1

Reviewing Core Python

WHAT YOU WILL LEARN IN THIS CHAPTER:

- The basic features of the Python language
- How to use the Python module mechanism
- How to create a new module
- How to create a new package

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com downloads for this chapter at `www.wrox.com/go/pythonprojects` on the Download Code tab. The code is in the Chapter 1 download and individually named according to the names throughout the chapter.

This chapter starts with a brief review of Python—in case you have forgotten some of the basics—and provides a foundation upon which the rest of the book is built. If you are confident in your ability with basic Python coding, feel free to skip ahead until you see content that might be of interest to you. After all, you can always come back to this chapter later if you find you need a refresher.

In this chapter you start off by looking at the Python ecosystem, the data types, and the major control structures and then move on to defining functions and classes. Next, you look at the Python module and package system. And, finally, you create a basic new package of modules.

By the end of this chapter, you should be ready to take the next step and start working with the standard Python modules on real project tasks.

EXPLORING THE PYTHON LANGUAGE AND THE INTERPRETER

Python is a dynamic but strictly typed programming language. It is both interpreted and compiled in that the original source code is compiled into byte code and then interpreted, but this happens transparently to the user; you do not have to explicitly ask Python to compile your code.

The Python language has several implementations, but the most common is the version written in C, often referred to as *CPython*. Other implementations include Jython, written in Java, and IronPython, written for the Microsoft .NET platform. CPython is the implementation used in this book.

> ***NOTE*** *At the time of writing, there are two version streams of Python: versions 2.x and 3.x. This book focuses on version 3, and the code has been tested on several releases within that stream—up to release 3.4. Where major compatibility issues arise with 2.x, reference will be made to version 2.7.*

Python programs are written in text files that customarily have the extension `.py`. The Python interpreter, called *python* (in lowercase) does not actually care about the extension; it is only for the user's benefit (and in some operating systems to allow the file and interpreter to be linked).

You can also input Python code directly to the interpreter. This method makes for a highly interactive development style where ideas are prototyped or tested in the interpreter and then transferred into a code editor. The Python interpreter is a powerful learning tool when you are starting to use a new concept or code module.

When working in this mode, you start the interpreter by typing **`python`** at an operating system command prompt. The system will respond with a message telling you the Python version and some build details, followed by the interactive prompt at which you type code. It looks like this:

```
ActivePython 3.3.2.0 (ActiveState Software Inc.) based on
Python 3.3.2 (default, Sep 16 2013, 23:10:06) [MSC v.1600 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This message says that this interpreter is for version 3.3.2.0 of Python, it is the ActiveState distribution (as opposed to the `python.org` distribution), and it was built for 32-bit Windows. Your message may differ slightly, but should contain the same types of information.

If instead of running the Python interpreter interactively you want to execute a program stored in a file, then at the operating system prompt you simply append the name of the file after the `python` command:

```
$ python myscript.py
```

> ***NOTE*** *Usually you can also double-click the file in your file explorer tool, and the operating system makes the connection to python and runs the program automatically. However, this often results in the program opening in a window, completing, and the window closing again before you can see the results, so you may prefer to type the python* `filename` *command in full at a command-line prompt.*

Python comes with two helpful functions that assist you in exploring the language: `dir(name)` and `help(name)`. `dir(name)` tells you all of the names available in the object identified by `name`. `help(name)` displays information about the object called `name`. When you first import a new module, you will often not know what functions or classes are included. By looking at the `dir()` listing of the module, you can see what is available. You can then use `help()` on any of the features listed. Be sure to experiment with these functions; they are an invaluable source of information.

REVIEWING THE PYTHON DATA TYPES

Python supports many powerful data types. Superficially, these look like their counterparts in other programming languages, but in Python they often come with super powers. Everything in Python is an object and, therefore, has methods. This means that you can perform a host of operations on any variable. The built-in `dir()` and `help()` functions will reveal all. In this section you look at the standard data types and their most important operations.

> ***TIP*** *The Python Reference Manual (*`http://docs.python.org/3.3/reference/`*) provides the full detail should you need it.*

You need to be aware of some underlying concepts in Python. First, Python variables are just names. You create variable names by assigning them to objects that are instances of types. Variables do not, of themselves, have a type; it is the object to which they are bound that has a type. The name is just a label and, as such, it can be reassigned to a completely different object. Assignment is performed using `=`, so assigning a value to a variable looks like this:

```
aVariable = aValue
```

This code binds the value `aValue` to the variable name `aVariable` and, if the name does not already exist, the interpreter adds the name to the appropriate namespace.

The distinction between a variable and its underlying value (an object) is thus crucial in Python. You can test variables for equality using a double equal sign (`==`) and object identity (that is, if two names refer to the same object) is compared using the `is` operator, as shown:

```
>>> aString = 'I love spam'
>>> anotherString = 'I love spam'
```

```
>>> anInt = 6
>>> intAlias = anInt
>>> aString == anotherString    # test equality of value
True
>>> aString is anotherString    # test object identity
False
>>> anInt == intAlias           # same value
True
>>> anInt is intAlias           # also same object identity
True
```

Python groups types according to how you can use them. For example, all types are either categorized as *mutable* or *immutable*. If a type is immutable, it means you can't change an object of that type once it's created. You can create a new data item and assign it to the same variable, but you cannot change the original immutable value.

Python also supports several collection types, sometimes referred to as *sequences*. (Strictly speaking collections are a subset of sequences, the distinction will be made clearer later in the chapter.) Sequences share a common set of operations, although not all sequences support all of the operations.

Some Python data types are *callable*. That means you can use the type name like a function to produce a new instance of the type. If no value is given, a default value is returned. You will see examples of this behavior in the following descriptions of the individual data types.

Now that you understand the basics of working with Python data types, it's time to take a look at the different data types, including the numeric, boolean, and None types, as well as the various collection types.

Numeric Types: Integer and Float

Python supports several numeric types including the most basic forms: integer and floating point.

Python integers are unusual in that they are theoretically of infinite size. In practice, integers are limited by the size of your computer's memory. Integers support all the usual numeric operations, such as addition, subtraction, multiplication and so on. You perform arithmetic operations using traditional infix notation. For example, to add two integers,

```
>>> 5 + 4
9
```

or:

```
>>> result = 12 + 8
>>> print (result)
20
```

Literal integer values are, by default, expressed in decimal. You can use other bases by prefixing the number with a zero and the base's initial. Thus, binary is represented as `0bnnn`, octal as `0onnn`, and hexadecimal as `0xnnn`.

The type of an integer is `int`, and you can use it to create integers from floating-point numbers or numeric string representations such as `'123'`, like this:

```
>>> int(5.0)
5
>>> int('123')
123
```

`int` can also convert from nondecimal bases (covering any base up to 36, not just the usual binary, octal, and hexadecimal) using a second, optional, parameter. To convert a hexadecimal (base 16) string representation to an integer, you can use:

```
>>> intValue = int('AB34',16)
43828
```

Python floating-point numbers are of type `float`. Like `int` you can use `float()` to convert string representations, like `'12.34'` to float, and you can also use it to convert an integer number to a float value. Unlike integers, `float()` cannot handle strings for different bases.

The `float` type also supports the normal arithmetic operations, as well as several rounding options. Python floats are based on the Institute of Electrical & Electronic Engineering (IEEE) standards and have the same ranges as the underlying computer architecture. They also suffer the same levels of imprecision that make comparing float values a risky option. Python provides modules for handling fixed precision decimal numbers (`decimal`) and rational fractions (`fractions`) to help alleviate this issue. Python also natively supports a complex, or imaginary, number type called `complex`. These are all typically used for fairly special purposes, so they are not covered here.

The Boolean Type

Python supports a Boolean type, `bool`, with literal values `True` and `False`. The default value of a `bool` is `False`; that is, `bool()` yields `False`.

Python also supports the concept of *truth-like* values for other types. For example, integers are considered `False` if their value is zero. Anything else is considered `True`. The same applies to float values where `0.0` is `False` and anything else is `True`.

You can convert Boolean values to integers using `int()`, in which case `False` is represented as `0` and `True` as `1`.

The Boolean type has most of the Boolean algebra operations you'd expect, including `and`, `or`, and `not`, but—surprisingly—not `xor`.

> ***NOTE*** *Booleans are implemented as a subclass of integer and so also support a bunch of operations that you might not expect, such as exponentiation. You can type things like* `True**False` *and get a result of 1. You should basically just pretend these "features" don't exist and treat them as an implementation detail; otherwise, your code will become very confusing.*

In addition to the Boolean type, Python also supports bitwise Boolean operations on integers. That is to say that Python treats each corresponding pair of bits within two integers as Boolean values in their own right and applies the corresponding operation to each pair of bits. These operations include bitwise and (`&`), or (`|`), not (`^`) and, this time, xor (`~`), as well as bit shift operators for moving bit patterns left (`<<`) or right (`>>`). You look more closely at these bitwise operations later in the chapter.

The None Type

The `None` type represents a null object. There is only one `None` object in the Python environment, and all references to `None` use that same single instance. This means that equality tests with `None` are usually replaced by an identity test, like so,

```
aVariable is None
```

rather than:

```
aVariable == None
```

`None` is the default return value of a Python function. It is also often used as a place marker or flag for default parameters in functions. `None` is not callable and so cannot be used as a conversion function to convert other types to `None`. `None` is considered to have a Boolean value of `False`.

Collection Types

As already mentioned Python has several types representing different kinds of collections or sequences. These are: strings, bytes, tuples, lists, dictionaries and sets. You will see the similarities and differences in each as they are discussed in the following sections. A standard library module called `collections` provides several other more specialized collection types. You will see occasional references to these in the sections that follow.

> ***NOTE*** *In many of the following discussions, you will see references to operations accepting a collection type. Usually this includes what Python calls* iterables*, which are objects that conform to Python's iteration protocol. In simple terms iterables are objects that you can use in loop constructs. In most cases you will not need to worry about them, but they are described in the Python documentation if you are interested in reading the technical details. A good place to start is:* `https://wiki.python.org/moin/Iterator`.

Several common features apply to all collections, and rather than bore you by repeating them for each type, they are covered here.

You can get the length of any collection in Python by using the built-in `len()` function. It takes a collection object argument and returns the number of elements.

You can access the individual elements of a collection using indexing. You do this by providing an index number (or a valid key value for dictionaries) inside square brackets. Collection indices start

at zero. You can also index backward from the end by using negative indices so that the last item in the collection will have an index of –1.

Whereas indexing is used to access just one particular element of a collection, you can use *slicing* to access multiple items in the collection. Slicing consists of a start index, an end index, and a step size, and the numbers are separated by colons. Slicing is not valid for dictionaries or sets. The step size argument enables you to, for example, select every other element. All values are optional, and the defaults are the start of the collection, the last item in the collection, and a step size of one. The slice returned consists of all (selected) elements from `start` to `end-1`.

Here are a few examples of slicing applied to a string, entered at the Python interactive prompt:

