1

A Tutorial Introduction

THIS CHAPTER PROVIDES A QUICK INTRODUCTION to Python. The goal is to illustrate Python's essential features without getting too bogged down in special rules or details. To do this, the chapter briefly covers basic concepts such as variables, expressions, control flow, functions, classes, and input/output. This chapter is not intended to provide comprehensive coverage, nor does it cover all of Python's more advanced features. However, experienced programmers should be able to extrapolate from the material in this chapter to create more advanced programs. Beginners are encouraged to try a few examples to get a feel for the language.

Running Python

Python programs are executed by an interpreter. On most machines, the interpreter can be started by simply typing **`python`**. However, many different programming environments for Python are currently available (for example, ActivePython, PythonWin, IDLE, and PythonIDE). In this case, Python is started by launching the appropriate application. When the interpreter starts, a prompt appears at which you can start typing programs into a simple read-evaluation loop. For example, in the following output, the interpreter displays its copyright message and presents the user with the `>>>` prompt, at which the user types the familiar "Hello World" command:

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>>
```

Programs can also be placed in a file such as the following:

```
# helloworld.py
print "Hello World"
```

Python source files are ordinary text files and normally have a `.py` suffix. The `#` character denotes a comment that extends to the end of the line.

To execute the `helloworld.py` file, you provide the filename to the interpreter as follows:

```
% python helloworld.py
Hello World
%
```

On Windows, Python programs can be started by double-clicking a `.py` file or typing the name of the program into the "run" command on the Windows "Start" menu. This launches the interpreter and runs the program in a console window. In this case, the console window disappears immediately after the program completes its execution (often before you can read its output). To prevent this problem, you should use an integrated development environment such as PythonWin. An alternative approach is to launch the program using a `.bat` file containing a statement such as `python -i helloworld.py` that instructs the interpreter to enter interactive mode after program execution.

Within the interpreter, the `execfile()` function runs a program, as in the following example:

```
>>> execfile("helloworld.py")
Hello World
```

On UNIX, you can also invoke Python using `#!` in a shell script:

```
#!/usr/local/bin/python
print "Hello World"
```

The interpreter runs until it reaches the end of the input file. If it's running interactively, you can exit the interpreter by typing the EOF (end of file) character or by selecting Exit from a pull-down menu. On UNIX, EOF is Ctrl+D; on Windows, it's Ctrl+Z. A program can also exit by calling the `sys.exit()` function or raising the `SystemExit` exception. For example:

```
>>> import sys
>>> sys.exit()
```

or

```
>>> raise SystemExit
```

Variables and Arithmetic Expressions

The program in Listing 1.1 shows the use of variables and expressions by performing a simple compound-interest calculation.

Listing 1.1 **Simple Compound-Interest Calculation**

```
principal = 1000        # Initial amount
rate = 0.05             # Interest rate
numyears = 5            # Number of years
year = 1
while year <= numyears:
        principal = principal*(1+rate)
        print year, principal
        year += 1
```

The output of this program is the following table:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
```

Python is a dynamically typed language in which names can represent values of different types during the execution of a program. In fact, the names used in a program are

really just labels for various quantities and objects. The assignment operator simply creates an association between a name and a value. This is different from C, for example, in which a name represents a fixed size and location in memory into which results are placed. The dynamic behavior of Python can be seen in Listing 1.1 with the `principal` variable. Initially, it's assigned to an integer value. However, later in the program it's reassigned as follows:

```
principal = principal*(1+rate)
```

This statement evaluates the expression and reassociates the name `principal` with the result. When this occurs, the original binding of `principal` to the integer `1000` is lost. Furthermore, the result of the assignment may change the type of the variable. In this case, the type of `principal` changes from an integer to a floating-point number because `rate` is a floating-point number.

A newline terminates each individual statement. You also can use a semicolon to separate statements, as shown here:

```
principal = 1000; rate = 0.05; numyears = 5;
```

The `while` statement tests the conditional expression that immediately follows. If the tested statement is true, the body of the `while` statement executes. The condition is then retested and the body executed again until the condition becomes false. The body of the loop is denoted by indentation; the three statements following `while` in Listing 1.1 execute on each iteration. Python doesn't specify the amount of required indentation, as long as it's consistent within a block.

