

6

Strings

This chapter covers

- Understanding strings as sequences of characters
- Using basic string operations
- Inserting special characters and escape sequences
- Converting from objects to strings
- Formatting strings
- Using the byte type

Handling text—from user input, to filenames, to processing chunks of text—is a common chore in programming. Python comes with powerful tools to handle and format text. This chapter discusses the standard string and string-related operations in Python.

6.1 *Strings as sequences of characters*

For the purposes of extracting characters and substrings, strings can be considered sequences of characters, which means you can use index or slice notation:

```
>>> x = "Hello"
>>> x[0]
'H'
```

```
>>> x[-1]
'o'
>>> x[1:]
'ello'
```

One use for slice notation with strings is to chop the newline off the end of a string, usually a line that's just been read from a file:

```
>>> x = "Goodbye\n"
>>> x = x[:-1]
>>> x
'Goodbye'
```

This is just an example—you should know that Python strings have other, better methods to strip unwanted characters, but this illustrates the usefulness of slicing.

You can also determine how many characters are in the string by using the `len` function, just like finding out the number of elements in a list:

```
>>> len("Goodbye")
7
```

But strings aren't lists of characters. The most noticeable difference between strings and lists is that, unlike lists, *strings can't be modified*. Attempting to say something like `string.append('c')` or `string[0] = 'H'` will result in an error. You'll notice in the previous example that we stripped off the newline from the string by creating a string that was a slice of the previous one, not by modifying the previous string directly. This is a basic Python restriction, imposed for efficiency reasons.

6.2 *Basic string operations*

The simplest (and probably most common) way of combining Python strings is to use the string concatenation operator `+`:

```
>>> x = "Hello " + "World"
>>> x
'Hello World'
```

There is an analogous string multiplication operator that I have found sometimes, but not often, useful:

```
>>> 8 * "x"
'xxxxxxxx'
```

6.3 *Special characters and escape sequences*

You've already seen a few of the character sequences Python regards as special when used within strings: `\n` represents the newline character and `\t` represents the tab character. Sequences of characters that start with a backslash and that are used to represent other characters are called *escape sequences*. Escape sequences are generally used to represent *special characters*—that is, characters (such as tab and newline) that don't have a standard one-character printable representation. This section covers escape sequences, special characters, and related topics in more detail.

6.3.1 Basic escape sequences

Python provides a brief list of two-character escape sequences to use in strings (table 6.1).

Table 6.1 Escape sequences

Escape sequence	Character represented
\'	Single-quote character
\"	Double-quote character
\\	Backslash character
\a	Bell character
\b	Backspace character
\f	Formfeed character
\n	Newline character
\r	Carriage return character (not the same as \n)
\t	Tab character
\v	Vertical tab character

The ASCII character set, which is the character set used by Python and the standard character set on almost all computers, defines quite a few more special characters. They're accessed by the numeric escape sequences, described in the next section.

6.3.2 Numeric (octal and hexadecimal) and Unicode escape sequences

You can include any ASCII character in a string by using an octal (base 8) or hexadecimal (base 16) escape sequence corresponding to that character. An octal escape sequence is a backslash followed by three digits defining an octal number; the ASCII character corresponding to this octal number is substituted for the octal escape sequence. A hexadecimal escape sequence is similar but starts with `\x` rather than just `\` and can consist of any number of hexadecimal digits. The escape sequence is terminated when a character is found that's not a hexadecimal digit. For example, in the ASCII character table, the character *m* happens to have decimal value 109. This is octal value 155 and hexadecimal value 6D, so:

```
>>> 'm'
'm'
>>> '\155'
'm'
>>> '\x6D'
'm'
```

All three expressions evaluate to a string containing the single character *m*. But these forms can also be used to represent characters that have no printable representation. The newline character `\n`, for example, has octal value 012 and hexadecimal value 0A:

```
>>> '\n'
'\n'
```

```
>>> '\012'
'\n'
>>> '\x0A'
'\n'
```

Because all strings in Python 3 are Unicode strings, they can also contain almost every character from every language available. Although a discussion of the Unicode system is far beyond this book, the following examples illustrate that you can also escape any Unicode character, either by number similar to that shown earlier or by Unicode name:

```
>>> unicode_a = '\N{LATIN SMALL LETTER A}'
>>> unicode_a
'a'
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> "\u00E1"
'á'
>>>
```

Escapes by Unicode name

Escapes by number, using \u

The Unicode character set includes the common ASCII characters ❶.

6.3.3 *Printing vs. evaluating strings with special characters*

We talked before about the difference between evaluating a Python expression interactively and printing the result of the same expression using the `print` function. Although the same string is involved, the two operations can produce screen outputs that look different. A string that is evaluated at the top level of an interactive Python session will be shown with all of its special characters as octal escape sequences, which makes clear what is in the string. Meanwhile, the `print` function passes the string directly to the terminal program, which may interpret special characters in special ways. For example, here's what happens with a string consisting of an `a` followed by a newline, a tab, and a `b`:

```
>>> 'a\n\tb'
'a\n\tb'
>>> print('a\n\tb')
a
    b
```

In the first case, the newline and tab are shown explicitly in the string; in the second, they're used as newline and tab characters.

A normal `print` function also adds a newline to the end of the string. Sometimes (that is, when you have lines from files that already end with newlines) you may not want this behavior. Giving the `print` function an `end` parameter of `" "` causes the `print` function to not append the newline:

```
>>> print("abc\n")
abc

>>> print("abc\n", end=" ")
abc
>>>
```

6.4 String methods

Most of the Python string methods are built into the standard Python string class, so all string objects have them automatically. The standard `string` module also contains some useful constants. Modules will be discussed in detail in chapter 10.

For the purposes of this section, you need only remember that most string methods are attached to the string object they operate on by a dot (`.`), as in `x.upper()`. That is, they're prepended with the string object followed by a dot.

Because strings are immutable, the string methods are used only to obtain their return value and don't modify the string object they're attached to in any way.

We'll begin with those string operations that are the most useful and commonly used and then go on to discuss some less commonly used but still useful operations. At the end, we'll discuss a few miscellaneous points related to strings. Not all of the string methods are documented here. See the documentation for a complete list of string methods.

6.4.1 The split and join string methods

Anyone who works with strings is almost certain to find the `split` and `join` methods invaluable. They're the inverse of one another—`split` returns a list of substrings in the string, and `join` takes a list of strings and puts them together to form a single string with the original string between each element. Typically, `split` uses whitespace as the delimiter to the strings it's splitting, but you can change that via an optional argument.

String concatenation using `+` is useful but not efficient for joining large numbers of strings into a single string, because each time `+` is applied, a new string object is created. Our previous "Hello World" example produced two string objects, one of which was immediately discarded. A better option is to use the `join` function:

```
>>> " ".join(["join", "puts", "spaces", "between", "elements"])
'join puts spaces between elements'
```

By changing the string used to `join`, you can put anything you want between the joined strings:

```
>>> "::".join(["Separated", "with", "colons"])
'Separated::with::colons'
```

You can even use an empty string, `" "`, to join elements in a list:

```
>>> "".join(["Separated", "by", "nothing"])
'Separatedbynothing'
```

The most common use of `split` is probably as a simple parsing mechanism for string-delimited records stored in text files. By default, `split` splits on any whitespace, not just a single space character, but you can also tell it to split on a particular sequence by passing it an optional argument:

```
>>> x = "You\t\t can have tabs\t\n \t and newlines \n\n " \
        "mixed in"
>>> x.split()
['You', 'can', 'have', 'tabs', 'and', 'newlines', 'mixed', 'in']
>>> x = "Mississippi"
```

```
>>> x.split("ss")
['Mi', 'i', 'ippi']
```

Sometimes it's useful to permit the last field in a joined string to contain arbitrary text, including, perhaps, substrings that may match what `split` splits on when reading in that data. You can do this by specifying how many splits `split` should perform when it's generating its result, via an optional second argument. If you specify n splits, then `split` will go along the input string until it has performed n splits (generating a list with $n+1$ substrings as elements) or until it runs out of string. Here are some examples:

```
>>> x = 'a b c d'
>>> x.split(' ', 1)
['a', 'b c d']
>>> x.split(' ', 2)
['a', 'b', 'c d']
>>> x.split(' ', 9)
['a', 'b', 'c', 'd']
```

When using `split` with its optional second argument, you must supply a first argument. To get it to split on runs of whitespace while using the second argument, use `None` as the first argument.