```
>>> '0123456789'[:]
'0123456789'
>>> '0123456789'[3:]
'3456789'
>>> '0123456789'[:3]
'012'
>>> '0123456789'[3:7]
'3456'
>>> '0123456789'[3:7:2]
'35'
>>> '0123456789'[::3]
'0369'
```

You can sort most collections by using the `sorted()` function. The return value is a sorted list containing the original collection elements. Optional arguments to `sorted()` provide flexibility in how the elements are sorted and in what order.

In general, empty collections are treated as `False` in Boolean expressions and `True` otherwise. Two functions, `any()` and `all()`, refine the concept to allow more precise tests. The `any()` function takes a collection as an argument and returns `True` if any member of the collection is true. The `all()` function takes a collection as an argument and returns `True` if—and only if—all the members are true.

Strings

Python strings are essentially collections of Unicode characters. (The implications of using Unicode are discussed in Chapter 4.) The default encoding is UTF8. If you are working in English, most things will work as you expect. Once you start to use non-English characters, things get more interesting! For now you will be working in English and sticking with UTF8.

Python requires that literal strings be enclosed within quotation marks. Python is extremely flexible in this regard and accepts single quotes ('Joe'), double quotes ("Joe"), triple single quotes ('''Joe'''), and triple double quotes ("""Joe""") to delimit a string. Obviously, the start and end quotes must be of the same type, but any other quote can be contained inside the string. This is most useful for apostrophes and similar grammatical cases (‘He said, “Hi!”’ or “My brother’s hat”). Triple quotes of either type can span multiple lines. Here are a few examples:

```
>>> 'using single quotes'
'using single quotes'
>>> "using double quotes"
```

```
'using double quotes'
>>> print('''triple single quotes spanning
... multiple lines ''')
triple single quotes spanning
multiple lines
```

A literal string at the start of a module, class, or function that is not assigned to a variable is treated as documentation and displayed as part of the built-in `help()` output for that object.

Special characters such as tabs (`\t`) or newlines (`\n`) must be prefixed, or quoted, with a backslash character, and literal backslashes must be quoted so they look like double backslashes. You can avoid this by preceding the entire string with the letter `r` (for *raw*) to indicate that special character processing should not be done. Nonprintable characters can be included in a string using a backslash followed by their hex code. For example, the escape character is `\x1A`. (Note that there is no leading zero as is used for hexadecimal integer literals.)

Strings are immutable in that you cannot directly modify or add to a string once it is formed. You can, however, create a new string based on an existing one, and that is how many of the Python string operations work. Python supports a wide range of operations on strings, and these are mostly implemented as methods of the string class. Some of the most common operations are listed in Table 1-1.

Several other string operations are available, but those listed in Table 1-1 are the ones you will use most often.

Empty strings are treated as `False` in Boolean expressions. All other strings are treated as `True`.

Bytes and ByteArrays

Python supports two byte-oriented data types. A byte is an 8-bit value, equivalent to an integer from 0–255, and represents the raw bit patterns stored by the computer or transmitted over a network. They are very similar to strings in use and support many of the same methods. The type names are spelled as `byte` and `bytearray` respectively.

Literal byte strings are represented in quotes preceded by the letter `b`. Byte strings are immutable. Byte arrays are similar, but they are mutable.

In practice you will rarely use byte strings or byte arrays unless handling binary data from a file or network. One issue that can catch you by surprise is that if you access an individual element using indexing, the returned value is an integer. This means that comparing a single character byte string with an indexed string value results in a `False` response, which is different from what would happen using strings in the same way. Here is an example:

```
>>> s = b'Alphabet soup'
>>> c = b'A'
>>> s[0] == c
False
>>> s[0] == c[0]
True
```

As you can see, the key is to use indexing on both sides of the comparison.

TABLE 1-1: String Operations

OPERATION	DESCRIPTION
`+`	Concatenation. This is a somewhat inefficient operation, and you can usually avoid it by using `join()` instead.
`*`	Multiplication. This produces multiple copies of the string concatenated together.
`upper, lower, capitalize`	These change the case of a string.
`center, ljust, rjust`	These justify the string as needed within a given character width, padding as needed with the specified fill character (defaulting to a space).
`startswith, endswith`	These test a substring matching the start or end of a line. Optional parameters control the actual subsection of the string that is tested so the names are slightly misleading. They can also test multiple substrings at once if they are passed as a tuple.
`find, index, rfind`	These return the lowest index where the given substring is found. `find` returns `-1` on failure whereas `index` raises a `ValueError` exception. `rfind` starts at the right-hand side and, therefore, returns the highest index where the substring is found.
`isalpha, isdigit,isalnum, and so on.`	These test the string content. Several test types exist, the most commonly used being those listed; for alphabetic, numeric, and alphanumeric characters respectively.
`join`	This joins a list of strings using the active string as the separator. It is common to build a string using either a single space or no space as the separator. This is faster and more memory efficient than using string concatenation.
`split, splitlines, partition`	These split a string into a list of substrings based on a given separator (the default is whitespace). Note that the separators are removed in the process. `splitlines()` returns a list of lines, effectively splitting using the newline character. `partition()` splits a string based on the given separator, but only up to the first occurrence; it then returns the first string, the separator, and the remaining string.
`strip, lstrip,rstrip`	These strip whitespace (the default) or the specified characters from the ends of the string. `lstrip` strips only the left side; `rstrip` strips only the right. None of them remove whitespace from the middle of a string; they only remove outer characters. To globally remove characters, use the `replace` operation.
`replace`	This performs string replacement. By specifying an empty string as the replacement, it can be used to effectively delete characters.
`format`	This replaces the older C printf-style string formatting used in Python version 2. Printf style is still available in version 3, but is deprecated in favor of `format()`. String formatting is explained in detail in the Python documentation. The basic concept is that pairs of braces embedded in the string form placeholders for data passed as arguments to `format()`. The braces can contain optional style information, such as padding characters. (You can find examples throughout this book.)

You can use the `struct` module to convert binary data from the bytes representation to normal Python types. Of course, to use this you will still have to know what types the byte patterns represent in the first place.

Empty byte strings are treated as `False` in Boolean expressions. All other byte strings are treated as `True`.

Tuples

Tuples are collections of arbitrary objects. The fact that they are collected together suggests that there is probably a logical connection between them, but the language puts no restriction on the objects contained. Tuples are often described as being the Python equivalent to records, or structs, in other languages.

Literal tuples consist of a series of values (or variables) separated by commas. Often, to prevent syntax ambiguity, the tuple as a whole will be contained in parentheses, but this is not a requirement of the tuple itself.

Tuples are immutable, so you cannot modify or extend the tuple once it is created. You can create a new tuple based on an existing one in the same way you did for strings, and you can create a new empty tuple using the `tuple()` type function. You can use a tuple as a key in a dictionary because they are immutable.

One feature of Python tuples that is very useful is known as *unpacking*. This enables you to extract the values of a tuple into discrete variables. You most often see this when a function returns a tuple of values and you want to store the individual values in separate variables. An example is shown here using the `divmod()` function, which returns the quotient and remainder of an integer division as a tuple:

```
>>> print(divmod(12,7))
(1, 5)
>>> q,r = divmod(12,7)
>>> print (q)
1
>>> print (r)
5
```

Notice how `q` and `r` can be treated as new, single-valued variables.

A `namedtuple` class in the `collections` module allows elements to be indexed by name rather than position. This combines some of the advantages of a dictionary with the compactness and immutability of a tuple.

Empty tuples are treated as `False` in Boolean expressions. All other tuples are treated as `True`.

Lists

Lists in Python are highly flexible and powerful data structures. They can be used to mimic the behavior of many classic data structures and to form the basis of others in the form of custom classes. They are dynamic and, like tuples, can hold any kind of object but, unlike tuples, they are mutable, so you can modify their contents. You can also use tuple-style unpacking to assign list items to discrete variables.

A literal list is expressed as a comma-separated sequence of objects enclosed in square brackets. You can create an empty list either by specifying a pair of empty square brackets or by using the default value of the `list()` type function. Lists come with several methods for adding and removing members, and they also support some arithmetic-style operations for concatenating and copying lists in a similar fashion to strings.

You can initialize a list directly by using lists of values, or you can build them programmatically using list comprehensions. The latter looks like a single-line `for` loop inside list brackets. Here is an example that builds a list of the even squares from 1 to 10:

```
>>> [n*n for n in range(1,11) if not n*n % 2]
[4, 16, 36, 64, 100]
```

Table 1-2 lists some of the most common list operations.

Empty lists are treated as `False` in Boolean expressions. All other lists are treated as `True`.

TABLE 1-2: List operations

OPERATION	DESCRIPTION
`+`	This concatenates two lists.
`*`	This creates a list of multiple copies of the first list. Note that the copies all refer to the same object, which often results in surprising side effects when an object is modified. Often, list slicing or list comprehension is a better option.
`append`	This adds an element to the end of an existing list. The new element could itself be a list. The operation is effective in-place, and `None` is returned.
`extend`	This adds the contents of a list to the end of another list, effectively joining the two lists. The original list is modified in place. `None` is returned.
`pop`	This removes an item from the end of a list or at the specified index if one is provided. Returns the item.
`index`	This returns the index of the first occurrence of an item in a list. Raise a `ValueError` if not found. (The string operation of the same name has similar behavior.)
`count`	This returns a count of the specified item in the list.
`insert`	This inserts an element before the specified index. If the index is too large, it is appended to the end of the list.
`remove`	This removes the first occurrence of the specified item. It raises a `ValueError` if the item does not exist.
`reverse`	This reverses the elements of the list in-place.
`sort`	This sorts the elements of the list in-place. Optional parameters provide flexibility in how the sort is performed. To get a sorted copy of the list without modifying the original, use the `sorted()` function.

Dictionaries

Dictionaries are super powerful data structures that beginners frequently overlook. They offer solutions to lots of common programming problems. A dictionary is used like a list, but its elements are accessed by using a key-value mechanism rather than a numeric index. The elements of a Python dictionary are thus a (non-ordered) sequence of key-value pairs.

The key can be any immutable value, including a tuple. Keys must be unique. The value can be any kind of Python object, including another dictionary, a list, or anything else that Python recognizes as an object.

Dictionaries are highly optimized, so lookup times are very fast. In fact Python makes extensive use of dictionaries internally, to implement namespaces and classes, among other things. Dictionaries also provide a solution anywhere that dynamically named values need to be stored and accessed. Dictionaries are also efficient where the keys are not sequential because Python uses a hashing algorithm to map the keys into a sparse array structure. (If you didn't understand that last sentence, don't worry, you are not alone, and you don't really need to know. What it means for you is that Python dictionaries are fast, and they use memory efficiently.)

Dictionary literals are composed of comma-separated key-value pairs. The keys and their values are separated by colons, and the whole is contained in a pair of curly braces, or {}. It looks like this:

```
>>> {'aKey':'avalue', 2:7, 'booleans':{False:0,True:1}}
{'aKey': 'avalue', 2: 7, 'booleans': {False: 0, True: 1}}
```

You access the stored values by "indexing" the dictionary using the key rather than a numeric index. If the preceding example were stored in a variable called `D`, you could access the `aKey` and the `True` values like this:

```
>>> D['aKey']
'avalue'
>>> D['booleans'][True]
1
```

You can create an empty dictionary by using an empty pair of braces or by using the default value of the `dict()` type function.

Dictionaries come with some extra operations for extracting lists of keys and values and handling default values. Some of these are described in Table 1-3.

Dictionaries are, by the nature of their implementation, unsorted. Indeed the order may change when new data is inserted. The `collections` module contains an `OrderedDict` class that maintains the order of insertion should that be required. The `sorted()` function returns a sorted list of keys if the keys are comparable. If the keys are incompatible (as in the preceding example), `sorted()` raises a `TypeError`.

The `collections` module also provides a `defaultdict` class that enables you to specify a default value that is returned any time a nonexistent key is used. In addition to returning the default value, it also creates a new element for the given key with the default value. This is similar to the `setdefault` method described earlier. This can be a mixed blessing because it can result in bogus entries for badly spelled keys!

TABLE 1-3: Dictionary Operations

OPERATION	DESCRIPTION
`keys, values, items`	These methods return list-like objects (called *dictionary views*) containing the keys, values, and key-value tuples respectively. These views are dynamic, so any changes to the dictionary (deletions and so on) after they are created are reflected in the view.
`get, pop`	These methods take a key and an optional default value. `get` returns the value for a key from the dictionary if the key exists or a specified default if the key does not exist. `pop` works the same way, but also removes the item from the dictionary if it exists. `get` has a default value of `None`, but `pop` raises a `KeyError` if no default is given.
`setdefault`	This operation acts like `get`, but also creates a new key-value pair in the dictionary using the given key and the default value if the key does not exist.
`fromkeys`	This operation initializes a dictionary using a sequence to provide the keys and a given default value (or `None` if no value is given). Usually called directly as `dict.fromkeys()` rather than on an existing dictionary.

Empty dictionaries are treated as `False` in Boolean expressions. All other dictionaries are treated as `True`.

Sets

Sets embody a mathematical concept that is frequently used in programming where a unique group of elements is required. In Python, sets are a lot like dictionaries with keys but no corresponding values.

Sets have the Python type `set`. The same basic rules apply as for dictionary keys in that set values must be immutable and unique (indeed that's what makes it a set!). The default value from `set()` is the empty set, and this is the only way to express an empty set because {} is already used for an empty dictionary. The `set()` function accepts any kind of collection as its argument and converts it to a set (dictionary values will be lost).

There is another type of set in Python, called `frozenset`, that is immutable and is basically a read-only set. Its constructor works like `set()`, but it supports only a subset of the operations. You can use a `frozenset` as an element of a regular set because `frozenset` is immutable.

Set literals are expressed with curly braces surrounding elements separated by commas:

```
myset = {1,2,3,4,5}
```

Sets do not support indexing or slicing and, like dictionaries, they do not have any inherent order. The `sorted()` function returns an ordered list of values. Sets have a bunch of math-style set operations that are distinct from other collections. Table 1-4 shows those operations that are common to both the `set` and `frozenset` types.

TABLE 1-4: Set Operations

OPERATION	DESCRIPTION
`in`	This tests whether a single element exists in the set. Note: If the tested item is itself a set, S1, the result will be true only if S1, as a set, is an element of the target set. This is different from the `subset()` test.
`issubset, <=, <`	These test whether one set is a subset of the target—that is, whether all members of the tested set are also members of the target. If the sets are identical, the test will return `True` in the first two cases, but `False` for the `<` operator.
`issuperset, >=, >`	These test whether one set is a superset of the other—that is, whether all the members of the target set are members of the source. If they are equal, the first two operations return `True`; the last operation will return `False`.
`union, |`	These return the union of two (or more) sets. The union method takes a comma-separated list of sets as arguments while the pipe operator is used infixed between the sets.
`intersection, &`	These return the intersection set of two (or more) sets. Usage is similar to `union()`.
`difference, -`	These return the members of the source set that are not members of the target set.
`symmetric_difference, ^`	Returns the elements of both sets that are not in the intersection. The method works only for two sets, but the ^ operator can be applied to multiple sets in infix style.

Note that the method variations in Table 1-4 will accept any collection type as an argument whereas the infix operations will work only with sets.

Table 1-5 looks at the modifier operations that are applicable only to sets. These cannot be used on a `frozenset`, although a `frozenset` can be used as an argument for several of them. Note that these operations adjust the source set itself; they do not return a set—they return the Python default value of `None`. The infix operations work only on two sets (unlike those in Table 1-4) and work only on actual sets, not other collection types. You can use the methods on multiple sets, and other collection types are converted to a set where needed.

Empty sets are treated as `False` in Boolean expressions. All other sets are treated as `True`.

In the next section, you will use the data types in code as you explore the different control structures that Python offers.

TABLE 1-5: Set Modifier Operations

OPERATION	DESCRIPTION
`update, |=`	These add the elements of the target set (or sets) to the source set.
`intersection_update, &=`	These remove all elements except those in the intersection of source and target sets. If more than two sets are involved, the result is the intersection of all sets involved.
`difference_update, -=`	These remove all elements found in the intersection of the sets. If multiple sets are provided, the items removed are in the intersection of the source with any of the other sets.
`symmetric_difference_update, ^=`	These return the set of values in both sets involved excepting those in the intersection. Note that this operation works only on two sets at a time.
`add`	This adds the given element to the set.
`remove`	This removes the specified element from the set. If the element is not found it raises a `KeyError`.
`discard`	This removes the given element from the set if it is present. No `KeyError` is raised in this case if the element is not found.
`pop`	This removes and returns an arbitrary member from the set. Raises `KeyError` if the set is empty.
`clear`	This removes all elements from a set.

USING PYTHON CONTROL STRUCTURES

In this section you first look at the overall structure of a Python program and then consider each of the basic structures: sequence, selection, and iteration. Finally, you look at how Python handles errors, review context managers, and investigate how to exchange data with the outside world.

Structuring Your Program

Python programs do not have any required, predefined entry point (for example a `main()` function) and are simply expressed as source code in a text file that is read and executed in order starting at the top of the file. (Definitions, such as functions, are executed in the sense that the function is created and assigned to a name, but the internal code is not executed until the function is called.)

Python does not have any special syntax to indicate whether a source file is a program or a module and, as you will see, a given file can be used in either role. A typical executable program file

consists of a set of import statements to bring in any code modules needed, some function and class definitions, and some directly executable code.

In practice, for a nontrivial program, most function and class definitions will exist in module files and be included in the imports. This leaves a short section of driver code to start the application. Often this code will be placed in a function, and the function will often be called `main()`, but that is purely a nod to programming convention, not a requirement of Python.

Finally this "main" function needs to be called. This is often done within a special `if` statement at the end of the main script. It looks like this:

```
if __name__ == "__main__":
    main()
```

When Python detects that a program file is being executed by the interpreter rather than imported as a module, it sets the special variable `__name__` (note the double underscores on either side) to `"__main__"`. This means that any code inside this `if` block is executed only when the script is run as a main program and not when the file is imported by another program. If the file is only ever expected to be used as a module, the `main()` function may be replaced by a `test()` function that executes a set of unit tests. Again, the actual name used is of no significance to Python.

Using Sequences, Blocks and Comments

The most fundamental programming structure is a sequence of statements. Normally Python statements occur on a line by themselves, so a sequence is simply a series of lines.

```
x = 2
y = 7
z = 9
```

In this case the statements are all assignments. Other valid statements include function calls, module imports or definitions. Definitions include functions and classes. The various control structures described in the following sections are also valid statements.

> ***NOTE*** *Python does enable you to include multiple statements on a single line by separating them with a semicolon. Thus the following line of code consists of three statements:*
>
> ```
> x = 2; y = 7; z = 9
> ```
>
> *This style is not recommended by the Python community; using separate lines is preferred.*

Python is a block-structured language, and blocks of code are indicated by indentation level. The amount of indentation is quite flexible; although most Python programmers stick to using three

or four spaces to optimize readability, Python doesn't care. Different Integrated Development Environments (IDEs) and text editors have their own ideas about how indentation is done. If you use multiple programming tools, you may find you get indentation errors reported because the tools have used different combinations of tabs and spaces. If possible, set up your editor to use spaces instead of tabs.

The exception to the indentation rule is comments. A Python comment starts with a `#` symbol and continues to the end of the line. Python accepts comments that start anywhere on the line regardless of current indentation level, but by convention, programmers tend to retain indentation level, even for comments.

Selecting an Execution Path

Python supports a limited set of selection options. The most basic structure is the `if/elif/else` construct. The `elif` and `else` parts are optional. It looks like this:

```
if pages < 9:
    print("It's too short")
elif pages > 99:
    print("It's too long")
else: print("Perfect")
```

Notice the colon at the end of each test expression. That is Python's indicator that a new block of code is coming up. It has no start and end block markers (such as {}); the colon is the only indication, and the block must either occur on the same line as the colon, if it's only a single line block, or as an indented block of code. Many Python programmers prefer the indented block style even for single line blocks.

Also note that there can be an arbitrary number of `elif` tests, but only a single `else` clause—or none at all.

The other selection structure you will find in Python is the conditional expression selector. This produces one of several values depending on the given test conditions. It looks like this:

```
<a value> if <an expression> else <another value>
```

An example might be where a screen coordinate is being incremented until a certain limit (perhaps the screen's maximum resolution) and then reset to `0`. That could be written as:

```
coord = coord + increment if coord < limit else 0
```

This is equivalent to the more traditional structure shown here:

```
if coord < limit
   coord += increment
else:
   coord = 0
```

You should use caution when using conditional expressions because it is very easy to create obscure code. If in doubt you should use the expanded if/else form.

One final comment on Python's comparison expressions is worth making here. In many programming languages, if you want to test whether a value lies between two limits, you need two separate tests, like this:

```
if aValue < upperLimit and aValue > lowerLimit:
    # do something here
```

Python will be quite happy to process code like that, but it offers a useful shortcut in that you can combine the comparisons as shown here:

```
if lowerLimit < avalue < upperLimit:
   # do something here
```

Iteration

Python offers several alternatives for iteration. The most fundamental and general is the `while` loop. It looks like this:

```
while BooleanExpression:
    aBlockOfCode
else:
    anotherBlock
```

Notice the colon (:) at the end of the `while` statement, which signifies that a block of code follows. Also notice the indentation of the block. The block will, in principle, be executed for as long as `BooleanExpression` remains true. However, there are two ways you can exit out of a `while` loop regardless of the `BooleanExpression` value. These are the `break` statement, which exits the loop immediately, and a `return` statement if the loop is inside a function definition. A `return` statement exits the function immediately and so will also exit any loop within the function.

The `else` clause is optional and is rarely used in practice. It is executed any time the `BooleanExpression` is `False`, including when the loop exits normally. It will not be executed if the loop is exited by a `break` or `return` statement.

One very common `while` loop idiom is to use `True` as the test condition to create an infinite loop and then have a `break` test within the body of the loop. Here is an example where the loop reads user commands and processes them. If the command contains the letter `q`, it exits.