One problem with the program in Listing 1.1 is that the output isn't very pretty. To make it better, you could right-align the columns and limit the precision of `principal` to two digits by modifying `print` to use a format string, like this:

```
print "%3d  %0.2f" % (year, principal)
```

Now the output of the program looks like this:

```
1  1050.00
2  1102.50
3  1157.63
4  1215.51
5  1276.28
```

Format strings contain ordinary text and special formatting-character sequences such as `"%d"`, `"%s"`, and `"%f"`. These sequences specify the formatting of a particular type of data such as an integer, string, or floating-point number, respectively. The special-character sequences can also contain modifiers that specify a width and precision. For example, `"%3d"` formats an integer right-aligned in a column of width 3, and `"%0.2f"` formats a floating-point number so that only two digits appear after the decimal point. The behavior of format strings is almost identical to the C `sprintf()` function and is described in detail in Chapter 4, "Operators and Expressions."

Conditionals

The `if` and `else` statements can perform simple tests. Here's an example:

```
# Compute the maximum (z) of a and b
if a < b:
        z = b
```

```
else:
        z = a
```

The bodies of the `if` and `else` clauses are denoted by indentation. The `else` clause is optional.

To create an empty clause, use the `pass` statement as follows:

```
if a < b:
        pass      # Do nothing
else:
        z = a
```

You can form Boolean expressions by using the `or`, `and`, and `not` keywords:

```
if b >= a and b <= c:
        print "b is between a and c"
if not (b < a or b > c):
        print "b is still between a and c"
```

To handle multiple-test cases, use the `elif` statement, like this:

```
if a == '+':
        op = PLUS
elif a == '-':
        op = MINUS
elif a == '*':
        op = MULTIPLY
else:
        raise RuntimeError, "Unknown operator"
```

To denote truth values, you can use the Boolean values `True` and `False`. Here's an example:

```
if c in '0123456789':
    isdigit = True
else:
    isdigit = False
```

File Input and Output

The following program opens a file and reads its contents line by line:

```
f = open("foo.txt")        # Returns a file object
line = f.readline()        # Invokes readline() method on file
while line:
        print line,        # trailing ',' omits newline character
        line = f.readline()
f.close()
```

The `open()` function returns a new file object. By invoking methods on this object, you can perform various file operations. The `readline()` method reads a single line of input, including the terminating newline. An empty string is returned at the end of the file.

In the example, the program is simply looping over all the lines in the file `foo.txt`. Whenever a program loops over a collection of data like this (for instance input lines, numbers, strings, and so on), it is commonly known as "iteration." Because iteration is such a common operation, Python provides a number of shortcuts for simplifying the process. For instance, the same program can be written much more succinctly as follows:

```
for line in open("foo.txt"):
print line,
```

To make the output of a program go to a file, you can supply a file to the `print` statement using `>>`, as shown in the following example:

```
f = open("out","w")     # Open file for writing
while year <= numyears:
        principal = principal*(1+rate)
        print >>f,"%3d   %0.2f" % (year,principal)
        year += 1
f.close()
```

In addition, file objects support a `write()` method that can be used to write raw data. For example, the `print` statement in the previous example could have been written this way:

```
f.write("%3d   %0.2f\n" % (year,principal))
```

Although these examples have worked with files, the same techniques apply to the standard output and input streams of the interpreter. For example, if you wanted to read user input interactively, you can read from the file `sys.stdin`. If you want to write data to the screen, you can write to `sys.stdout`, which is the same file used to output data produced by the `print` statement. For example:

```
import sys
sys.stdout.write("Enter your name :")
name = sys.stdin.readline()
```

The preceding code can also be shortened to the following:

```
name = raw_input("Enter your name :")
```

Strings

To create string literals, enclose them in single, double, or triple quotes as follows:

```
a = "Hello World"
b = 'Python is groovy'
c = """What is footnote 5?"""
```

The same type of quote used to start a string must be used to terminate it. Triple-quoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single- and double-quoted strings, which must be specified on one logical line. Triple-quoted strings are useful when the contents of a string literal span multiple lines of text such as the following:

```
print '''Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
'''
```

Strings are sequences of characters indexed by integers, starting at zero. To extract a single character, use the indexing operator *`s[i]`* like this:

```
a = "Hello World"
b = a[4]                # b = 'o'
```

To extract a substring, use the slicing operator *s*[*i*:*j*]. This extracts all elements from *s* whose index *k* is in the range *i* <= *k* < *j*. If either index is omitted, the beginning or end of the string is assumed, respectively:

```
c = a[:5]              # c = "Hello"
d = a[6:]               # d = "World"
e = a[3:8]              # e = "lo Wo"
```

Strings are concatenated with the plus (+) operator:

```
g = a + " This is a test"
```

Other data types can be converted into a string by using either the `str()` or `repr()` function or backquotes (`), which are a shortcut notation for `repr()`. For example:

```
s = "The value of x is " + str(x)
s = "The value of y is " + repr(y)
s = "The value of y is " + `y`
```

In many cases, `str()` and `repr()` return identical results. However, there are subtle differences in semantics that are described in later chapters.

