making
- Functions and modules
- Classes and object-oriented programming

Of course, there is much more to the Python language than what this short chapter has outlined. Much of it you'll discover as you work through the projects in this book.

Let's get started!