```
while True:
    command = input('Enter a command[rwq]: ')
    if 'q' in command.lower(): break
    if command.lower() == 'r':
       # process 'r'
    elif command.lower() == 'w':
       # process 'w'
    else:
       print('Invalid command, try again')
```

There is a companion statement to `break`, namely `continue`. Whereas `break` exits both the block and the loop, `continue` exits the block for the current loop iteration only. Control then returns to the `while` statement and, if appropriate, a new iteration of the block will commence.

The next significant looping construct in Python is the `for` loop, which looks like this:

```
for item in <iterable>:
    code block
else:
    another code block
```

The `for` loop takes each item in the iterable and executes the code block once per item. You can terminate the loop early using `break` or `return` as described for the `while` loop. You can terminate a single iteration of the loop using `continue` as described earlier.

The `else` block is executed when all the iterations are completed. It will not be executed if the loop exits via a `break` or `return`.

The iterable is anything that complies with Python's iterator protocol. In practice this is usually a collection such as a list, tuple, or a function that returns a collection of values such as `range()`. The `open()` function returns a file iterator that enables you to loop over a file without first reading it into memory. It's possible to define your own custom iterable classes, too.

One common function that is particularly used with `for` loops is `enumerate()`. This function returns tuples containing both the iterable item and a sequence number that, by default, is equivalent to a list index. This means that the `for` block can more easily update the iterable directly. `enumerate()` takes a second optional argument that specifies the sequence starting number, which you could use, for example, to indicate the line number in a file, starting with 1 rather than the 0 default.

Here is an example that illustrates some of these points printing a file with associated line numbers:

```
for number, line in enumerate(open('myfile.txt')):
      print(number, '\t', line)
```

Finally, Python has a couple of inline loop structures. You saw one of these, the list comprehension, in the discussion of lists earlier in the chapter.

A list comprehension is a specific application of a more general loop form known as a *generator expression* that you can use where you might otherwise have a sequence of literal values. If you recall the list comprehension example earlier in the chapter, you used it to populate a list with the even squares from 1 to 10, like this:

```
>>> [n*n for n in range(1,11) if not n*n % 2]
[4, 16, 36, 64, 100]
```

The part inside the square brackets is a generator expression and the general form is as follows:

```
<result expression> for <loop variable> in <iterable> if <filter expression>
```

Comparing that with the list comprehension example you see that the result expression was `n*n`, the loop variable was `n`, and the iterable was `range(1,11)`. The filter expression was `if not n*n % 2`.

You can rewrite that as a conventional `for` loop, like this:

```
result = []
for n in range(1,11):
    if not n*n % 2:
        result.append(n)
```

One important point to appreciate about generator expressions is that they do not generate the whole data set at once. Rather they generate (hence the name) the data items on demand, which can lead to a significant saving in memory resources when dealing with large data sets. You find out more about this later in the chapter when you look at a special type of function called a *generator function*.

Handling Exceptions

There are two approaches to detecting errors. The first involves explicitly checking each action as it's performed; the other attempts the operations and relies on the system generating an error condition, or *exception*, if something goes wrong. Although the first approach is appropriate in some situations, in Python, it's far more common to use the second. Python supports this technique with the `try/except/else/finally` construct. In its general form, it looks like this:

```
try:
     A block of application code
except <an error type> as <anExceptionObject>:
    A block of error handling code
else:
   Another block of application code
finally:
   A block of clean-up code
```

The `except`, `else`, and `finally` are all optional, although at least one of `except` or `finally` must exist if a `try` statement is used. There can be multiple `except` clauses, but only one `else` or `finally`. You can omit the `as…` part of an `except` statement line if the exception details are not required.

The `try` block is executed and, if an error is encountered, the exception class is tested. If an `except` statement exists for that type of error, the corresponding block is executed. (If multiple exception blocks specify the same exception type, only the first matching clause is executed.) If no matching `except` statement is found, the exception is propagated upwards until the top-level interpreter is reached and Python generates its usual traceback error report. Note that an empty `except` statement will catch any error type; however, this is usually a bad idea because it hides the occurrence of any unexpected errors.

The `else` block is executed if the `try` block succeeds without any errors. In practice the `else` is rarely used. Regardless of whether an error is caught or propagated, or whether the `else` clause is executed, the `finally` clause will always be executed, thus providing an opportunity to release any computing resources in a locked state. This is true even when the `try/except` clause is left via a `break` or `return` statement.

You can use a single `except` statement to process multiple exception types. You do this by listing the exception classes in a tuple (parentheses are required). The optional exception object contains details of where the exception occurred and provides a string conversion method so that a meaningful error message may be provided by printing the object.

It is possible to raise exceptions from your own code. It is also possible to use any of the existing exception types or to define your own by subclassing from the `Exception` class. You can also pass arguments to exceptions you raise, and you can access these in the exception object in the `except` clause using the `args` attribute of the error object.

Here is an example of raising a standard `ValueError` with a custom argument and then catching that error and printing the given argument.

```
>>> try:
...     raise ValueError('wrong value')
... except ValueError as error:
...     print (error.args)
...
('wrong value',)
```

Note that you didn't get a full traceback, only the print output from the `except` block. You can also re-raise the original exception after processing by simply calling `raise` with no arguments.

Managing Context

Python has the concept of a runtime *context*. This typically includes a temporary resource of some kind that your program wants to interact with. A typical example might be an open file or a concurrent thread of execution. To handle this Python uses the keyword `with` and a *context manager* protocol. This protocol enables you to define your own context manager classes, but you will mostly use the managers provided by Python.

You use a context manager by invoking the `with` statement:

```
with open(filename, mode) as contextName:
     process file here
```

The context manager ensures the file is closed after use. This is fairly typical of a context manager's role—to ensure that valuable resources are freed after use or that proper sharing precautions are taken on first use. Context managers often remove the need for a `try/finally` construct. The `contextlib` module provides support for building your own context managers.

You have now seen the different types of data that Python can process as well as the control structures you can use to do that processing. It is now time to find out how to get data into and out of your Python programs and that is the subject of the next section.

GETTING DATA IN AND OUT OF PYTHON

Basic input and output of data is a major requirement of any programming language. You need to consider how your programs will interact with users and with data stored in files.

Interacting with Users

To send data to users via `stdout`, you use the `print()` function, which you've seen several times already. You learn how to control the output more precisely in this section. To read data from users, you use the `input()` function, which prompts the user for input and then returns a string of raw characters from `stdin`.

The `print()` function is more complex than it first appears in that it has several optional parameters. At its simplest level, you simply pass a string and `print()` displays it on `stdout`

followed by an end-of-line (`eol`) character. The next level of complexity involves passing non-string data that `print()` then converts to a string (using the `str()` type function) before displaying the result. Stepping up a gear, you can pass multiple items at once to `print()`, and it will convert and display them in turn separated by a space.

The previous paragraph identified three fixed elements in `print()`'s behavior:

- It displays output on `stdout`.
- It terminates with an `eol` character.
- It separates items with a space.

In fact, none of these are really fixed, and `print()` enables you to modify any or all of them using optional parameters. You can change the output by specifying a `file` argument; the separator is defined by the `sep` argument, and the terminating character is defined by the `end` argument. The following line prints the infamous "hello world" message, specified as two strings, to a file using a hyphen separator and the string `"END"` as an end marker:

```
with open("tmp.txt", "w") as tmp:
    print("Hello", "World", end="END", sep="-", file=tmp)
```

The content of the file should be: `"Hello-WorldEND"`.

The string `format()` method really comes into its own when combined with `print()`. This combination is capable of presenting your data neatly and clearly separated. In addition using `format()` can often be more efficient that trying to print a long list of objects and string fragments. There are many examples of how to use `format()` in the Python documentation.

You can also communicate with the user using the `input()` function that reads values typed by the user in response to a given on-screen prompt. It is your responsibility to convert the returned characters to whatever data type is represented and handle any errors resulting from that conversion.

> ***NOTE*** *In Python version 2, the* `raw _ input()` *function was used instead of* `input()`*. The version 2* `input()` *function behaved rather differently. It evaluated whatever the user typed. This created a security issue because malicious code could be input. The version 2* `input()` *function was removed in version 3 and* `raw _ input()` *was renamed to* `input()`*.*

Here is an example that asks the user to enter a number. If the number is too high or too low it prints a warning. (This could form the core of a guessing game if you cared to experiment with it.)