Lists

Lists are sequences of arbitrary objects. You create a list as follows:

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:

```
a = names[2]             # Returns the third item of the list, "Ann"
names[0] = "Jeff"        # Changes the first item to "Jeff"
```

To append new items to the end of a list, use the `append()` method:

```
names.append("Kate")
```

To insert an item in the list, use the `insert()` method:

```
names.insert(2, "Sydney")
```

You can extract or reassign a portion of a list by using the slicing operator:

```
b = names[0:2]                    # Returns [ "Jeff", "Mark" ]
c = names[2:]                     # Returns [ "Sydney", "Ann", "Phil", "Kate" ]
names[1] = 'Jeff'                 # Replace the 2nd item in names with 'Jeff'
names[0:2] = ['Dave','Mark','Jeff'] # Replace the first two items of
                                  # the list with the list on the right.
```

Use the plus (+) operator to concatenate lists:

```
a = [1,2,3] + [4,5]     # Result is [1,2,3,4,5]
```

Lists can contain any kind of Python object, including other lists, as in the following example:

```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

Nested lists are accessed as follows:

```
a[1]            # Returns "Dave"
a[3][2]         # Returns 9
a[3][3][1]      # Returns 101
```

The program in Listing 1.2 illustrates a few more advanced features of lists by reading a list of numbers from a file specified on the command line and outputting the minimum and maximum values.

Listing 1.2 **Advanced List Features**

```
import sys                        # Load the sys module
if len(sys.argv) != 2             # Check number of command line arguments :
     print "Please supply a filename"
     raise SystemExit
f = open(sys.argv[1])             # Filename on the command line
svalues = f.readlines()           # Read all lines into a list
f.close()

# Convert all of the input values from strings to floats
fvalues = [float(s) for s in svalues]

# Print min and max values
print "The minimum value is ", min(fvalues)
print "The maximum value is ", max(fvalues)
```

The first line of this program uses the `import` statement to load the `sys` module from the Python library. This module is being loaded in order to obtain command-line arguments.

The `open()` method uses a filename that has been supplied as a command-line option and stored in the list `sys.argv`. The `readlines()` method reads all the input lines into a list of strings.

The expression `[float(s) for s in svalues]` constructs a new list by looping over all the strings in the list `svalues` and applying the function `float()` to each element. This particularly powerful method of constructing a list is known as a *list comprehension*.

After the input lines have been converted into a list of floating-point numbers, the built-in `min()` and `max()` functions compute the minimum and maximum values.

Tuples

Closely related to lists is the tuple data type. You create tuples by enclosing a group of values in parentheses, like this:

```
a = (1,4,5,-9,10)
b = (7,)                                  # Singleton (note extra ,)
person = (first_name, last_name, phone)
```

Sometimes Python recognizes that a tuple is intended, even if the parentheses are missing:

```
a = 1,4,5,-9,10
b = 7,
person = first_name, last_name, phone
```

Tuples support most of the same operations as lists, such as indexing, slicing, and concatenation. The only difference is that you cannot modify the contents of a tuple after creation (that is, you cannot modify individual elements or append new elements to a tuple).

Sets

A set is used to contain an unordered collection of objects. To create a set, use the `set()` function and supply a sequence of items such as follows:

```
s = set([3,5,9,10])       # Create a set of numbers
t = set("Hello")          # Create a set of characters
```

Unlike lists and tuples, sets are unordered and cannot be indexed in the same way. Moreover, the elements of a set are never duplicated. For example, if you print the value of `t` from the preceding code, you get the following:

```
>>> print t
set(['H', 'e', 'l', 'o'])
```

Notice that only one `'l'` appears.

Sets support a standard collection of set operations, including union, intersection, difference, and symmetric difference. For example:

```
a = t | s           # Union of t and s
b = t & s           # Intersection of t and s
c = t - s           # Set difference (items in t, but not in s)
d = t ^ s           # Symmetric difference (items in t or s, but not both)
```

New items can be added to a set using `add()` or `update()`:

```
t.add('x')
s.update([10,37,42])
```

An item can be removed using `remove()`:

```
t.remove('H')
```

Dictionaries

A *dictionary* is an associative array or hash table that contains objects indexed by keys. You create a dictionary by enclosing the values in curly braces ({ }) like this:

```
a = {
        "username" : "beazley",
        "home" : "/home/beazley",
        "uid" : 500
     }
```

To access members of a dictionary, use the key-indexing operator as follows:

```
u = a["username"]
d = a["home"]
```

Inserting or modifying objects works like this:

```
a["username"] = "pxl"
a["home"] = "/home/pxl"
a["shell"] = "/usr/bin/tcsh"
```

Although strings are the most common type of key, you can use many other Python objects, including numbers and tuples. Some objects, including lists and dictionaries, cannot be used as keys, because their contents are allowed to change.

Dictionary membership is tested with the `has_key()` method, as in the following example:

```
if a.has_key("username"):
     username = a["username"]
else:
     username = "unknown user"
```

This particular sequence of steps can also be performed more compactly as follows:

```
username = a.get("username", "unknown user")
```

To obtain a list of dictionary keys, use the `keys()` method:

```
k = a.keys()            # k = ["username","home","uid","shell"]
```

Use the `del` statement to remove an element of a dictionary:

```
del a["username"]
```

Iteration and Looping

The simple loop shown earlier used the `while` statement. The other looping construct is the `for` statement, which is used to iterate over a collection of items. Iteration is one of Python's most rich features. However, the most common form of iteration is to simply loop over all the members of a sequence such as a string, list, or tuple. Here's an example:

```
for i in range(1,10):
        print "2 to the %d power is %d" % (i, 2**i)
```

The `range(`*i*`,`*j*`)` function constructs a list of integers with values from *i* to *j*`-1`. If the starting value is omitted, it's taken to be zero. An optional stride can also be given as a third argument. For example:

```
a = range(5)          # a = [0,1,2,3,4]
b = range(1,8)        # b = [1,2,3,4,5,6,7]
c = range(0,14,3)     # c = [0,3,6,9,12]
d = range(8,1,-1)     # d = [8,7,6,5,4,3,2]
```

The `range()` function works by constructing a list and populating it with values according to the starting, ending, and stride values. For large ranges, this process is expensive in terms of both memory and runtime performance. To avoid this, you can use the `xrange()` function, as shown here:

```
for i in xrange(1,10):
        print "2 to the %d power is %d" % (i, 2**i)

a = xrange(100000000)       # a = [0,1,2, ..., 99999999]
b = xrange(0,100000000,5)   # b = [0,5,10, ...,99999995]
```

Rather than creating a sequence populated with values, the sequence returned by `xrange()` computes its values from the starting, ending, and stride values whenever it's accessed.

The `for` statement is not limited to sequences of integers and can be used to iterate over many kinds of objects, including strings, lists, and dictionaries. For example:

```
a = "Hello World"
# Print out the characters in a
for c in a:
        print c

b = ["Dave","Mark","Ann","Phil"]
```

```
# Print out the members of a list
for name in b:
        print name

c = { 'a' : 3, 'name': 'Dave', 'x': 7.5 }
# Print out all of the members of a dictionary
for key in c:
        print key, c[key]
```

In addition, the `for` statement can be applied to any object that supports a special iteration protocol. In an earlier example, iteration was used to loop over all the lines in a file:

```
for line in open("foo.txt"):
print line,
```

This works because files provide special iteration methods that work as follows:

```
>>> i = f.__iter__()    # Return an iterator object
>>> i.next()            # Return first line
>>> i.next()            # Return next line
... continues ...
>>> i.next()            # No more data
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

Underneath the covers, the `for` statement relies on these methods to iterate over lines in the file.

Instead of iterating over a collection of items such as the elements of a list, it is also possible to iterate over an object that knows how to generate items on demand. This sort of object is called a *generator* and is defined using a function. For example, if you wanted to iterate over the Fibonacci numbers, you could do this:

```
# Generate fibonacci numbers
def fibonacci(max):
    s = 1
    t = 1
    while s < max:
        yield s          # Produce a value
        w = s + t
        s = t
        t = w
    return

# Print fibonacci numbers less than 1000
for n in fibonacci(1000):
    print n
```

In this case, the `yield` statement produces a value used in iteration. When the next value is requested, the function resumes execution right after `yield`. Iteration stops when the generator function returns. More details about iterators and generators can be found in Chapter 6, "Functions and Functional Programming."

Functions

You use the `def` statement to create a function, as shown in the following example:

```
def remainder(a,b):
        q = a // b       # // is truncating division.
        r = a - q*b
        return r
```

To invoke a function, simply use the name of the function followed by its arguments enclosed in parentheses, such as `result = remainder(37,15)`. You can use a tuple to return multiple values from a function, as shown here:

```
def divide(a,b):
        q = a // b        # If a and b are integers, q is integer
        r = a - q*b
        return (q,r)
```

When returning multiple values in a tuple, it's often useful to invoke the function as follows:

```
quotient, remainder = divide(1456,33)
```

To assign a default value to a parameter, use assignment:

```
def connect(hostname,port,timeout=300):
    # Function body
```