I use `split` and `join` extensively, usually when working with text files generated by other programs. But you should know that if you're able to define your own data file format for use solely by your Python programs, there's a much better alternative to storing data in text files. We'll discuss it in chapter 13 when we talk about the `Pickle` module.

6.4.2 Converting strings to numbers

You can use the functions `int` and `float` to convert strings into integer or floating-point numbers, respectively. If they're passed a string that can't be interpreted as a number of the given type, they will raise a `ValueError` exception. Exceptions are explained in chapter 14, "Reading and writing files." In addition, you may pass `int` an optional second argument, specifying the numeric base to use when interpreting the input string:

```
>>> float('123.456')
123.456
>>> float('xxyy')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for float(): xxyy
>>> int('3333')
3333
>>> int('123.456')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 10: 123.456
>>> int('10000', 8)
4096
>>> int('101', 2)
5
>>> int('ff', 16)
```

Can't have decimal
point in integer

Interprets 10000
as octal number

```

255
>>> int('123456', 6)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 6: '123456'

```

Can't interpret 123456
as base 6 number

Did you catch the reason for that last error? We requested that the string be interpreted as a base 6 number, but the digit 6 can never appear in a base 6 number. Sneaky!

6.4.3 Getting rid of extra whitespace

A trio of simple methods that are surprisingly useful are the `strip`, `lstrip`, and `rstrip` functions. `strip` returns a new string that's the same as the original string, except that any whitespace at the *beginning or end* of the string has been removed. `lstrip` and `rstrip` work similarly, except that they remove whitespace only at the left or right end of the original string, respectively:

```

>>> x = " Hello,      World\t\t "
>>> x.strip()
'Hello,      World'
>>> x.lstrip()
'Hello,      World\t\t '
>>> x.rstrip()
' Hello,      World'

```

In this example, tab characters are considered to be whitespace. The exact meaning may differ across operating systems, but you can always find out what Python considers to be whitespace by accessing the `string.whitespace` constant. On my Windows system, it gives the following:

```

import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> " \t\n\r\v\f"
' \t\n\r\x0b\x0c'

```

The characters given in backslashed hex (`\xnn`) format represent the vertical tab and formfeed characters. The space character is in there as itself. It may be tempting to change the value of this variable, to attempt to affect how `strip` and so forth work, but don't do it. Such an action isn't guaranteed to give you the results you're looking for.

But you can change which characters `strip`, `rstrip`, and `lstrip` remove by passing a string containing the characters to be removed as an extra parameter:

```

>>> x = "www.python.org"
>>> x.strip("w")
'.python.org'
>>> x.strip("gor")
'www.python.'
>>> x.strip(".gorw")
'python'

```

Strips off
all ws

1 Strips off all
gs, os, and rs

Strips of all dots,
gs, os, rs and ws

Note that `strip` removes any and all of the characters in the extra parameter string, no matter in which order they occur 1.

The most common use for these functions is as a quick way of cleaning up strings that have just been read in. This is particularly helpful when you're reading lines from files (discussed in chapter 13), because Python always reads in an entire line, including the trailing newline, if it exists. When you get around to processing the line read in, you typically don't want the trailing newline. `rstrip` is a convenient way to get rid of it.

6.4.4 String searching

The string objects provide a number of methods to perform simple string searches. Before I describe them, though, let's talk about another module in Python: `re`. (This module will be discussed in depth in chapter 17, "Regular expressions.")

Another method for searching strings: the `re` module

The `re` module also does string searching but in a far more flexible manner, using *regular expressions*. Rather than searching for a single specified substring, an `re` search can look for a string pattern. You could look for substrings that consist entirely of digits, for example.

Why am I mentioning this, when `re` is discussed fully later? In my experience, many uses of basic string searches are inappropriate. You'd benefit from a more powerful searching mechanism but aren't aware that one exists, and so you don't even look for something better. Perhaps you have an urgent project involving strings and don't have time to read this entire book. If basic string searching will do the job for you, that's great. But be aware that you have a more powerful alternative.

The four basic string-searching methods are all similar: `find`, `rfind`, `index`, and `rindex`. A related method, `count`, counts how many times a substring can be found in another string. We'll describe `find` in detail and then examine how the other methods differ from it.

`find` takes one required argument: the substring being searched for. `find` returns the position of the first character of the first instance of `substring` in the `string` object, or `-1` if `substring` doesn't occur in the string:

```
>>> x = "Mississippi"
>>> x.find("ss")
2
>>> x.find("zz")
-1
```

`find` can also take one or two additional, optional arguments. The first of these, if present, is an integer `start`; it causes `find` to ignore all characters before position `start` in `string` when searching for `substring`. The second optional argument, if present, is an integer `end`; it causes `find` to ignore characters at or after position `end` in `string`:

```
>>> x = "Mississippi"
>>> x.find("ss", 3)
5
```



```
>>> x.find("ss", 0, 3)
-1
```

`rfind` is almost the same as `find`, except that it starts its search at the end of `string` and so returns the position of the first character of the last occurrence of `substring` in `string`:

```
>>> x = "Mississippi"
>>> x.rfind("ss")
5
```

`rfind` can also take one or two optional arguments, with the same meanings as those for `find`.

`index` and `rindex` are identical to `find` and `rfind`, respectively, except for one difference: if `index` or `rindex` fails to find an occurrence of `substring` in `string`, it doesn't return `-1` but rather raises a `ValueError` exception. Exactly what this means will be clear after you read chapter 14, "Exceptions."

`count` is used identically to any of the previous four functions but returns the number of non-overlapping times the given substring occurs in the given string:

```
>>> x = "Mississippi"
>>> x.count("ss")
2
```

You can use two other string methods to search strings: `startswith` and `endswith`. These methods return a `True` or `False` result depending on whether the string they're used on starts or ends with one of the strings given as parameters:

```
>>> x = "Mississippi"
>>> x.startswith("Miss")
True
>>> x.startswith("Mist")
False
>>> x.endswith("pi")
True
>>> x.endswith("p")
False
```

Both `startswith` and `endswith` can look for more than one string at a time. If the parameter is a tuple of strings, both methods check for all of the strings in the tuple and return a `True` if any one of them is found:

```
>>> x.endswith(("i", "u"))
True
```

`startswith` and `endswith` are useful for simple searches.

6.4.5 Modifying strings

Strings are immutable, but string objects have a number of methods that can operate on that string and return a new string that's a modified version of the original string. This provides much the same effect as direct modification for most purposes. You can find a more complete description of these methods in the documentation.

You can use the `replace` method to replace occurrences of `substring` (its first argument) in the string with `newstring` (its second argument). It also takes an optional third argument (see the documentation for details):

```
>>> x = "Mississippi"
>>> x.replace("ss", "+++")
'Mi+++i+++ippi'
```

As with the string search functions, the `re` module provides a much more powerful method of substring replacement.

The functions `string.maketrans` and `string.translate` may be used together to translate characters in strings into different characters. Although rarely used, these functions can simplify your life when they're needed.

Let's say, for example, that you're working on a program that translates string expressions from one computer language into another. The first language uses `~` to mean logical not, whereas the second language uses `!`; the first language uses `^` to mean logical and, whereas the second language uses `&`; the first language uses `(` and `)`, where the second language uses `[` and `]`. In a given string expression, you need to change all instances of `~` to `!`, all instances of `^` to `&`, all instances of `(` to `[`, and all instances of `)` to `]`. You could do this using multiple invocations of `replace`, but an easier and more efficient way is

```
>>> x = "~x ^ (y % z)"
>>> table = x.maketrans("~^()", "!&[]")
>>> x.translate(table)
'!x & [y % z]'
```

The first line uses `maketrans` to make up a translation table from its two string arguments. The two arguments must each contain the same number of characters, and a table will be made such that looking up the *n*th character of the first argument in that table gives back the *n*th character of the second argument.