```
target = 66

while True :
    value = input("Enter an integer between 1 and 100")
    try:
```

```
            value = int(value)
            break
        except ValueError:
            print("I said enter an integer!")

if value > target:
    print (value, "is too high")
elif int(value) < target:
    print("too low")
else:
    print("Pefect")
```

Here the user is provided with a prompt to enter an integer in the appropriate range. The value read is then converted to an integer using `int()`. If the conversion fails, a `ValueError` exception is raised, and the error message is then displayed. If the conversion succeeds, you can break out of the `while` loop and proceed to test it against the target, confident that you have a valid integer.

Using Text Files

Text files are the workhorses of programming when it comes to saving data, and Python supports several functions for dealing with text files.

> ***NOTE*** *The file interface in Python is really a specialization of a higher-level abstract interface starting with a class called* `io.IOBase`*. You can mostly ignore these; they simply create a standardized set of operations that applies to text files and other "file like" objects.*

You saw the `open()` function used in previous sections and it takes a filename and a mode as arguments. The mode can be any of `r`, `w`, `rw`, and `a` for read, write, read-write, and append respectively. (Some other modes are less often used. There are also a few optional parameters that control how the data is interpreted, see the documentation for details.) The `r` mode requires the file to exist; the `w` and `rw` modes create a new empty file (or overwrite any existing file of the same name). The `a` mode opens an existing file or creates a new empty file if a file of the specified filename does not already exist. The file object returned is also a context manager and can be used in a `with` block as you saw in the context manager section. If a `with` block is not used, you should explicitly close the file using the `close()` method when you are finished with it, thus ensuring that any data sitting in memory buffers is sent to the physical file on disk. The `with` construct calls `close()` automatically, which is one of the advantages of using the context manager approach.

Once you have an open file object, you can use `read()`, `readlines()`, or `readline()` as required. `read()`reads the entire file contents as a single string complete with embedded newline characters. `readlines()` reads line by line into a list, and the newline characters are retained. `readline()` reads the next line in the file, again retaining the newline. The file object is an iterable, so you can use it

directly in a `for` loop without the need for any of the read methods. The recommended pattern for reading the lines in a file is therefore:

```
with open(filename, mode) as fileobject:
    for line in fileobject:
        # process line
```

You can write to a writable file object using the `write()` or `writelines()` methods, which are the equivalents to the similarly named read methods. Note that there is no `writeline()` method for writing a single line.

If you are using the `rw` mode, you might want to move to a specific position in the file to overwrite the existing data. You can find your current position in the file using the `tell()` method. You can go to a specific position (possibly one you recorded with `tell()` earlier) using the `seek()` method. `seek()` has several modes of calculating position; the default is simply the offset from the start of the file.

You now have all of the basic skills to write working Python programs. However, to tackle larger projects, which are the focus of this book, you will want to extend Python's capabilities. The next section starts to explore how you can do just that.

EXTENDING PYTHON

The simplest way of extending Python is by writing your own functions. You can define these in the same file as the code that uses them, or you can create a new module and import the functions from there. You look at modules in the next section; for now you will create the functions and use them in the same file. In fact, you will mostly be using the interactive prompt for the examples in this section.

The next step in creating new functionality in Python is to define your own classes and create objects from them. Again, it is common to create classes in modules, and you see how to do so in the next section. The examples here are simple enough that you can just use the Python prompt.

Python programmers frequently use documentation strings in their programs. Documentation strings are string literals that are not assigned to a variable and respect the indentation level at which they are defined. You use documentation strings to describe functions, classes, or modules. The `help()` function reads and presents documentation strings.

Defining and Using Functions

Several types of functions are available in Python. This section looks at the standard variety first, followed by a generator function, and concludes with the slightly enigmatic lambda function.

You define functions in Python using the `def` keyword. The form looks like this:

```
def functionName(parameter1, param2,...):
    function block
```

Python functions always return a value. You can specify an explicit return value using the `return` keyword; otherwise, Python returns `None` by default. (If you find unexpected `None` values appearing in your output, check that the function concerned has an explicit `return` statement in its body.) You

can give default values to parameters by following the name with an equals sign and the value. You see an example in the `odds()` generator function in the next section.

You can most easily understand how a function definition is created and used by trying it out.

TRY IT OUT Creating and Using a Function

In this Try It Out, you create a new function that takes several input parameters and returns a value. This function uses the mathematical equation of a straight line to return the corresponding y-coordinate for a given gradient, x-coordinate, and constant. You then use the function to generate a set of coordinates for a line.

1. Start the Python interpreter.
2. Type the following code to define the function:

```
>>> def straight_line(gradient, x, constant):
...     ''' returns y coordinate of a straight line
      -> gradient * x + constant'''
...     return gradient*x + constant
...
>>>
```

3. Now that you have defined the function, test it using some simple values that you can calculate in your head. Try calling the function with a `gradient` of `2`, an `x` value of `4`, and a `constant` of `-3`:

```
>>> # test with a single value first
>>> straight_line(2,4,-3)
5
```

4. Let's now try a more complex test of the function, using the following code:

```
>>> for x in range(10):
...     print(x,straight_line(2,x,-3))
...
0 -3
1 -1
2 1
3 3
4 5
5 7
6 9
7 11
8 13
9 15
```

5. Finally, check that the `help()` function correctly recognizes the function:

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

straight_line(gradient, x, constant)
    returns y coordinate of a straight line
    -> gradient * x + constant
(END)
```

How It Works

In the first line in step 2, you created the function definition. You named it `straight _ line` and said it had three required parameters: `gradient`, `x`, and `constant`. These parameters correspond to the values used in the mathematical equation $y = mx+c$, where m is the gradient and c is a constant.

The second line is a documentation string that describes what the function is for and how it should be used.

The third line is the function code block. It could be arbitrarily complex and several lines long, but in this case it's a one-liner and you are returning the result so you prefixed it with the keyword `return`. Note that the code line has to start at the same indentation level as the start of the documentation string; otherwise, you will get an indentation error.

You then tested the function with some simple values. Some mental arithmetic confirms that the return value of 5 does indeed equal (2*4-3). The function seems to work, at least for a simple case.

You then used the function to generate a set of *x*-,*y*-coordinate pairs using a `for` loop with a fixed value for the `gradient` (`2`) and `constant` (`-3`) but supplying the loop variable as `x`. If you have some paper handy, you can try plotting the coordinates listed to confirm that they form a straight line.

Finally, you used the `help()` function to confirm that the documentation string was correctly detected and displayed.

Generator Functions

The next form of function you look at is the generator function. Generator functions look exactly like standard functions except that instead of using `return` to pass back a value, they use the keyword `yield`. (In theory they can use return in addition to `yield`, but only the `yield` expressions produce the generator behavior.)

The bit of Pythonic magic that makes generator functions special is that they act like a freeze-frame camera. When a standard function reaches a `return` statement, it returns the value and the function then throws away all of its internal data. The next time the function is called, everything starts off from scratch.

The `yield` statement does something different. It returns a value, just like `return` does, but it doesn't cause the function to throw away its data; instead, everything is preserved. The next call of the function picks up from the `yield` statement, even if that's in the middle of a block or part of a loop. There can even be multiple `yield` statements in a single function. Because the `yield` statement can be inside a loop, it is possible to create a function that effectively returns an infinite series of results. Here is an example that returns an incrementing series of odd numbers:

```
def odds(start=1):
     ''' return all odd numbers from start upwards'''
     if int(start) % 2 == 0: start = int(start) + 1
     while True:
          yield start
          start += 2
```

In this function you first check that the `start` argument passed is an odd integer (an even integer divided by 2 has a zero remainder), and if not, you force it to be the next highest odd integer by adding 1. You then create an infinite `while` loop. Normally this would be a very bad idea because your program would just block forever. However, because this is a generator function, you are using `yield` to return the value of `start` so the function exits at this point, returning the value of `start` at this moment in time. The next time the function is called, execution starts right where you left off. So `start` is incremented by 2, and the loop goes round again, yielding the next odd number and exiting the function until the next time.

Python ensures that generator functions become iterators so that you can use them in `for` loops, like so:

```
for n in odds():
      if n > 7: break
      else: print(n)
```

You use `odds()` just like a collection. Every time the loop goes around, it calls the generator function and receives the next odd value.

You avoid an infinite loop by inserting the `break` test so you never call `odds()` beyond 7.

> ***NOTE*** *If you use* `odds()` *a second time in the same program, it creates a brand-new instance of the iterator and the sequence starts over.*

Now that you understand how generator functions work, you may have realized that the generator expressions introduced earlier in this chapter are effectively just anonymous generator functions. Generator expressions are effectively a disguised form of a generator function without a name.

This provides a perfect segue to the final function type we will be learning about here—the lambda function.

Lambda Functions

The term *lambda* comes from a form of mathematical calculus invented by Alonzo Church. The good news is that you don't need to know anything about the math to use lambda functions! The idea behind a lambda function is that it is an anonymous function block, usually very small, that you can insert into code that then calls the lambda function just like a normal function. Lambda functions are not things you will use often, but they are very handy when you would otherwise have to create a lot of small functions that are used only once. They are often used in GUI or network programming contexts, where the programming toolkit requires a function to call back with a result.

A lambda function is defined like this:

```
lambda <param1, param2, ...,paramN> : <expression>
```

That's the literal word `lambda`, followed by an optional, comma-separated list of parameter names, a colon, and an arbitrary Python expression that, usually, uses the input parameters. Note that the expression does not have the word `return` in front of it.

Some languages allow lambda functions to be arbitrarily complex; however, Python limits you to a single expression. The expression can become quite complex, but in practice it's better to create a standard function if that is the case because it will be easier to read and debug if things go wrong.

You can assign lambda functions to variables, in which case those variables look exactly like standard Python function names. For example, here is the `straight _ line` example function re-implemented as a lambda function:

```
>>> straight_line = lambda m,x,c: m*x+c
>>> straight_line(2,4,-3)
5
```

You see lambda functions popping up later in the book. Just remember that they are simply a concise way to express a short, single expression, function.

Defining and Using Classes and Objects

Python supports object-oriented programming using a traditional class-based approach. Python classes support multiple inheritance and operator overloading (but not method overloading), as well as the usual mechanisms of encapsulation and message passing. Python classes do not directly implement data hiding, although some naming conventions are used to provide a thin layer of privacy for attributes and to suggest when attributes should not be used directly by clients. Class methods and data are supported as well as the concepts of *properties* and *slots*. Classes have both a constructor (`__new__ ()`) and an initializer (`__init__ ()`), as well as a destructor mechanism (`__del__ ()`), although the latter is not always guaranteed to be called. Classes act as namespaces for the methods and data defined therein.

Objects are instances of classes. Instances can have their own attributes added after instantiation, although this is not normal practice.

A class is defined using the `class` keyword followed by the class name and a parenthesized list of super-classes. The class definition contains a set of class data and method definitions. A method definition has as its first parameter a reference to the calling instance, traditionally called `self`. A simple class definition looks like this:

```
class MyClass(object):
    instance_count = 0
    def __init__(self, value):
        self.__value = value
        MyClass.instance_count += 1
        print("instance No {} created".format(MyClass.instance_count))
    def aMethod(self, aValue):
        self.__value *= aValue
    def __str__(self):
        return "A MyClass instance with value: " + str(self.__value)
    def __del__(self):
        MyClass.instance_count -= 1
```

The class name traditionally starts with an uppercase letter. In Python 3 the super class is always `object` unless specifically stated otherwise, so the use of `object` as the super class in the preceding example is actually redundant. The `instance _ count` data item is a class attribute because it

does not appear inside any of the class' methods. The `__init__()` function is the initializer (constructors are rarely used in Python unless inheriting from a built-in class). It sets the instance variable `self.__value`, increments the previously defined class variable `instance_count`, and then prints a message. The double underscores before `value` indicate that it is effectively private data and should not be used directly. The `__init__()` method is called automatically by Python immediately after the object is constructed. The instance method `aMethod()` modifies the instance attribute created in the `__init__()` method. The `__str__()` method is a special method used to return a formatted string so that instances passed to the print function, for example, will be printed in a meaningful way. The destructor `__del__()` decrements the class variable `instance_count` when the object is destroyed.