When default values are given in a function definition, they can be omitted from subsequent function calls. When omitted, the argument will simply take on the default value. For example:

```
connect('www.python.org', 80)
```

You also can invoke functions by using keyword arguments and supplying the arguments in arbitrary order. However, this requires you to know the names of the arguments in the function definition. For example:

```
connect(port=80,hostname="www.python.org")
```

When variables are created or assigned inside a function, their scope is local. That is, the variable is only defined inside the body of the function and is destroyed when the function returns. To modify the value of a global variable from inside a function, use the `global` statement as follows:

```
a = 4.5
...
def foo():
        global a
        a = 8.8             # Changes the global variable a
```

Classes

The `class` statement is used to define new types of objects and for object-oriented programming. For example, the following class defines a simple stack with `push()`, `pop()`, and `length()` operations:

```
class Stack(object):
        def __init__(self):           # Initialize the stack
                self.stack = [ ]
        def push(self,object):
                self.stack.append(object)
        def pop(self):
                return self.stack.pop()
```

```
        def length(self):
                return len(self.stack)
```

In the first line of the class definition, the statement `class Stack(object)` declares `Stack` to be an `object`. The use of parentheses is how Python specifies inheritance—in this case, `Stack` inherits from `object`, which is the root of all Python types. Inside the class definition, methods are defined using the `def` statement. The first argument in each method always refers to the object itself. By convention, `self` is the name used for this argument. All operations involving the attributes of an object must explicitly refer to the `self` variable. Methods with leading and trailing double underscores are special methods. For example, `__init__` is used to initialize an object after it's created.

To use a class, write code such as the following:

```
s = Stack()           # Create a stack
s.push("Dave")        # Push some things onto it
s.push(42)
s.push([3,4,5])
x = s.pop()           # x gets [3,4,5]
y = s.pop()           # y gets 42
del s                 # Destroy s
```

In this example, an entirely new object was created to implement the stack. However, a stack is almost identical to the built-in list object. Therefore, an alternative approach would be to inherit from `list` and add an extra method:

```
class Stack(list):
        # Add push() method for stack interface
        # Note: lists already provide a pop() method.
        def push(self,object):
                self.append(object)
```

Normally, all of the methods defined within a class apply only to instances of that class (that is, the objects that are created). However, different kinds of methods can be defined, such as static methods familiar to C++ and Java programmers. For example:

```
class EventHandler(object):
        @staticmethod
        def dispatcherThread():
               while (1):
                        # Wait for requests
                     ...

EventHandler.dispatcherThread()          # Call method as a function
```

In this case, `@staticmethod` declares the method that follows to be a static method. `@staticmethod` is actually an example of using an object known as a decorator—a topic that is discussed further in the chapter on functions and functional programming.

Exceptions

If an error occurs in your program, an exception is raised and an error message such as the following appears:

```
Traceback (innermost last):
 File "<interactive input>", line 42, in foo.py
NameError: a
```

The error message indicates the type of error that occurred, along with its location. Normally, errors cause a program to terminate. However, you can catch and handle exceptions using the `try` and `except` statements, like this:

```
try:
    f = open("file.txt","r")
except IOError, e:
    print e
```

If an `IOError` occurs, details concerning the cause of the error are placed in `e` and control passes to the code in the `except` block. If some other kind of exception is raised, it's passed to the enclosing code block (if any). If no errors occur, the code in the `except` block is ignored. When an exception is handled, program execution resumes with the statement that immediately follows the `except` block. The program does not return back to the location where the exception occurred.

The `raise` statement is used to signal an exception. When raising an exception, you can use one of the built-in exceptions, like this:

```
raise RuntimeError, "Unrecoverable error"
```

Or you can create your own exceptions, as described in the section "Defining New Exceptions" in Chapter 5, "Control Flow."

Modules

As your programs grow in size, you'll probably want to break them into multiple files for easier maintenance. To do this, Python allows you to put definitions in a file and use them as a module that can be imported into other programs and scripts. To create a module, put the relevant statements and definitions into a file that has the same name as the module. (Note that the file must have a `.py` suffix.) Here's an example:

```
# file :  div.py
def divide(a,b):
    q = a//b        # If a and b are integers, q is an integer
    r = a - q*b
    return (q,r)
```

To use your module in other programs, you can use the `import` statement:

```
import div
a, b = div.divide(2305, 29)
```

The `import` statement creates a new namespace that contains all the objects defined in the module. To access this namespace, simply use the name of the module as a prefix, as in `div.divide()` in the preceding example.