Next, the table produced by `maketrans` is passed to `translate`. Then, `translate` goes over each of the characters in its `string` object and checks to see if they can be found in the table given as the second argument. If a character can be found in the translation table, `translate` replaces that character with the corresponding character looked up in the table, to produce the translated string.

You can give an optional argument to `translate`, to specify characters that should be removed from the string entirely. See the documentation for details.

Other functions in the `string` module perform more specialized tasks. `string.lower` converts all alphabetic characters in a string to lowercase, and `upper` does the opposite. `capitalize` capitalizes the first character of a string, and `title` capitalizes all words in a string. `swapcase` converts lowercase characters to uppercase and uppercase to lowercase in the same string. `expandtabs` gets rid of tab characters in a string by replacing each tab with a specified number of spaces. `ljust`, `rjust`, and `center` pad a string with spaces, to justify it in a certain field width. `zfill` left-pads a numeric string with zeros. Refer to the documentation for details of these methods.

6.4.6 Modifying strings with list manipulations

Because strings are immutable objects, there's no way to directly manipulate them in the same way you can lists. Although the operations that operate on strings to produce new strings (leaving the original strings unchanged) are useful for many things, sometimes you want to be able to manipulate a string as if it were a list of characters. In that case, just turn it into a list of characters, do whatever you want, and turn the resulting list back into a string:

```
>>> text = "Hello, World"
>>> wordList = list(text)
>>> wordList[6:] = []
>>> wordList.reverse()
>>> text = "".join(wordList)
>>> print(text)
,olleH
```

← Removes everything after comma

← Joins with no space between

Although you can use `split` to turn your string into a list of characters, the type-conversion function `list` is easier to use and to remember (and, for what it's worth, you can turn a string into a tuple of characters using the built-in `tuple` function). To turn the list back into a string, use `"".join`.

You shouldn't go overboard with this method because it causes the creation and destruction of new `string` objects, which is relatively expensive. Processing hundreds or thousands of strings in this manner probably won't have much of an impact on your program. Processing millions probably will.

6.4.7 Useful methods and constants

`string` objects also have several useful methods to report qualities of the string, whether it consists of digits or alphabetic characters, is all uppercase or lowercase, and so on:

```
>>> x = "123"
>>> x.isdigit()
True
>>> x.isalpha()
False
>>> x = "M"
>>> x.islower()
False
>>> x.isupper()
True
```

For a fuller list of all the possible string methods, refer to the string section of the official Python documentation.

Finally, the `string` module defines some useful constants. You've already seen `string.whitespace`, which is a string made up of the characters Python thinks of as whitespace on your system. `string.digits` is the string `'0123456789'`. `string.hexdigits` includes all the characters in `string.digits`, as well as `'abcdefABCDEF'`, the extra characters used in hexadecimal numbers. `string.octdigits` contains

'01234567'—just those digits used in octal numbers. `ascii_string.lowercase` contains all lowercase alphabetic characters; `ascii_string.uppercase` contains all uppercase alphabetic characters; `ascii_string.letters` contains all of the characters in `ascii_string.lowercase` and `ascii_string.uppercase`. You might be tempted to try assigning to these constants to change the behavior of the language. Python would let you get away with this, but it would probably be a bad idea.

Remember that strings are sequences of characters, so you can use the convenient Python `in` operator to test for a character's membership in any of these strings, although usually the existing string methods is simpler and easier.

The most common string operations are shown in table 6.2.

Table 6.2 String operations

String operation	Explanation	Example
<code>+</code>	Adds two strings together	<code>x = "hello " + "world"</code>
<code>*</code>	Replicates a string	<code>x = " " * 20</code>
<code>upper</code>	Converts a string to uppercase	<code>x.upper()</code>
<code>lower</code>	Converts a string to lowercase	<code>x.lower()</code>
<code>title</code>	Capitalizes the first letter of each word in a string	<code>x.title()</code>
<code>find</code> , <code>index</code>	Searches for the target in a string	<code>x.find(y)</code> <code>x.index(y)</code>
<code>rfind</code> , <code>rindex</code>	Searches for the target in a string, from the end of the string	<code>x.rfind(y)</code> <code>x.rindex(y)</code>
<code>startswith</code> , <code>endswith</code>	Checks the beginning or end of a string for a match	<code>x.startswith(y)</code> <code>x.endswith(y)</code>
<code>replace</code>	Replaces the target with a new string	<code>x.replace(y, z)</code>
<code>strip</code> , <code>rstrip</code> , <code>lstrip</code>	Removes whitespace or other characters from the ends of a string	<code>x.strip()</code>
<code>encode</code>	Converts a Unicode string to a bytes object	<code>x.encode("utf_8")</code>

Note that these methods don't change the string itself but return either a location in the string or a new string.

6.5 *Converting from objects to strings*

In Python, almost anything can be converted to some sort of a string representation, using the built-in `repr` function. Lists are the only complex Python data types you're familiar with so far, so let's turn some lists into their representations:

```
>>> repr([1, 2, 3])
'[1, 2, 3]'
```

```
>>> x = [1]
>>> x.append(2)
>>> x.append([3, 4])
>>> 'the list x is ' + repr(x)
'the list x is [1, 2, [3, 4]]'
```

The example uses `repr` to convert the list `x` into a string representation, which is then concatenated with the other string to form the final string. Without the use of `repr`, this wouldn't work. In an expression like `"string" + [1, 2] + 3`, are you trying to add strings, or add lists, or just add numbers? Python doesn't know what you want in such a circumstance, and it will do the safe thing (raise an error) rather than make any assumptions. In the previous example, all the elements had to be converted to string representations before the string concatenation would work.

Lists are the only complex Python objects that have been described to this point, but `repr` can be used to obtain some sort of string representation for almost any Python object. To see this, try `repr` around a built-in complex object—an actual Python function:

```
>>> repr(len)
'<built-in function len>'
```

Python hasn't produced a string containing the code that implements the `len` function, but it has at least returned a string—`<built-in function len>`—that describes what that function is. If you keep the `repr` function in mind and try it on each Python data type (dictionaries, tuples, classes, and the like) as we get to them in the book, you'll see that no matter what type of Python object you have, you can get a string saying something about that object.

This is great for debugging programs. If you're in doubt as to what's held in a variable at a certain point in your program, use `repr` and print out the contents of that variable.

We've covered how Python can convert any object into a string that describes that object. The truth is, Python can do this in either of two different ways. The `repr` function always returns what might be loosely called the *formal string representation* of a Python object. More specifically, `repr` returns a string representation of a Python object from which the original object can be rebuilt. For large, complex objects, this may not be the sort of thing you wish to see in debugging output or status reports.

Python also provides the built-in `str` function. In contrast to `repr`, `str` is intended to produce *printable* string representations, and it can be applied to any Python object. `str` returns what might be called the *informal string representation* of the object. A string returned by `str` need not define an object fully and is intended to be read by humans, not by Python code.

You won't notice any difference between `repr` and `str` when you first start using them, because until you begin using the object-oriented features of Python, there is no difference. `str` applied to any built-in Python object always calls `repr` to calculate its result. It's only when you start defining your own classes that the difference between `str` and `repr` becomes important. This will be discussed in chapter 15.

So why talk about this now? Basically, I wanted you to be aware that there's more going on behind the scenes with `repr` than being able to easily write `print` functions for debugging. As a matter of good style, you may want to get into the habit of using `str` rather than `repr` when creating strings for displaying information.

6.6 Using the format method

You can format strings in Python 3 in two ways. The newer way to format strings in Python is to use the string class's `format` method. The `format` method combines a format string containing replacement fields marked with `{ }` with replacement values taken from the parameters given to the `format` command. If you need to include a literal `{` or `}` in the string, you double it to `{{` or `}}`. The `format` command is a powerful string-formatting mini-language and offers almost endless possibilities for manipulating string formatting. On the other hand, it's fairly simple to use for the most common use cases, so we'll look at a few basic patterns. Then, if you need to use the more advanced options, you can refer to the string-formatting section of the standard library documentation.