You can create an instance of the class like this:

```
myInstance = MyClass(42)
```

This creates an instance in memory and then calls `MyClass.__init__()` with the new instance as `self` and `42` as `value`.

You can call the `aMethod()` method using dot notation like this:

```
myInstance.aMethod(66)
```

This is translated to the more explicit invocation,

```
MyClass.aMethod(myInstance, 66)
```

and results in the desired behavior whereby the value of the `__value` attribute is adjusted.

You can see the `__str__()` method in action if you print the instance, like this:

```
print(myInstance)
```

This should print the message:

```
A MyClass instance with value: 2772
```

You could also print the `instance_count` value before and after creating/destroying an instance:

```
print(MyClass.instance_count)
inst = MyClass(44)
print(MyClass.instance_count)
del(inst)
print(MyClass.instance_count)
```

This should show the count being incremented and then later decremented again. (There may be a slight delay before the destructor is called during garbage collection, but it should only be a few moments.)

The `__init__()`, `__del__()`, and `__str__()` methods are not the only special methods. Several of these exist, all signified by the use of double underscores (they are sometimes called *dunder* methods). Operator overloading is supported via a set of these special methods including: `__add__()`, `__sub__()`, `__mul__()`, `__div__()`, and so on. Other methods provide for the implementation of Python protocols such as iteration or context management. You can override

these methods in your own classes. You should never define your own dunder methods; otherwise, future enhancements to Python could break your code.

You can override methods in subclasses, and the new definitions can invoke the inherited version of the method by using the `super()` function, like this:

```
class SubClass(Parent):
    def __init__(self, aValue):
        super().__init__(aValue)
```

The call to `super().__init__()` translates to a call to the `__init__()` method of `Parent`. Using `super()` avoids problems, particularly with multiple inheritance, where a class could be inherited multiple times and you usually don't want it to be initialized more than once.

> ***NOTE*** *The use of* `super()` *in Python 3 has been greatly simplified compared to its Python 2 form. The* `super()` *line in Python 2 would look like* `super(SubClass, self).__init__(aValue)`*, which is much less intuitive to use.*

Slots are a memory-saving device, and you invoke them by using the `__slots__` special attribute and providing a list of the object attribute names. Often `__slots__` are a premature optimization, and you should use them only if you have a specific, known need.

Properties are another feature available for data attributes. They enable you to make an attribute read only (or even write only) by forcing access to be via a set of methods even though the usual method syntax is not used. This is best seen by an example where you create a `Circle` class with a `radius` attribute and `area()` method. You want the `radius` value to always be positive, so you don't want clients changing it directly in case they pass a negative value. You also want `area` to look like a read-only data attribute even though it is implemented as a method. You achieve both objectives by making `radius` and `area` properties.

TRY IT OUT Creating a Property within a Class (testCircle.py)

In this Try It Out, you start by creating a simple `Circle1` class that has only one attribute and two callable methods: `setRadius()` and `area()`. You then create a second class, `Circle2`, which makes `radius` and `area` properties. Finally, you see how the use of properties simplifies the use of the class in client code.

1. Start your favorite programming editor or IDE and create a new file called `testCircle.py` (or load the file from the book download site).
2. Enter the following code:

```
class Circle1:
    def __init__(self, radius):
        self.__radius = radius
    def setRadius(self,newValue):
        if newValue >= 0:
```

```
            self.__radius = newValue
        else: raise ValueError("Value must be positive")
    def area(self):
        return 3.14159 * (self.__radius ** 2)

class Circle2:
    def __init__(self, radius):
        self.__radius = radius

    def __setRadius(self, newValue):
        if newValue >= 0:
            self.__radius = newValue
        else: raise ValueError("Value must be positive")
    radius = property(None, __setRadius)

    @property
    def area(self):
        return 3.14159 * (self.__radius ** 2)
```

3. Save the code.

4. Start the Python interpreter and type the following code to use `Circle1`:

```
>>> import testCircle as tc
>>> c1 = tc.Circle1(42)
>>> c1.area()
5541.76476
>>> print(c1.__radius)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'Circle1' object has no attribute '__radius'
>>> c1.setRadius(66)
>>> c1.area()
13684.766039999999
>>> c1.setRadius(-4)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "D:\PythonCode\Chapter1\testCircle.py", line 7, in setRadius
    else: raise ValueError("Value must be positive")
ValueError: Value must be positive
```

5. Play with `Circle 2` using the following code:

```
>>> c2 = tc.Circle2(42)
>>> c2.area
5541.76476
>>> print(c2.radius)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: unreadable attribute
>>> c2.radius = 12
>>> c2.area
452.38896
>>> c2.radius = -4
Traceback (most recent call last):
```

```
  File "<interactive input>", line 1, in <module>
  File "D:\PythonCode\Chapter1\testCircle.py", line 18, in __setRadius
    else: raise ValueError("Value must be positive")
ValueError: Value must be positive
>>>
```

How It Works

In `testCircle.py` you created two classes. The first, `Circle1`, achieved what you wanted to do by forcing the user to change the radius value via the `setRadius()` method. You did this by prefixing the attribute `self.__radius` with two underscores, which is how Python makes things appear private. You then created the `setRadius()` method that validated the supplied value before applying it and raised an error if a negative value was found. You also provided an `area()` method so that the user could evaluate the area using the usual method calling technique.

The second class, `Circle2`, went about things rather differently. It used Python's property definition feature to create an attribute called `radius` that was write only. It also created the `area` method as a read-only attribute. This made the user code for `Circle2` much more intuitive, as you saw when you exercised the classes in the interpreter. The key lies in the `property()` type function that you called like this:

```
radius = property(None, __setRadius)
```

This code takes as arguments a set of functions for read, write, and delete (as well as a documentation string). The default value of each is `None`. In this case you created the `radius` property with a `None` read function but with the (now private) `__setRadius()` method as a write function. The other values were left at their default of `None`. The result was that `radius` could be accessed by the user as if it were a public data attribute when assigning a value but, under the covers, Python called the `__setRadius()` method. Any attempt to read (or delete) the attribute would be ignored because the action gets routed to `None`.

The `area` property is slightly different and uses a Python property decorator (`@property`), which is just a shortcut for creating a read-only property. This is a very common use of properties.

Looking at the interactive session, you created a `Circle1` instance and printed the area using the `area()` method. You then tried to print the radius directly by accessing `__radius`, but Python pretended that it had no such attribute (because of the double underscore private setting) and raised an `AttributeError`. When you used the `setRadius()` method, all was well, and printing the area a second time showed that the modification worked. Finally, you tried to set a negative radius and, as expected, the method raised a `ValueError` exception with a custom error message: "Value must be positive."

In the session using `Circle2`, you can see how much simpler the code is. You simply evaluate the `area` property name to get the area and you assign a value to the `radius` property name. When you try to assign a negative value, the method again raises a `ValueError`. Printing the radius directly again generates an `AttributeError`, although this time it has a slightly different message.

Properties require a small amount of extra effort on the programmer's part, but can greatly simplify the usage of the class.

Having seen how to extend Pythons capabilities using functions and classes, the next section shows you how to enclose these extensions in modules and packages for reusability.

CREATING AND USING MODULES AND PACKAGES

Modules are fundamental to most programming environments intended for nontrivial programming tasks. They allow programs to be broken up into manageable chunks and provide a mechanism for code reuse across projects. In Python, modules are simply source files, ending in `.py` and located somewhere that Python can find them. In practice, that means the file must be located in the current working directory or a folder listed in the `sys.path` variable. You can add your own folders to this path by specifying them in the `PYTHONPATH` environment variable for your system or dynamically at run time.

Although modules provide a useful way of packaging up small amounts of source code for reuse, they are not entirely satisfactory for larger projects such as GUI frameworks or mathematical function libraries. For these Python provides the concept of a *package*. A package is essentially a folder full of modules. The only requirement is that the folder should contain a file called `__init__.py`, which may be empty. To the user a package looks a lot like a module, and the submodules within the package look like module attributes.

Using and Creating Modules

You access modules using the `import` keyword, which has many variations in Python. The most common forms are shown here:

```
import aModule
import aModule as anAlias
import firstModule, secondModule, thirdModule...
from aModule import anObject
from aModule import anObject as anAlias
from aModule import firstObject,secondObject, thirdObject...
from aModule import *
```

The last form imports all the visible names from `aModule` into the current namespace. (You learn how to control visibility shortly.) This carries a significant risk of creating naming conflicts with built-in names or names you have defined, or will define, locally. It is therefore recommended that you use only the last import form for testing modules at the Python prompt. The small amount of extra typing involved in using the other forms is a small price to pay compared to the confusion that can result from a name clash. The other `from...` forms are much safer because you only import specified names and, if necessary, rename them with an alias. This makes clashes with other local names much less likely.

Once you import a module using any of the first three forms, you can access its contents by prefixing the required name with the module name (or alias) using dot notation. You have already seen examples of that in the previous sections; for example, `sys.path` is an attribute of the `sys` module.

Having said that modules are simply source files, in practice, you should observe some do's and don'ts when creating modules. You should avoid top-level code that will be run when the module is imported, except, possibly, for some initialization of variables that may depend on the local environment. This means that the code you want to reuse should be packaged as a function or a class. It's also common to provide a test function that exercises all the functions and classes in the module. Module names are also traditionally lowercase only.

> ***NOTE*** *There is a Python style guide known as PEP8 that provides guidance on naming conventions and code layout rules. Its use is not mandatory, but you are strongly recommended to follow it, especially if submitting code for inclusion in the standard library. PEP8 can be found here:* `http://legacy.python.org/dev/peps/pep-0008/`.

You can control visibility of the objects within your module in one of two ways. The first is similar to the privacy mechanism used in classes in that you can prefix a name with an underscore. Such names will not be exported when a `from x import *` style statement is used. The other way to control visibility is to list only the names that you want exported in a top-level variable called `__all__`. This ensures that only the names you specifically want to be exported will be and is recommended over the underscore method if visibility is important to you.

> ***NOTE*** *One very important gotcha with modules is that the* `sys.path` *list is searched in order. This usually means that any modules you create will be found before the built-in or standard library modules. It is very important that you do not use a standard module name for your own module files; otherwise, strange things may happen and, even if you realize that it's your module, that's being accessed, other readers are likely to be fooled.*

You put most of this into practice in the next Try It Out, but first, you need to look at packages and how they differ from modules.

Using and Creating Packages

You discovered at the start of this section that a Python package is just a folder with a file called `__init__.py`. All other Python files within that folder are the modules of the package. Python considers packages as just another type of module, which means that Python packages can contain other packages within them to an arbitrary depth—provided each subpackage also has its own `__init__.py` file, it is a valid package.