If you want to import a module using a different name, supply the `import` statement with an optional `as` qualifier, as follows:

```
import div as foo
a,b = foo.divide(2305,29)
```

To import specific definitions into the current namespace, use the `from` statement:

```
from div import divide
a,b = divide(2305,29)       # No longer need the div prefix
```

To load all of a module's contents into the current namespace, you can also use the following:

```
from div import *
```

Finally, the `dir()` function lists the contents of a module and is a useful tool for interactive experimentation, because it can be used to provide a list of available functions and variables:

```
>>> import string
>>> dir(string)
['__builtins__', '__doc__', '__file__', '__name__', '_idmap',
 '_idmapL', '_lower', '_swapcase', '_upper', 'atof', 'atof_error',
 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
...
>>>
```

Getting Help

When working with Python, you have several sources of quickly available information. First, when Python is running in interactive mode, you can use the `help()` command to get information about built-in modules and other aspects of Python. Simply type **`help()`** by itself for general information or **`help('`*`modulename`*`')`** for information about a specific module. The `help()` command can also be used to return information about specific functions if you supply a function name.

Most Python functions have documentation strings that describe their usage. To print the doc string, simply print the `__doc__` attribute. Here's an example:

```
>>> print issubclass.__doc__
issubclass(C, B) -> bool

Return whether class C is a subclass (i.e., a derived class) of class B.
When using a tuple as the second argument issubclass(X, (A, B, ...)),
is a shortcut for issubclass(X, A) or issubclass(X, B) or ... (etc.).
>>>
```

Last, but not least, most Python installations also include the command `pydoc`, which can be used to return documentation about Python modules. Simply type **`pydoc`** ***`topic`*** at a command prompt (for example, in the Unix command shell).

2

Lexical Conventions and Syntax

THIS CHAPTER DESCRIBES THE SYNTACTIC AND LEXICAL CONVENTIONS of a Python program. Topics include line structure, grouping of statements, reserved words, literals, operators, tokens, and source code encoding. In addition, the use of Unicode string literals is described in detail.

Line Structure and Indentation

Each statement in a program is terminated with a newline. Long statements can span multiple lines by using the line-continuation character (\), as shown in the following example:

```
a = math.cos(3*(x-n)) + \
    math.sin(3*(y-n))
```

You don't need the line-continuation character when the definition of a triple-quoted string, list, tuple, or dictionary spans multiple lines. More generally, any part of a program enclosed in parentheses `(...)`, brackets `[...]`, braces `{...}`, or triple quotes can span multiple lines without use of the line-continuation character because they denote the start and end of a definition.

Indentation is used to denote different blocks of code, such as the bodies of functions, conditionals, loops, and classes. The amount of indentation used for the first statement of a block is arbitrary, but the indentation of the entire block must be consistent. For example:

```
if a:
   statement1     # Consistent indentation
   statement2
else:
   statement3
     statement4   # Inconsistent indentation (error)
```

If the body of a function, conditional, loop, or class is short and contains only a few statements, they can be placed on the same line, like this:

```
if a:  statement1
else:  statement2
```

To denote an empty body or block, use the `pass` statement. For example:

```
if a:
   pass
else:
   statements
```

Although tabs can be used for indentation, this practice is discouraged. The use of spaces is universally preferred (and encouraged) by the Python programming community. When tab characters are encountered, they're converted into the number of spaces required to move to the next column that's a multiple of 8 (for example, a tab appearing in column 11 inserts enough spaces to move to column 16). Running Python with the `-t` option prints warning messages when tabs and spaces are mixed inconsistently within the same program block. The `-tt` option turns these warning messages into `TabError` exceptions.

To place more than one statement on a line, separate the statements with a semicolon (`;`). A line containing a single statement can also be terminated by a semicolon, although this is unnecessary and considered poor style.

The `#` character denotes a comment that extends to the end of the line. A `#` appearing inside a quoted string doesn't start a comment, however.

Finally, the interpreter ignores all blank lines except when running in interactive mode. In this case, a blank line signals the end of input when typing a statement that spans multiple lines.

Identifiers and Reserved Words

An *identifier* is a name used to identify variables, functions, classes, modules, and other objects. Identifiers can include letters, numbers, and the underscore character (_), but must always start with a nonnumeric character. Letters are currently confined to the characters A–Z and a–z in the ISO-Latin character set. Because identifiers are case sensitive, `FOO` is different from `foo`. Special symbols such as `$`, `%`, and `@` are not allowed in identifiers. In addition, words such as `if`, `else`, and `for` are reserved and cannot be used as identifier names. The following list shows all the reserved words:

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

Identifiers starting or ending with underscores often have special meanings. For example, identifiers starting with a single underscore such as `_foo` are not imported by the `from module import *` statement. Identifiers with leading and trailing double underscores such as `__init__` are reserved for special methods, and identifiers with leading double underscores such as `__bar` are used to implement private class members, as described in Chapter 7, "Classes and Object-Oriented Programming." General-purpose use of similar identifiers should be avoided.