6.6.1 The format method and positional parameters

The simplest use of the string `format` method uses numbered replacement fields that correspond to the parameters passed to the `format` function:

```
>>> "{0} is the {1} of {2}".format("Ambrosia", "food", "the gods")  ← ❶
'Ambrosia is the food of the gods'
>>> "{{Ambrosia}} is the {0} of {1}".format("food", "the gods")  ← ❷
'{Ambrosia} is the food of the gods'
```

Note that the `format` method is applied to the format string, which can also be a string variable ❶. Doubling the `{ }` characters escapes them so that they don't mark a replacement field ❷.

This example has three replacement fields, `{0}`, `{1}`, and `{2}`, which are in turn filled by the first, second, and third parameters. No matter where in the format string we place `{0}`, it will always be replaced by the first parameter, and so on.

You can also use the positional parameters.

6.6.2 The format method and named parameters

The `format` method also recognizes named parameters and replacement fields:

```
>>> "{food} is the food of {user}".format(food="Ambrosia",
...     user="the gods")
'Ambrosia is the food of the gods'
```

In this case, the replacement parameter is chosen by matching the name of the replacement field to the name of the parameter given to the `format` command.

You can also use both positional and named parameters, and you can even access attributes and elements within those parameters:

```
>>> "{0} is the food of {user[1]}".format("Ambrosia",
```

```
...         user=["men", "the gods", "others"])
'Ambrosia is the food of the gods'
```

In this case, the first parameter is positional, and the second, `user[1]`, refers to the second element of the named parameter `user`.

6.6.3 Format specifiers

Format specifiers let you specify the result of the formatting with even more power and control than the formatting sequences of the older style of string formatting. The format specifier lets you control the fill character, alignment, sign, width, precision, and type of the data when it's substituted for the replacement field. As noted earlier, the syntax of format specifiers is a mini-language in its own right and too complex to cover completely here, but the following examples give you an idea of its usefulness:

```
>>> "{0:10} is the food of gods".format("Ambrosia")
'Ambrosia  is the food of gods'
>>> "{0:{1}} is the food of gods".format("Ambrosia", 10)
'Ambrosia  is the food of gods'
>>> "{food:{width}} is the food of gods".format(food="Ambrosia", width=10)
'Ambrosia  is the food of gods'
>>> "{0:>10} is the food of gods".format("Ambrosia")
'  Ambrosia is the food of gods'
>>> "{0:&>10} is the food of gods".format("Ambrosia")
'&&Ambrosia is the food of gods'
```

`:10` is a format specifier that makes the field 10 spaces wide and pads with spaces ❶. `:{1}` takes the width from the second parameter ❷. `:>10` forces right justification of the field and pads with spaces ❸. `:&>10` forces right justification and pads with `&` instead of spaces ❹.

6.7 Formatting strings with %

This section covers formatting strings with the *string modulus* (`%`) operator. It's used to combine Python values into formatted strings for printing or other use. C users will notice a strange similarity to the `printf` family of functions. The use of `%` for string formatting is the old style of string formatting, and I cover it here because it was the standard in earlier versions of Python and you're likely to see it in code that's been ported from earlier versions of Python or was written by coders familiar with those versions. This style of formatting shouldn't be used in new code, because it's slated to be deprecated and then removed from the language in the future.

Here's an example:

```
>>> "%s is the %s of %s" % ("Ambrosia", "food", "the gods")
'Ambrosia is the food of the gods'
```

The string modulus operator (the bold `%` that occurs in the middle, not the three instances of `%s` that come before it in the example) takes two parts: the left side, which is a string; and the right side, which is a tuple. The string modulus operator scans the left string for special *formatting sequences* and produces a new string by substituting the

values on the right side for those formatting sequences, in order. In this example, the only formatting sequences on the left side are the three instances of `%s`, which stands for “stick a string in here.”

Passing in different values on the right side produces different strings:

```
>>> "%s is the %s of %s" % ("Nectar", "drink", "gods")
'Nectar is the drink of gods'
>>> "%s is the %s of the %s" % ("Brussels Sprouts", "food",
...    "foolish")
'Brussels Sprouts is the food of the foolish'
```

The members of the tuple on the right will have `str` applied to them automatically by `%s`, so they don’t have to already be strings:

```
>>> x = [1, 2, "three"]
>>> "The %s contains: %s" % ("list", x)
"The list contains: [1, 2, 'three']"
```

6.7.1 Using formatting sequences

All formatting sequences are substrings contained in the string on the left side of the central `%`. Each formatting sequence begins with a percent sign and is followed by one or more characters that specify what is to be substituted for the formatting sequence and how the substitution is accomplished. The `%s` formatting sequence used previously is the simplest formatting sequence, and it indicates that the corresponding string from the tuple on the right side of the central `%` should be substituted in place of the `%s`.

Other formatting sequences can be more complex. This one specifies the field width (total number of characters) of a printed number to be six, specifies the number of characters after the decimal point to be two, and left-justifies the number in its field. I’ve put in angle brackets so you can see where extra spaces are inserted into the formatted string:

```
>>> "Pi is <%-6.2f>" % 3.14159 # use of the formatting sequence: %-6.2f
'Pi is <3.14  >'
```

All the options for characters that are allowable in formatting sequences are given in the documentation. There are quite a few options, but none are particularly difficult to use. Remember, you can always try a formatting sequence interactively in Python to see if it does what you expect it to do.

6.7.2 Named parameters and formatting sequences

Finally, one additional feature is available with the `%` operator that can be useful in certain circumstances. Unfortunately, to describe it we’re going to have to employ a Python feature we haven’t used yet—dictionaries, commonly called *hashtables* or *associative arrays* by other languages. You can skip ahead to the next chapter, “Dictionaries,” to learn about dictionaries, skip this section for now and come back to it later, or read straight through, trusting to the examples to make things clear.

Formatting sequences can specify what should be substituted for them by name rather than by position. When you do this, each formatting sequence has a name in parentheses, immediately following the initial % of the formatting sequence, like so:

```
"%(pi).2f"
```

← **Note name in parentheses**

In addition, the argument to the right of the % operator is no longer given as a single value or tuple of values to be printed but rather as a dictionary of values to be printed, with each named formatting sequence having a correspondingly named key in the dictionary. Using the previous formatting sequence with the string modulus operator, we might produce code like this:

```
>>> num_dict = {'e': 2.718, 'pi': 3.14159}
>>> print("%(pi).2f - %(pi).4f - %(e).2f" % num_dict)
3.14 - 3.1416 - 2.72
```

This is particularly useful when you're using format strings that perform a large number of substitutions, because you no longer have to keep track of the positional correspondences of the right-side tuple of elements with the formatting sequences in the format string. The order in which elements are defined in the `dict` argument is irrelevant, and the template string may use values from `dict` more than once (as it does with the `'pi'` entry).

Controlling output with the print function

Python's built-in `print` function also has some options that can make handling simple string output easier. When used with one parameter, `print` prints the value and a newline character, so that a series of calls to `print` print each value on a separate line:

```
>>> print("a")
a
>>> print("b")
b
```

But `print` can do more than that. You can also give the `print` function a number of arguments, and they will be printed on the same line, separated by a space and ending with a newline:

```
>>> print("a", "b", "c")
a b c
```

If that's not quite what you need, you can give the `print` function additional parameters to control what separates each item and what ends the line:

```
>>> print("a", "b", "c", sep="|")
a|b|c
>>> print("a", "b", "c", end="\n\n")
a b c

>>>
```

In chapter 12, you'll also see that the `print` function can be used to print to files as well as console output.

Using the `print` function's options gives you enough control for simple text output, but more complex situations are best served by using the `format` method.

6.8 Bytes

A `bytes` object is similar to a `string` object but with an important difference. A `string` is an immutable sequence of Unicode characters, whereas a `bytes` object is a sequence of integers with values from 0 to 255. Bytes can be necessary when you're dealing with binary data—for example, reading from a binary data file.