> ***NOTE*** *Having just said that a package was defined by having an* `__init__.py`*, this is not strictly true. The real defining feature of a package is that it has a* `__path__` *attribute. However, in practice, you don't need to provide that because Python does it for you. So, if you create an* `__init__.py`*, all will be well.*

The `__init__.py` file itself is not particularly special; it is just another Python file, and it will be loaded when you import the package. This means that the file can be empty, in which case importing the package simply gives access to the included modules, or it can have Python code

within it like any other module. In particular it can define an `__all__` list, as described earlier, to control visibility, effectively enabling a package to have private implementation files that are not exported when a client imports the package.

A common requirement when you create a package is to have a set of functions or data shared between all the included modules. You can achieve this by putting the shared code into a module file called, say, `common.py` at the top level of the package and having all of the other modules `import common`. It will be visible to them as part of the package, but if it is not explicitly included in the `__all__` list, external consumers of the packages will not see it.

When it comes to using a package, you treat it much like any other module. You have all the usual styles of import available, but the naming scheme is extended by using dot notation to specify which submodules you need from the package hierarchy. When you import a submodule using the dot notation, you effectively define two new names: the package and the module. Consider, for example, this statement:

```
import os.path
```

This code imports the `path` submodule of the `os` package. But it also makes the whole of the `os` module visible as well. You can proceed to access `os` functions without having a separate import statement for `os` itself. One implication of this is that Python automatically executes all of the `__init__.py` files that it finds in the imported hierarchy. So in the case of `os.path`, it executes `os.__init__.py` and then `path.__init__.py`.

On the other hand, if you use an alias after the import, like this,

```
import os.path as pth
```

only the `os.path` module is exposed. If you want to use the `os` module functions, you will need an explicit import. Although only `path` is exposed, as the name `pth`, both `os` and `path` `__init__.py` files will still be run.

The Python standard library contains several packages including the `os` package just mentioned. Others of note include the UI frameworks `tkinter` and `curses`, the `email` package, and the web-focused `urllib`, `http`, and `html` packages. You use several of these later in the book.

NAMESPACE PACKAGES

Python 3.3 introduced a new type of package called a *namespace package*. A namespace package contains a number of *portions*. A portion is a reference to an object that may, or may not, have a physical representation and may be located on the network or in a different part of the local file system. Namespace packages do not use the `__init__.py` file technique; rather, they depend on being part of the `sys.path` definition used to find modules during imports.

Namespace packages are so new that, at the time of writing, it is not clear how extensively they will be used. In the short term, you will probably not meet many of them in practice, and the intention is that, to a user, they should not appear significantly different from the traditional-style packages.

You have now covered all of the theory about modules and packages. In the next section, you put this information to work by creating some modules and a package.

CREATING AN EXAMPLE PACKAGE

You've read the theory; now it's time to put it into practice. In this section you create a couple of modules and bundle them as a package. You utilize the bitwise logical operators mentioned in "The Boolean Type" section. The intention is to provide a functional interface to those operators and extend their scope to include testing of individual bit values. In doing this you also see several of the Python core language features that were discussed previously. The modules you develop are not optimized for performance, but are designed to illustrate the concepts. However, it would not be difficult to refine them into genuinely useful tools.

TRY IT OUT Creating a Module (bits.py)

In this Try It Out, you start out by creating a simple, conventional module based on integer inputs. You then create another module that defines a class that can be used to represent a piece of binary data and expose the bitwise functions as methods. Finally, you create a package containing both modules.

1. Create a new folder called `bitwise`. This eventually becomes your package.
2. In that folder create a Python script called `bits.py` containing the following code (or load it from the book's downloadable filenamed `bits.py`):

```
#! /bin/env python3
''' Functional wrapper around the bitwise operators.
Designed to make their use more intuitive to users not
familiar with the underlying C operators.
Extends the functionality with bitmask read/set operations.

The inputs are integer values and
return types are 16 bit integers or boolean.
bit indexes are zero based

Functions implemented are:
    NOT(int)            -> int
    AND(int, int)       -> int
    OR(int,int)         -> int
    XOR(int, int)       -> int
    shiftleft(int, num)  -> int
    shiftright(int, num) -> int
    bit(int,index)      -> bool
    setbit(int, index) -> int
    zerobit(int,index) -> int
    listbits(int,num) -> [int,int...,int]
'''

def NOT(value):
    return ~value
```

```
def AND(val1,val2):
    return val1 & val2

def OR(val1, val2):
    return val1 | val2

def XOR(val1,val2):
    return val1^val2

def shiftleft(val, num):
    return val << num

def shiftright(val, num):
    return val >> num

def bit(val,idx):
    mask = 1 << idx    # all 0 except idx
    return bool(val & mask)

def setbit(val,idx):
    mask = 1 << idx    # all 0 except idx
    return val | mask

def zerobit(val, idx):
    mask = ~(1 << idx)   # all 1 except idx
    return val & mask

def listbits(val):
    num = len(bin(val)) - 2
    result = []
    for n in range(num):
        result.append( 1 if bit(val,n) else 0 )
    return list( reversed(result) )
```

3. Save the file and, while still in your `bitwise` folder, start the Python interpreter.
4. Type the following code to test your new module:

```
>>> import bits
>>> bits.NOT(0b0101)
-6
>>> bin(bits.NOT(0b0101))
'-0b110'
>>> bin(bits.NOT(0b0101) & 0xF)
'0b1010'
>>> bin(bits.AND(0b0101, 0b0011) & 0xF)
'0b1'
>>> bin(bits.AND(0b0101, 0b0100) & 0xF)
'0b100'
>>> bin(bits.OR(0b0101, 0b0100) & 0xF)
'0b101'
>>> bin(bits.OR(0b0101, 0b0011) & 0xF)
'0b111'
>>> bin(bits.XOR(0b0101, 0b11) & 0xF)
```

```
'0b110'
>>> bin(bits.XOR(0b0101, 0b0101) & 0xF)
'0b0'
>>> bin(bits.shiftleft(0b10,1))
'0b100'
>>> bin(bits.shiftleft(0b10,4))
'0b100000'
>>> bin(bits.shiftright(0b1000,2))
'0b10'
>>> bin(bits.shiftright(0b1000,6))
'0b0'
>>> bits.bit(0b0101,0)
True
>>> bits.bit(0b0101,1)
False
>>> bin(bits.setbit(0b1000,1))
'0b1010'
>>> bin(bits.zerobit(0b1000,1))
'0b1000'
>>> bits.listbits(0b10111)
[1, 0, 1, 1, 1]
```

How It Works

The module is a fairly straightforward list of functions that wrap the built-in bitwise operators for not (`~`), and (`&`), or (`|`), xor (`^`), shift left (`<<`), and shift right (`>`). These operations work on binary data—that is, simply a sequence of 1s and 0s stored as a unit within the computer. All data in the computer is, ultimately, stored in binary form.

These wrapper operations are complemented by a set of functions that test whether a bit has a value of 1 (this is known as being "set"), set a bit (to 1), or zero a bit (also known as "resetting" the bit). The bit number counts from the right, starting at zero. The tests are done using a bit pattern (also known as a *bitmask*) that, in all cases except `zerobit()`, consists of all zeros except for the bit you want to test or set. You created the mask by shifting 1 left by the required number of bits. `zerobit()` uses the bitwise complement of the usual mask to create one that consists of all 1s apart from a 0 where you want to reset the bit.

Finally, you have a function that lists the individual bits of the given value. This last function is slightly more complex and demonstrates some of Python's coding features. You first determine the length of the number by converting to a binary string with `bin()` and subtracting 2 (to account for the leading `0b` characters). You then create an empty result list and loop over the bits. For each bit you append either a 1 or 0, depending on whether or not the bit is set, using Python's conditional expression construct.

The testing of the module throws up some interesting issues. You start off by importing your new module. Because you are in the folder where the file lives, Python can see it without modifying the `sys.path` value. You start testing with the `NOT()` function (prefixed, of course, with the module name, `bits`), and straightaway you can see an anomaly in that the Python interpreter prints the decimal representation as the result. To get around that, you can use the `bin()` function to convert the number to a binary string representation. However, there is still a problem because the number is negative. This is because Python integers are signed, that is, they can represent positive or negative numbers. Python does this internally by having the leftmost bit represent the sign. By inverting all of the bits, you also

invert the sign! You can avoid the confusion by using a bitmask of 0xF (or decimal 15 if you prefer) to retrieve only the rightmost 4 bits. By converting this with `bin()`, you now see the inverted bit pattern you expected. Obviously, if the value you were inverting was bigger than 16, you would need to use a longer bitmask. Just remember that each hex digit is 4 bits, so all you need to do is add an extra F to your mask.

The next set of tests—covering the functions `AND()` through to `shiftleft()`—should be straightforward, and you can check the results by visually inspecting the input bit patterns and the results. The `shiftright()` examples do show one interesting outcome in that shifting the bits too far to the right produces a zero result. In other words, Python fills the "empty" space left by the shift operations with zeros.

Moving on to the new functionality, you used `bit()`, `setbit()`, and `zerobit()` to test and modify individual bits within the given value. Again, you can visually inspect the input and result patterns to see that the correct results are produced. Remember that the index parameter counts from zero starting from the right.

Finally, you tested the `listbits()` function. Once more, you can easily compare the binary input pattern with the resultant list of numbers.

So you see that you now have a working module that you can import and use just like any other module in Python. You could enhance the module further by providing a test function and wrapping that in an `if __name__` clause if you wanted, but for now you can proceed to look at how to move from a single module to a package.

TRY IT OUT Creating a Package (bitmask.py)

In this Try It Out, you build a class that replicates the functions in `bits.py` as a set of methods. You then bundle both modules into a package.