Literals

There are five built-in numeric types:

- Booleans
- Integers
- Long integers
- Floating-point numbers
- Complex numbers

The identifiers `True` and `False` are interpreted as Boolean values with the integer values of 0 and 1, respectively. A number such as `1234` is interpreted as a decimal integer. To specify an octal or hexadecimal integer, precede the value with `0` or `0x`, respectively (for example, `0644` or `0x100fea8`). Long integers are typically written with a trailing `l` (ell) or `L` character, as in `1234567890L`. Unlike integers, which are limited by machine precision, long integers can be of any length (up to the maximum memory of the machine). Although the trailing `L` is used to denote long integers, it may be omitted. In this case, a large integer value will automatically be converted into a long integer if it exceeds the precision of the standard integer type. Numbers such as `123.34` and `1.2334e+02` are interpreted as floating-point numbers. An integer or floating-point number with a trailing `j` or `J`, such as `12.34J`, is a complex number. You can create complex numbers with real and imaginary parts by adding a real number and an imaginary number, as in `1.2 + 12.34J`.

Python currently supports two types of string literals:

- 8-bit character data (ASCII)
- Unicode (16-bit-wide character data)

The most commonly used string type is 8-bit character data, because of its use in representing characters from the ASCII or ISO-Latin character set as well as representing raw binary data as a sequence of bytes. By default, 8-bit string literals are defined by enclosing text in single (`'`), double (`"`), or triple (`'''` or `"""`) quotes. You must use the same type of quote to start and terminate a string. The backslash (`\`) character is used to escape special characters such as newlines, the backslash itself, quotes, and nonprinting characters. Table 2.1 shows the accepted escape codes. Unrecognized escape sequences are left in the string unmodified and include the leading backslash. Furthermore, it's legal for strings to contain embedded null bytes and binary data. Triple-quoted strings can span multiple lines and include unescaped newlines and quotes.

Table 2.1 **Standard Character Escape Codes**

Character	Description
`\`	Newline continuation
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote

Table 2.1 **Continued**

Character	Description
\a	Bell
\b	Backspace
\e	Escape
\0	Null
\n	Line feed
\v	Vertical tab
\t	Horizontal tab
\r	Carriage return
\f	Form feed
\OOO	Octal value (\000 to \377)
\xhh	Hexadecimal value (\x00 to \xff)

Unicode strings are used to represent multibyte international character sets and allow for 65,536 unique characters. Unicode string literals are defined by preceding an ordinary string literal with a `u` or `U`, such as in `u"hello"`. In Unicode, each character is internally represented by a 16-bit integer value. For the purposes of notation, this value is written as `U+`*`XXXX`*, where *`XXXX`* is a four-digit hexadecimal number. (Note that this notation is only a convention used to describe Unicode characters and is not Python syntax.) For example, `U+0068` is the Unicode character for the letter *h* in the Latin-1 character set. When Unicode string literals are defined, standard characters and escape codes are directly mapped as Unicode ordinals in the range [U+0000, U+00FF]. For example, the string `"hello\n"` is mapped to the sequence of ASCII values 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0a, whereas the Unicode string `u"hello\n"` is mapped to the sequence U+0068, U+0065, U+006C, U+006C, U+006F, U+000A. Arbitrary Unicode characters are defined using the `\u`*`XXXX`* escape sequence. This sequence can only appear inside a Unicode string literal and must always specify a four-digit hexadecimal value. For example:

```
s = u"\u0068\u0065\u006c\u006c\u006f\u000a"
```

In older versions of Python, the `\x`*`XXXX`* escape sequence could be used to define Unicode characters. Although this is still allowed, the `\u`*`XXXX`* sequence should be used instead. In addition, the `\OOO` octal escape sequence can be used to define Unicode characters in the range [U+0000, U+01FF]. If you know the standard Unicode name for a character (consult http://www.unicode.org/charts for reference), it can be included using the special `\N{`*`character name`*`}` escape sequence. For example:

```
s = u"M\N{LATIN SMALL LETTER U WITH DIAERESIS}ller"
```

Unicode string literals should not be defined using a sequence of raw bytes that correspond to a multibyte Unicode data encoding such as UTF-8 or UTF-16. For example, writing a raw UTF-8 encoded string such as `u'M\303\274ller'` produces the seven-character Unicode sequence U+004D, U+00C3, U+00BC, U+006C, U+006C, U+0065, U+0072, which is probably not what you want. This is because in UTF-8, the multibyte sequence `\303\274` is supposed to represent the single character U+00FC,

not the two characters U+00C3 and U+00BC. However, Python programs can specify a source code encoding that allows UTF-8, UTF-16, and other encoded strings to appear directly in the source code. This is described in the "Source Code Encoding" section at the end of this chapter. For more details about Unicode encodings, see Chapter 3, "Types and Objects" Chapter 4, "Operators and Expressions," and Chapter 9, "Input and Output."