The key thing to remember is that `bytes` objects may look like strings, but they can't be used exactly like a string and they can't be combined with strings:

```
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> xb = unicode_a_with_acute.encode()
>>> xb
b'\xc3\xa1'
>>> xb += 'A'
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    xb += 'A'
TypeError: can't concat bytes to str
>>> xb.decode()
'á'
```

The first thing you can see is that to convert from a regular (Unicode) string to `bytes`, you need to call the string's `encode` method ❶. After it's encoded to a `bytes` object, the character is now 2 bytes and no longer prints the same way the string did ❷. Further, if you attempt to add a `bytes` object and a string object together, you get a type error, because the two are incompatible types ❸. Finally, to convert a `bytes` object back to a string, you need to call that object's `decode` method ❹.

Most of the time, you shouldn't need to think about Unicode or bytes at all. But when you need to deal with international character sets, an increasingly common issue, you must understand the difference between regular strings and `bytes`.

6.9 Summary

Python's string type gives you several powerful tools for text processing. Almost all of those tools are the methods attached to any `string` object, although there's an even more powerful set of tools in the `re` module. The standard string methods can search and replace, trim off extra characters, change case, and much more. Because strings are immutable—that is, they can't be changed—the operations that “change” strings return a copy with the changes, but the original remains untouched.

After lists and strings, the next important Python data structure to consider is the dictionary, before we move on to control structures.



Dictionaries

This chapter covers

- Defining a dictionary
- Using dictionary operations
- Determining what can be used as a key
- Creating sparse matrices
- Using dictionaries as caches
- Trusting the efficiency of dictionaries

This chapter discusses dictionaries, Python’s name for associative arrays, which it implements using hash tables. Dictionaries are amazingly useful, even in simple programs.

Because dictionaries are less familiar to many programmers than other basic data structures such as lists and strings, some of the examples illustrating dictionary use are slightly more complex than the corresponding examples for other built-in data structures. It may be necessary to read parts of the next chapter, “Control flow,” to fully understand some of the examples in this chapter.

7.1 What is a dictionary?

If you've never used associative arrays or hash tables in other languages, then a good way to start understanding the use of dictionaries is to compare them with lists:

- Values in lists are accessed by means of integers called *indices*, which indicate where in the list a given value is found.
- Dictionaries access values by means of integers, strings, or other Python objects called *keys*, which indicate where in the dictionary a given value is found. In other words, both lists and dictionaries provide indexed access to arbitrary values, but the set of items that can be used as dictionary indices is much larger than, and contains, the set of items that can be used as list indices. Also, the mechanism that dictionaries use to provide indexed access is quite different than that used by lists.
- Both lists and dictionaries can store objects of any type.
- Values stored in a list are implicitly *ordered* by their position in the list, because the indices that access these values are consecutive integers; you may or may not care about this ordering, but you can use it if desired. Values stored in a dictionary aren't implicitly ordered relative to one another because dictionary keys aren't just numbers. Note that if you're using a dictionary, you can define an ordering on the items in a dictionary by using another data structure (often a list) to store such an ordering explicitly; this doesn't change the fact that dictionaries have no implicit (built-in) ordering.

In spite of the differences between them, use of dictionaries and lists often appears alike. As a start, an empty dictionary is created much like an empty list, but with curly braces instead of square brackets:

```
>>> x = []
>>> y = {}
```

Here, the first line creates a new, empty list and assigns it to `x`. The second creates a new, empty dictionary, and assigns it to `y`.

After you create a dictionary, values may be stored in it as if it were a list:

```
>>> y[0] = 'Hello'
>>> y[1] = 'Goodbye'
```

Even in these assignments, there is already a significant operational difference between the dictionary and list usage. Trying to do the same thing with a list would result in an error, because in Python it's illegal to assign to a position in a list that doesn't already exist. For example, if we try to assign to the 0th element of the list `x`, we receive an error:

```
>>> x[0] = 'Hello'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

This isn't a problem with dictionaries; new positions in dictionaries are created as necessary.

Having stored some values in the dictionary, we can now access and use them:

```
>>> print(y[0])
Hello
>>> y[1] + ", Friend."
'Goodbye, Friend.'
```

All in all, this makes a dictionary look pretty much like a list. Now for the big difference; let's store (and use) some values under keys that aren't integers:

```
>>> y["two"] = 2
>>> y["pi"] = 3.14
>>> y["two"] * y["pi"]
6.2800000000000002
```

This is definitely something that can't be done with lists! Whereas list indices must be integers, dictionary keys are much less restricted—they may be numbers, strings, or one of a wide range of other Python objects. This makes dictionaries a natural for jobs that lists can't do. For example, it makes more sense to implement a telephone directory application with dictionaries than with lists, because the phone number for a person can be stored indexed by that person's last name.

7.1.1 Why dictionaries are called dictionaries

A dictionary is a way of mapping from one set of arbitrary objects to an associated but equally arbitrary set of objects. Actual dictionaries, thesauri, or translation books are a good analogy in the real world. To see how natural this correspondence is, here is the start of an English-to-French color translator:

```
>>> english_to_french = {}
>>> english_to_french['red'] = 'rouge'
>>> english_to_french['blue'] = 'bleu'
>>> english_to_french['green'] = 'vert'
>>> print("red is", english_to_french['red'])
red is rouge
```

7.2 Other dictionary operations

Besides basic element assignment and access, dictionaries support a number of other operations. You can define a dictionary explicitly as a series of key/value pairs separated by commas:

```
>>> english_to_french = {'red': 'rouge', 'blue': 'bleu', 'green': 'vert'}
```

`len` returns the number of entries in a dictionary:

```
>>> len(english_to_french)
3
```

You can obtain all the keys in the dictionary with the `keys` method. This is often used to iterate over the contents of a dictionary using Python's `for` loop, described in chapter 8:

```
>>> list(english_to_french.keys())
['green', 'blue', 'red']
```

The order of the keys in a list returned by `keys` has no meaning—they aren't necessarily sorted, and they don't necessarily occur in the order they were created. Your Python may print out the keys in a different order than my Python did. If you need keys sorted, you can store them in a list variable and then sort that list.

It's also possible to obtain all the values stored in a dictionary, using `values`:

```
>>> list(english_to_french.values())
['vert', 'bleu', 'rouge']
```

This method isn't used nearly as often as `keys`.

You can use the `items` method to return all keys and their associated values as a sequence of tuples:

```
>>> list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
```

Like `keys`, this is often used in conjunction with a `for` loop to iterate over the contents of a dictionary.

The `del` statement can be used to remove an entry (key/value pair) from a dictionary:

```
>>> list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
>>> del english_to_french['green']
>>> list(english_to_french.items())
[('blue', 'bleu'), ('red', 'rouge')]
```

Dictionary view objects

The `keys`, `values`, and `items` methods return not lists but rather views that behave like sequences but are dynamically updated whenever the dictionary changes. That's why we need to use the `list` function to make them appear as a list in our examples. Otherwise, they behave like sequences, allowing code to iterate over them in a `for` loop, use `in` to check membership in them, and so on.

The view returned by `keys` (and in some cases the view returned by `items`) also behaves like a set, with union, difference, and intersection operations.

Attempting to access a key that isn't in a dictionary is an error in Python. To handle this, you can test the dictionary for the presence of a key with the `in` keyword, which returns `True` if a dictionary has a value stored under the given key and `False` otherwise:

```
>>> 'red' in english_to_french
True
>>> 'orange' in english_to_french
False
```

Alternatively, you can use the `get` function. It returns the value associated with a key, if the dictionary contains that key, but returns its second argument if the dictionary doesn't contain the key:

```
>>> print(english_to_french.get('blue', 'No translation'))
bleu
>>> print(english_to_french.get('chartreuse', 'No translation'))
No translation
```

The second argument is optional. If it isn't included, `get` returns `None` if the dictionary doesn't contain the key.

Similarly, if you want to safely get a key's value *and* make sure it's set to a default in the dictionary, you can use the `setdefault` method:

```
>>> print(english_to_french.setdefault('chartreuse', 'No translation'))
No translation
```

The difference between `get` and `setdefault` is that after the `setdefault` call, there is a key in the dictionary `'chartreuse'` with the value `'No translation'`.

You can obtain a copy of a dictionary using the `copy` method:

```
>>> x = {0: 'zero', 1: 'one'}
>>> y = x.copy()
>>> y
{0: 'zero', 1: 'one'}
```

This makes a shallow copy of the dictionary. This will likely be all you need in most situations. For dictionaries that contain any modifiable objects as values (that is, lists or other dictionaries), you may want to make a deep copy using the `copy.deepcopy` function. See “Nested lists and deep copies” (section 5.6) of “Lists, tuples, and sets” (chapter 5) for an introduction to the concept of shallow and deep copies.

The `update` method updates a first dictionary with all the key/value pairs of a second dictionary. For keys that are common to both, the values from the second dictionary override those of the first:

```
>>> z = {1: 'One', 2: 'Two'}
>>> x = {0: 'zero', 1: 'one'}
>>> x.update(z)
>>> x
{0: 'zero', 1: 'One', 2: 'Two'}
```

Dictionary methods give you a full set of tools to manipulate and use dictionaries. For quick reference, table 7.1 contains some of the main dictionary functions in table format.

Table 7.1 Dictionary operations

Dictionary operation	Explanation	Example
<code>{}</code>	Creates an empty dictionary	<code>x = {}</code>
<code>len</code>	Returns the number of entries in a dictionary	<code>len(x)</code>
<code>keys</code>	Returns a view of all keys in a dictionary	<code>x.keys()</code>

Table 7.1 Dictionary operations (continued)

Dictionary operation	Explanation	Example
<code>values</code>	Returns a view of all values in a dictionary	<code>x.values()</code>
<code>items</code>	Returns a view of all items in a dictionary	<code>x.items()</code>
<code>del</code>	Removes an entry from a dictionary	<code>del(x[key])</code>
<code>in</code>	Tests whether a key exists in a dictionary	<code>'y' in x</code>
<code>get</code>	Returns the value of a key or a configurable default	<code>x.get('y', None)</code>
<code>setdefault</code>	Returns the value if the key is in the dictionary; otherwise, sets the value for the key to the default and returns the value	<code>x.setdefault('y', None)</code>
<code>copy</code>	Makes a copy of a dictionary	<code>y = x.copy()</code>
<code>update</code>	Combines the entries of two dictionaries	<code>x.update(z)</code>

This isn't a complete list of all dictionary operations. For a complete list, see the appendix or refer to the official Python documentation.

7.3 Word counting

Assume that we have a file that contains a list of words, one word per line. We want to know how many times each word occurs in the file. Dictionaries can be used to do this easily:

```
>>> sample_string = "To be or not to be"
>>> occurrences = {}
>>> for word in sample_string.split():
...     occurrences[word] = occurrences.get(word, 0) + 1
...
>>> for word in occurrences:
...     print("The word", word, "occurs", occurrences[word], \
...           "times in the string")
...
```



We increment the `occurrences` count for each word ❶. This is a good example of the power of dictionaries. The code is not only simple, but because dictionary operations are highly optimized in Python, it's also quite fast.

7.4 What can be used as a key?

The previous examples use strings as keys, but Python permits more than just strings to be used in this manner. Any Python object that is immutable and hashable can be used as a key to a dictionary.

In Python, as discussed earlier, any object that can be modified is called *mutable*. Lists are mutable, because list elements can be added, changed, or removed. Dictionaries are also mutable, for the same reason. Numbers are immutable. If a variable `x` is

holding the number 3, and you assign 4 to `x`, you've changed the value in `x`, but you haven't changed the number 3; 3 is still 3. Strings are also immutable. `list[n]` returns the *n*th element of `list`, `string[n]` returns the *n*th character of `string`, and `list[n] = value` changes the *n*th element of `list`, but `string[n] = character` is illegal in Python and causes an error.

Unfortunately, the requirement that keys be immutable and hashable means that lists can't be used as dictionary keys. But there are many instances when it would be convenient to have a listlike key. For example, it's convenient to store information about a person under a key consisting of both their first and last names, which could be easily done if we could use a two-element list as a key.

Python solves this difficulty by providing tuples, which are basically immutable lists—they're created and used similarly to lists, except that when you have them, you can't modify them. But there's one further restriction: keys must also be hashable, which takes things a step further than just immutable. To be hashable, a value must have a hash value (provided by a `__hash__` method) that never changes throughout the life of the value. That means that tuples containing mutable values, although they themselves are immutable, aren't hashable. Only tuples that don't contain any mutable objects nested within them are hashable and valid to use as keys for dictionaries. Table 7.2 illustrates which of Python's built-in types are immutable, hashable, and eligible to be dictionary keys.

Table 7.2 Python values eligible to be used as dictionary keys

Python type	Immutable?	Hashable?	Dictionary key?
<code>int</code>	Yes	Yes	Yes
<code>float</code>	Yes	Yes	Yes
<code>boolean</code>	Yes	Yes	Yes
<code>complex</code>	Yes	Yes	Yes
<code>str</code>	Yes	Yes	Yes
<code>bytes</code>	Yes	Yes	Yes
<code>bytearray</code>	No	No	No
<code>list</code>	No	No	No
<code>tuple</code>	Yes	Sometimes	Sometimes
<code>set</code>	No	No	No
<code>frozenset</code>	Yes	Yes	Yes
<code>dictionary</code>	No	No	No

The next sections give examples illustrating how tuples and dictionaries can work together.

7.5 Sparse matrices

In mathematical terms, a *matrix* is a two-dimensional grid of numbers, usually written in textbooks as a grid with square brackets on each side, as shown at right.

3	0	-2	11
0	9	0	0
0	7	0	0
0	0	0	-5

A fairly standard way to represent such a matrix is by means of a list of lists. In Python, it's presented like this:

```
matrix = [[3, 0, -2, 11], [0, 9, 0, 0], [0, 7, 0, 0], [0, 0, 0, -5]]
```

Elements in the matrix can then be accessed by row and column number:

```
element = matrix[rownum][colnum]
```

But in some applications, such as weather forecasting, it's common for matrices to be very large—thousands of elements to a side, meaning millions of elements in total. It's also common for such matrices to contain many zero elements. In some applications, all but a small percentage of the matrix elements may be set to zero. In order to conserve memory, it's common for such matrices to be stored in a form where only the nonzero elements are actually stored. Such representations are called *sparse matrices*.

It's simple to implement sparse matrices using dictionaries with tuple indices. For example, the previous sparse matrix can be represented as follows:

```
matrix = {(0, 0): 3, (0, 2): -2, (0, 3): 11,
          (1, 1): 9, (2, 1): 7, (3, 3): -5}
```

Now, you can access an individual matrix element at a given row and column number by the following bit of code:

```
if (rownum, colnum) in matrix:
    element = matrix[(rownum, colnum)]
else:
    element = 0
```

A slightly less clear (but more efficient) way of doing this is to use the dictionary `get` method, which you can tell to return `0` if it can't find a key in the dictionary and otherwise return the value associated with that key. This avoids one of the dictionary lookups:

```
element = matrix.get((rownum, colnum), 0)
```

If you're considering doing extensive work with matrices, you may want to look into [NumPy](#), the numeric computation package.

7.6 Dictionaries as caches

The following is an example of how dictionaries can be used as *caches*, data structures that store results to avoid recalculating those results over and over. A short while ago, I wrote a function called `sole`, which took three integers as arguments and returned a result. It looked something like this:

```
def sole(m, n, t):
    # . . . do some time-consuming calculations . . .
    return(result)
```

The problem with this function was that it really was time consuming, and because I was calling `sole` tens of thousands of times, the program ran too slowly.

But `sole` was called with only about 200 different combinations of arguments during any program run. That is, I might call `sole(12, 20, 6)` some 50 or more times during the execution of my program and similarly for many other combinations of arguments. By eliminating the recalculation of `sole` on identical arguments, I'd save a huge amount of time. I used a dictionary with tuples as keys, like so:

```
sole_cache = {}
def sole(m, n, t):
    if (m, n, t) in sole_cache:
        return sole_cache[(m, n, t)]
    else:
        # . . . do some time-consuming calculations . . .
        sole_cache[(m, n, t)] = result
    return result
```

The rewritten `sole` function uses a global variable to store previous results. The global variable is a dictionary, and the keys of the dictionary are tuples corresponding to argument combinations that have been given to `sole` in the past. Then, any time `sole` passes an argument combination for which a result has already been calculated, it returns that stored result, rather than recalculating it.

7.7 Efficiency of dictionaries

If you come from a traditional compiled-language background, you may hesitate to use dictionaries, worrying that they're less efficient than lists (arrays). The truth is that the Python dictionary implementation is quite fast. Many of the internal language features rely on dictionaries, and a lot of work has gone into making them efficient. Because all of Python's data structures are heavily optimized, you shouldn't spend much time worrying about which is faster or more efficient. If the problem can be solved more easily and cleanly by using a dictionary than by using a list, do it that way, and consider alternatives only if it's clear that dictionaries are causing an unacceptable slowdown.

7.8 Summary

Dictionaries are a basic and powerful Python data structure, used for many purposes even within Python itself. The ability to use any immutable object as a key to retrieve a corresponding value makes dictionaries able to handle collections of data with less code and more direct access than many other solutions.

We've now surveyed the main data structures in Python, so the next step is to look at the structures Python has to control the flow of a program.

Control flow

This chapter covers

- Repeating code with a `while` loop
- Making decisions: the `if-elif-else` statement
- Iterating over a list with a `for` loop
- Using list and dictionary comprehensions
- Delimiting statements and blocks with indentation
- Evaluating Boolean values and expressions

Python provides a complete set of control-flow elements, with loops and conditionals. This chapter examines each in detail.

8.1 The while loop

You've come across the basic `while` loop several times already. The full `while` loop looks like this:

```
while condition:
    body
else:
    post-code
```

`condition` is an expression that evaluates to a true or false value. As long as it's `True`, the `body` will be executed repeatedly. If it evaluates to `False`, the `while` loop will execute the `post-code` section and then terminate. If the `condition` starts out by being false, the `body` won't be executed at all—just the `post-code` section. The `body` and `post-code` are each sequences of one or more Python statements that are separated by newlines and are at the same level of indentation. The Python interpreter uses this level to delimit them. No other delimiters, such as braces or brackets, are necessary.

Note that the `else` part of the `while` loop is optional and not often used. That's because as long as there is no `break` in the `body`, this loop

```
while condition:
    body
else:
    post-code
```

and this loop

```
while condition:
    body
post-code
```

do the same things—and the second is simpler to understand. I probably wouldn't have mentioned the `else` clause except that if you haven't learned about it by now, you may have found it confusing if you found this syntax in another person's code. Also, it's useful in some situations.

8.1.1 The `break` and `continue` statements

The two special statements `break` and `continue` can be used in the `body` of a `while` loop. If `break` is executed, it immediately terminates the `while` loop, and not even the `post-code` (if there is an `else` clause) will be executed. If `continue` is executed, it causes the remainder of the `body` to be skipped over; the `condition` is evaluated again, and the loop proceeds as normal.

8.2 The if-elif-else statement

The most general form of the if-then-else construct in Python is

```
if condition1:
    body1
elif condition2:
    body2
elif condition3:
    body3
.
.
.
elif condition(n-1):
    body(n-1)
```

```
else:
    body(n)
```

It says: if `condition1` is `true`, execute `body1`; otherwise, if `condition2` is `true`, execute `body2`; otherwise ... and so on, until it either finds a condition that evaluates to `True` or hits the `else` clause, in which case it executes `body(n)`. As for the `while` loop, the `body` sections are again sequences of one or more Python statements that are separated by newlines and are at the same level of indentation.

Of course, you don't need all that luggage for every conditional. You can leave out the `elif` parts, or the `else` part, or both. If a conditional can't find any body to execute (no conditions evaluate to `True`, and there is no `else` part), it does nothing.

The `body` after the `if` statement is required. But you can use the `pass` statement here (as you can anywhere in Python where a statement is required). The `pass` statement serves as a placeholder where a statement is needed, but it performs no action:

```
if x < 5:
    pass
else:
    x = 5
```

There is no case (or switch) statement in Python.

8.3 The for loop

A `for` loop in Python is different from `for` loops in some other languages. The traditional pattern is to increment and test a variable on each iteration, which is what C `for` loops usually do. In Python, a `for` loop iterates over the values returned by any iterable object—that is, any object that can yield a sequence of values. For example, a `for` loop can iterate over every element in a list, a tuple, or a string. But an iterable object can also be a special function called `range` or a special type of function called a *generator*. This can be quite powerful. The general form is

```
for item in sequence:
    body
else:
    post-code
```

`body` will be executed once for each element of `sequence`. `variable` is set to be the first element of `sequence`, and `body` is executed; then, `variable` is set to be the second element of `sequence`, and `body` is executed; and so on, for each remaining element of the `sequence`.

The `else` part is optional. As with the `else` part of a `while` loop, it's rarely used. `break` and `continue` do the same thing in a `for` loop as in a `while` loop.

This small loop prints out the reciprocal of each number in `x`:

```
x = [1.0, 2.0, 3.0]
for n in x:
    print(1 / n)
```

8.3.1 The range function

Sometimes you need to loop with explicit indices (to use the position at which values occur in a list). You can use the `range` command together with the `len` command on lists to generate a sequence of indices for use by the `for` loop. This code prints out all the positions in a list where it finds negative numbers:

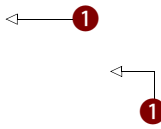
```
x = [1, 3, -7, 4, 9, -5, 4]
for i in range(len(x)):
    if x[i] < 0:
        print("Found a negative number at index ", i)
```

Given a number n , `range(n)` returns a sequence 0, 1, 2, ..., $n-2$, $n-1$. So, passing it the length of a list (found using `len`) produces a sequence of the indices for that list's elements. The `range` function doesn't build a Python list of integers—it just appears to. Instead, it creates a range object that produces integers on demand. This is useful when you're using explicit loops to iterate over really large lists. Instead of building a list with 10 million elements in it, for example, which would take up quite a bit of memory, you can use `range(10000000)`, which takes up only a small amount of memory and generates a sequence of integers from 0 up to 10000000 as needed by the `for` loop.

CONTROLLING STARTING AND STEPPING VALUES WITH RANGE

You can use two variants on the `range` function to gain more control over the sequence it produces. If you use `range` with two numeric arguments, the first argument is the starting number for the resulting sequence and the second number is the number the resulting sequence goes up to (but doesn't include). Here are a few examples:

```
>>> list(range(3, 7))
[3, 4, 5, 6]
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 3))
[]
```



`list()` is used only to force the items `range` would generate to appear as a list. It's not normally used in actual code ❶.

This still doesn't allow you to count backward, which is why the value of `range(5, 3)` is an empty list. To count backward, or to count by any amount other than 1, you need to use the optional third argument to `range`, which gives a step value by which counting proceeds:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

Sequences returned by `range` always include the starting value given as an argument to `range` and never include the ending value given as an argument.

8.3.2 Using `break` and `continue` in `for` loops

The two special statements `break` and `continue` can also be used in the `body` of a `for` loop. If `break` is executed, it immediately terminates the `for` loop, and not even the `post-code` (if there is an `else` clause) will be executed. If `continue` is executed in a `for` loop, it causes the remainder of the `body` to be skipped over, and the loop proceeds as normal with the next item. .

8.3.3 The `for` loop and tuple unpacking

You can use tuple unpacking to make some `for` loops cleaner. The following code takes a list of two-element tuples and calculates the value of the sum of the products of the two numbers in each tuple (a moderately common mathematical operation in some fields):

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0
for t in somelist:
    result = result + (t[0] * t[1])
```

Here's the same thing, but cleaner:

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0

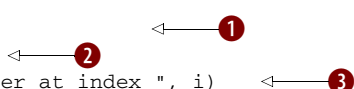
for x, y in somelist:
    result = result + (x * y)
```

We use a tuple `x, y` immediately after the `for` keyword, instead of the usual single variable. On each iteration of the `for` loop, `x` contains element 0 of the current tuple from `list`, and `y` contains element 1 of the current tuple from `list`. Using a tuple in this manner is a convenience of Python, and doing this indicates to Python that each element of the list is expected to be a tuple of appropriate size to unpack into the variable names mentioned in the tuple after the `for`.

8.3.4 The `enumerate` function

You can combine tuple unpacking with the `enumerate` function to loop over both the items and their index. This is similar to using `range` but has the advantage that the code is clearer and easier to understand. Like the previous example, the following code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i, n in enumerate(x):
    if n < 0:
        print("Found a negative number at index ", i)
```




The `enumerate` function returns tuples of (index, item) ❶. You can access the item without the index ❷. The index is also available ❸.