1. Navigate into your `bitwise` folder.
2. Create a new file called `bitmask.py` with the following code (or load it from the book's downloadable filename `bitmask.py`):

```
#! /bin/env python3
''' Class that represents a bit mask.
It has methods representing all
the bitwise operations plus some
additional features. The methods
return a new BitMask object or
a boolean result. See the bits
module for more on the operations
provided.
'''

class BitMask(int):
   def AND(self,bm):
      return BitMask(self & bm)
   def OR(self,bm):
```

```
        return BitMask(self | bm)
    def XOR(self,bm):
        return BitMask(self ^ bm)
    def NOT(self):
        return BitMask(~self)
    def shiftleft(self, num):
        return BitMask(self << num)
    def shiftright(self, num):
        return BitMask(self >> num)
    def bit(self, num):
        mask = 1 << num
        return bool(self & mask)
    def setbit(self, num):
        mask = 1 << num
        return BitMask(self | mask)
    def zerobit(self, num):
        mask = ~(1 << num)
        return BitMask(self & mask)
    def listbits(self, start=0,end=None):
        if end: end = end if end < 0 else end+2
        return [int(c) for c in bin(self)[start+2:end]]
```

3. Now save it so that you can test it in the Python interpreter.
4. Staying in the `bitwise` folder, start Python and type the following code:

```
>>> import bitmask
>>> bm1 = bitmask.BitMask()
>>> bm1
0
>>> bin(bm1.NOT() & 0xf)
'0b1111'
>>> bm2 = bitmask.BitMask(0b10101100)
>>> bin(bm2 & 0xFF)
'0b10101100'
>>> bin(bm2 & 0xF)
'0b1100'
>>> bm1.AND(bm2)
0
>>> bin(bm1.OR(bm2))
'0b10101100'
>>> bm1 = bm1.OR(0b110)
>>> bin(bm1)
'0b110'
>>> bin(bm2)
'0b10101100'
>>> bin(bm1.XOR(bm2))
'0b10101010'
>>> bm3 = bm1.shiftleft(3)
>>> bin(bm3)
'0b110000'
>>> bm1 == bm3.shiftright(3)
True
>>> bm4 = bitmask.BitMask(0b11110000)
>>> bm4.listbits()
```

```
[1, 1, 1, 1, 0, 0, 0, 0]
>>> bm4.listbits(2,5)
[1, 1, 0]
>>> bm4.listbits(2,-2)
[1, 1, 0, 0]
```

5. Quit the interpreter.

   Now that you have proved the new module works, you can go ahead and convert the `bitwise` directory into a Python package.

6. Create a new empty `__init__.py` file.

7. To test that the package works, you need to change your working directory to the directory above `bitwise`. Do that now.

   You now need to test that you can import the package and its contents and access the functionality.

8. Start the Python interpreter and type the following test code:

```
>>> import bitwise.bits as bits
>>> from bitwise import bitmask
>>> bits
<module 'bitwise.bits' from 'bitwise/bits.py'>
>>> bitmask
<module 'bitwise.bitmask' from 'bitwise/bitmask.py'>
>>> bin(bits.AND(0b1010,0b1100))
'0b1000'
>>> bin(bits.OR(0b1010,0b1100))
'0b1110'
>>> bin(bits.NOT(0b1010))
'-0b1011'
>>> bin(bits.NOT(0b1010) & 0xFF)
'0b11110101'
>>> bin(bits.NOT(0b1010) & 0xF)
'0b101'
>>> bm = bitmask.BitMask(0b1100)
>>> bin(bm)
'0b1100'
>>> bin(bm.AND(0b1110))
'0b1100'
>>> bin(bm.OR(0b1110))
'0b1110'
>>> bm.listbits()
[1, 1, 0]
```

How It Works

You created a class based on the built-in integer type, `int`. Because you are only providing new methods for the class and not storing any additional data attributes, you don't need to provide a `__new__()` constructor or `__init__()` initializer. The methods are all very similar to the functions written in `bits.py` except that you created a `BitMask` instance as the return type. The `listbits()` method also shows an alternative approach to deriving the list using the `bin()` string representation, and creating the list using a list comprehension based on a character-to-integer conversion using `int()`. `listbits()` has also been extended to provide a pair of `start` and `end` parameters that default to the full length of

the binary number, but could be used to extract a subset of bits. There is a small piece of work involved in adjusting the `end` value depending on whether it is a positive or negative index. Negative indices do not need the addition of two characters because they automatically apply from the right-hand end, so a Python conditional assignment ensures the correct end value is set.

Having created the class, you then tested it as a standard module by importing it from the local directory. You then repeated a similar set of tests to the ones you did for `bits.py`. A few points to note include the fact that you can mix and match the traditional bitwise operators with the new functional versions. You can also compare `BitMask` objects just like any other integer, as you saw in the `shiftright()` example. Finally, you proved that your new `listbits()` algorithm worked and the new additional arguments function as expected for both positive and negative values.

At this stage you had created two standard modules in a folder. You then created a blank `__init__.py` file that turned the folder into a Python package. To test that it worked, you moved up a directory level so that the package was visible to the interpreter. You then confirmed that you could import the package and modules within it and access some of the functionality. Congratulations, you now have a package with two contained modules.

Knowing how to create—and use—the standard modules and packages, as well as ones you create yourself, is a great starting point. However, there are many more modules and packages available on the internet, just waiting to be downloaded. The next section explains how you can do that.

USING THIRD-PARTY PACKAGES

Many third-party packages are available for Python. Binary distributions of many of these packages, complete with installer programs, are available for most common operating systems. If a binary installer is available, either on the package website or, for Linux users, in your package management tool, you should use it because it will be the simplest way of getting things up and running. If a binary package is not available, you need to download and install the base package.

You can find many of these third-party packages in the Python Package Index (PyPI) at `https://pypi.python.org/pypi`. They are distributed in a special format that itself requires the installation of a third-party package! This chicken-and-egg situation often confuses beginners, so this section describes how to set up your environment such that you can access these third-party packages.

PyPI packages come in the form of something called an *egg*. A Python egg is capable of delivering either a standard Python package or a binary package written in C, or a mix of Python and C code.

THE FUTURE OF PYTHON PACKAGING

The egg format has some issues and is itself being replaced by something called a *wheel*. This is all part of a wider strategy to rationalize the multiple methods of distributing Python packages and applications. The Python Package Authority is leading this project. Eventually, all the tools needed to both build and install Python

packages should be available in a standard Python installation. The roadmap starts to take effect in Python 3.4 with the inclusion of `pip` in the standard distribution.

You should refer to the latest guidance on the Python Package Authority website (`https://python-packaging-user-guide.readthedocs.org/en/latest/`) if you want to create your own distributable packages.

Installing an egg requires a tool called *pip*. Fortunately, installing pip does not require pip. As of version 3.4 of Python, pip is included in the standard library, which simplifies the process somewhat. If pip is not included in your version of Python, you can install pip by going to `https://pip.pypa.io/en/latest/reference/pip_install.html` and following the instructions.

Download the `get-pip.py` file using the link on the page into any convenient folder on your computer. Change into that folder, make sure you are connected to the Internet, and run the following:

```
python get-pip.py
```

This will take a few moments and downloads some stuff from the Internet. You will see a few messages like this:

```
$ python3 get-pip.py
Downloading/unpacking pip
  Downloading pip-1.5.2-py2.py3-none-any.whl (1.2MB): 1.2MB downloaded
Installing collected packages: pip
Successfully installed pip
Cleaning up...
```

Once pip is installed, you can use it to install a PyPI package using:

```
pip install SomePackageName
```

This installs the latest version of the specified package. You can uninstall a package just as easily using:

```
pip uninstall SomePackageName
```

Many other options are available to use, and they are described on the pip documentation page at: `https://pip.pypa.io/en/latest/reference/index.html`.

Not all packages use pip, and other install options and tools exist. The package documentation should explain what you need to do. The current state of confusion should resolve itself in the near future as explained in the earlier sidebar box: “The Future of Python Packaging.”

SUMMARY

In this chapter you reviewed the core language features of Python. You looked at the interpreter environment, the core data types, and the language control structures and syntax. You also considered how Python can be extended by writing functions, classes, modules, and packages.

The core data types are Boolean (`bool`), integer (`int`), and floating point (`float`) numbers, as well as the special `None` type. Python also supports several collection types including strings, bytes, lists, tuples, dictionaries, and sets.

The control structures cover all of the structured programming concepts including sequences, selection, and iteration. Selection is done using `if/elif/else` and a conditional expression form. Iteration is supported via two loop constructs: `for`, which iterates over a collection or iterable object, and `while`, which is a more general, and potentially infinite, loop. Python also supports exception management via a `try/except/finally` construct.

In addition to a large number of built-in functions and a standard library of modules, Python enables you to extend its capabilities by writing your own functions using the `def` or `lambda` keywords. You can also extend the standard data types by creating your own data types using the `class` keyword and then creating instances of those classes. Functions and classes can be stored in separate files, which constitute Python modules that can be imported into other code, thus facilitating cross program reuse. Modules can in turn be grouped into packages, which are simply folders containing an `__init__.py` file.

EXERCISES

1. How do you convert between the different Python data types? What data quality issues arise when converting data types?
2. Which of the Python collection types can be used as the key in a dictionary? Which of the Python data types can be used as a value in a dictionary?
3. Write an example program using an `if/elif` chain involving at least four different selection expressions.
4. Write a Python `for` loop to repeat a message seven times.
5. How can an infinite loop be created using a Python `while` loop? What potential problem might this cause? How could that issue be resolved?
6. Write a function that calculates the area of a triangle given the base and height measurements.
7. Write a class that implements a rotating counter from 0 to 9. That is, the counter starts at 0, increments to 9, resets to 0 again, and repeats that cycle indefinitely. It should have `increment()` and `reset()` methods, the latter of which returns the current count then sets the count back to 0.

Answers to exercises can be found in Appendix A.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Python infrastructure	The Python interpreter can be called without arguments to get an interactive shell. If a filename is given as an argument, the interpreter will execute it and exit.
Simple data types	Python supports integers, floating-point Boolean, and `None` data types. The type names can be used as conversion functions. Python types are objects and support a rich set of operations.
Collection data types	Python supports Unicode and byte strings plus lists, tuples, dictionaries, and sets. Strings and tuples are immutable (cannot be changed), and dictionaries and sets require immutable types as keys. Most collections are iterables and can be used in `for` loops.
Basic control structures	Python supports sequences, selection, and repetition. Sequences are simple lines of code; there are no block markers or statement terminators required. Selection is via the `if/elif/else` structure. Two loops are provided: `for` and `while`. Code blocks are indicated by a terminating colon on the previous line, and the block will be indented under that line. Restoring the indentation level ends the block.
Error handling	Python supports exception handling through the `try/except/.else/finally` structure. Users can define their own exceptions or parameterize the built-in errors.
Input/output	User input can be read, as a string, using the `input()` function. User output can be displayed via the `print()` function. Text files can be opened, read from, and written to. File navigation is possible using `tell()` and `seek()`.
Defining functions	New functions can be defined using the `def` or `lambda` keywords. Functions can receive input via parameters and provide results via the `return` keyword.
Defining classes	Classes are defined using the `class` keyword. Classes support single and multiple inheritance, polymorphism, operator overloading, and method overriding. Class attributes can be treated as properties and/or slots. Attributes are accessed via dot notation. Classes are objects, too.
Modules and packages	Modules are just files containing Python code that exist in any of the folders listed in `sys.path`. Packages are folders containing modules and a (possibly empty) file called `__init__.py`. Packages are modules, too. Names within a module are accessed via dot notation.