Optionally, you can precede a string with an `r` or `R`, such as in `r'\n\"'`. These strings are known as *raw strings* because all their backslash characters are left intact—that is, the string literally contains the enclosed text, including the backslashes. Raw strings cannot end in a single backslash, such as `r"\"`. When raw Unicode strings are defined, `\u`*XXXX* escape sequences are still interpreted as Unicode characters, provided that the number of preceding `\` characters is odd. For instance, `ur"\u1234"` defines a raw Unicode string with the character U+1234, whereas `ur"\\u1234"` defines a seven-character Unicode string in which the first two characters are slashes and the remaining five characters are the literal `"u1234"`. Also, when defining raw Unicode string literals the "r" must appear after the "u" as shown.

Adjacent strings (separated by whitespace or a newline) such as `"hello" 'world'` are concatenated to form a single string: `"helloworld"`. String concatenation works with any mix of ordinary, raw, and Unicode strings. However, whenever one of the strings is Unicode, the final result is always coerced to Unicode. Therefore, `"hello" u"world"` is the same as `u"hello" + u"world"`. In addition, due to subtle implementation aspects of Unicode, writing `"s1" u"s2"` may produce a result that's different from writing `u"s1s2"`. The details of this coercion process are described further in Chapter 4.

If Python is run with the `-U` command-line option, all string literals are interpreted as Unicode.

Values enclosed in square brackets `[...]`, parentheses `(...)`, and braces `{...}` denote lists, tuples, and dictionaries, respectively, as in the following example:

```
a = [ 1, 3.4, 'hello' ]           # A list
b = ( 10, 20, 30 )                # A tuple
c = { 'a': 3, 'b':42 }            # A dictionary
```

Operators, Delimiters, and Special Symbols

The following operators are recognized:

```
+     -     *     **    /     //    %     <<    >>    &     |
^     ~     <      >    <=    >=    ==    !=    <>    +=
-=    *=    /=    //=   %=    **=   &=    |=    ^=    >>=   <<=
```

The following tokens serve as delimiters for expressions, lists, dictionaries, and various parts of a statement:

```
(     )     [     ]     {     }     ,     :     .     `     =     ;
```

For example, the equal (=) character serves as a delimiter between the name and value of an assignment, whereas the comma (,) character is used to delimit arguments to a function, elements in lists and tuples, and so on. The period (.) is also used in floating-point numbers and in the ellipsis (...) used in extended slicing operations.

Finally, the following special symbols are also used:

```
'     "     #     \     @
```

The characters $ and ? have no meaning in Python and cannot appear in a program except inside a quoted string literal.

Documentation Strings

If the first statement of a module, class, or function definition is a string, that string becomes a documentation string for the associated object, as in the following example:

```
def fact(n):
    "This function computes a factorial"
    if (n <= 1): return 1
    else: return n*fact(n-1)
```

Code-browsing and documentation-generation tools sometimes use documentation strings. The strings are accessible in the __doc__ attribute of an object, as shown here:

```
>>> print fact.__doc__
This function computes a factorial
>>>
```

The indentation of the documentation string must be consistent with all the other statements in a definition.

Decorators

Any function or method may be preceded by a special symbol known as a decorator, the purpose of which is to modify the behavior of the definition that follows. Decorators are denoted with the @ symbol and must be placed on a separate line immediately before the corresponding function or method. For example:

```
class Foo(object):
    @staticmethod
    def bar():
         pass
```

More than one decorator can be used, but each one must be on a separate line. For example:

```
@foo
@bar
def spam():
    pass
```

More information about decorators can be found in Chapter 6, "Functions and Functional Programming," and Chapter 7, "Classes and Object-Oriented Programming."

Source Code Encoding

Python source programs are normally written in standard 7-bit ASCII. However, users working in Unicode environments may find this awkward—especially if they must write a lot of string literals.

It is possible to write Python source code in a different encoding by including a special comment in the first or second line of a Python program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

name = u'M\303\274ller'  # String in quotes is directly encoded in UTF-8.
```

When the special `coding:` comment is supplied, Unicode string literals may be specified directly in the specified encoding (using a Unicode-aware editor program for instance). However, other elements of Python, including identifier names and reserved words, are still restricted to ASCII characters.