8.3.5 The `zip` function

Sometimes it's useful to combine two or more iterables before looping over them. The `zip` function will take the corresponding elements from one or more iterables and combine them into tuples until it reaches the end of the shortest iterable:

```
>>> x = [1, 2, 3, 4]
>>> y = ['a', 'b', 'c']
>>> z = zip(x, y)
>>> list(z)
[(1, 'a'), (2, 'b'), (3, 'c')]
```



8.4 List and dictionary comprehensions

The pattern of using a `for` loop to iterate through a list, modify or select individual elements, and create a new list or dictionary is very common. Such loops often look a lot like the following:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = []
>>> for item in x:
...     x_squared.append(item * item)
...
>>> x_squared
[1, 4, 9, 16]
```

This sort of situation is so common that Python has a special shortcut for such operations, called a *comprehension*. You can think of a list or dictionary comprehension as a one-line `for` loop that creates a new list or dictionary from another list. The pattern of a list comprehension is as follows:

```
new_list = [expression for variable in old_list if expression]
```

And a dictionary comprehension looks like this:

```
new_dict = {expression:expression for variable in list if expression}
```

In both cases, the heart of the expression is similar to the beginning of a `for` loop—`for variable in list`—with some expression using that variable to create a new key or value and an optional conditional expression using the value of the variable to select whether it's included in the new list or dictionary. For example, the following code does exactly the same thing as the previous code but is a list comprehension:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x]
>>> x_squared
[1, 4, 9, 16]
```

You can even use `if` statements to select items from the list:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x if item > 2]
```

```
>>> x_squared
[9, 16]
```

Dictionary comprehensions are similar, but you need to supply both a key and a value. If we want to do something similar to the previous example but have the number be the key and the number's square be the value in a dictionary, we can use a dictionary comprehension, like so:

```
>>> x = [1, 2, 3, 4]
>>> x_squared_dict = {item: item * item for item in x}
>>> x_squared_dict
{1: 1, 2: 4, 3: 9, 4: 16}
```

List and dictionary comprehensions are very flexible and powerful, and when you get used to them they make list-processing operations much simpler. I recommend that you experiment with them and try them anytime you find yourself writing a `for` loop to process a list of items.

8.5 *Statements, blocks, and indentation*

Because the control flow constructs we encountered in this chapter are the first to make use of blocks and indentation, this is a good time to revisit the subject.

Python uses the indentation of the statements to determine the delimitation of the different blocks (or bodies) of the control-flow constructs. A block consists of one or more statements, which are usually separated by newlines. Examples of Python statements are the assignment statement, function calls, the `print` function, the placeholder `pass` statement, and the `del` statement. The control-flow constructs (`if-elif-else`, `while`, and `for` loops) are compound statements:

```
compound statement clause:
    block
compound statement clause:
    block
```

A compound statement contains one or more clauses that are each followed by indented blocks. Compound statements can appear in blocks just like any other statements. When they do, they create nested blocks.

You may also encounter a couple of special cases. Multiple statements may be placed on the same line if they are separated by semicolons. A block containing a single line may be placed on the same line after the colon of a clause of a compound statement:

```
>>> x = 1; y = 0; z = 0
>>> if x > 0: y = 1; z = 10
... else: y = -1
...
>>> print(x, y, z)
1 1 10
```

Improperly indented code will result in an exception being raised. You may encounter two forms of this. The first is

```
>>>
>>> x = 1
File "<stdin>", line 1
    x = 1
    ^
IndentationError: unexpected indent
>>>
```

We indented a line that should not have been indented. In the basic mode, the carat (^) indicates the spot where the problem occurred. In the IDLE Python Shell (see figure 8.1), the invalid indent is highlighted. The same message would occur if we didn't indent where necessary (that is, the first line after a compound statement clause).

One situation where this can occur can be confusing. If you're using an editor that displays tabs in four-space increments (or the Windows interactive mode, which indents the first tab only four spaces from the prompt) and indent one line with four spaces and then the next line with a tab, the two lines may appear to be at the same level of indentation. But you'll receive this exception because Python maps the tab to eight spaces. The best way to avoid this problem is to use only spaces in Python code. If you must use tabs for indentation, or if you're dealing with code that uses tabs, be sure never to mix them with spaces.

On the subject of the basic interactive mode and the IDLE Python Shell, you likely have noticed that you need an extra line after the outermost level of indentation:

```
>>> x = 1
>>> if x == 1:
...     y = 2
...     if v > 0:
...         z = 2
...         v = 0
...
>>> x = 2
```

No line is necessary after the line `z = 2`, but one is needed after the line `v = 0`. This line is unnecessary if you're placing your code in a module in a file.

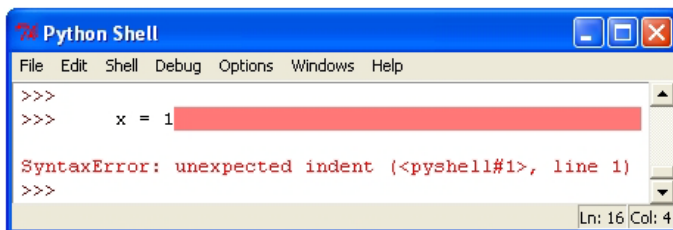


Figure 8.1 Indentation error

The second form of exception will occur if you indent a statement in a block less than the legal amount:

```
>>> x = 1
>>> if x == 1:
    y = 2
    z = 2
File "<stdin>", line 3
    z = 2
    ^
IndentationError: unindent does not match any outer indentation level
```

Here, the line containing `z = 2` isn't lined up properly under the line containing `y = 2`. This form is rare, but I mention it again because in a similar situation, it may be confusing.

Python will allow you to indent any amount and won't complain regardless of how much you vary it as long as you're consistent within a single block. Please don't take improper advantage of this. The recommended standard is to use four spaces for each level of indentation.

Before leaving indentation, I'll cover breaking up statements across multiple lines. This of course is necessary more often as the level of indentation increases. You can explicitly break up a line using the backslash character. You can also implicitly break any statement between tokens when within a set of `()`, `{}`, or `[]` delimiters (that is, when typing a set of values in a list, a tuple, or a dictionary, or a set of arguments in a function call, or any expression within a set of brackets). You can indent the continuation line of a statement to any level you desire:

```
>>> print('string1', 'string2', 'string3' \
...       , 'string4', 'string5')
string1 string2 string3 string4 string5
>>> x = 100 + 200 + 300 \
...     + 400 + 500
>>> x
1500
>>> v = [100, 300, 500, 700, 900,
...      1100, 1300]
>>> v
[100, 300, 500, 700, 900, 1100, 1300]
>>> max(1000, 300, 500,
...      800, 1200)
1200
>>> x = (100 + 200 + 300
...      + 400 + 500)
>>> x
1500
```

You can break a string with a `\` as well. But any indentation tabs or spaces will become part of the string, and the line *must* end with the `\`. To avoid this, you can use the fact that any set of string literals separated by whitespace is automatically concatenated:

```
>>> "strings separated by whitespace " \
...   ""are automatically"" ' concatenated'
'strings separated by whitespace are automatically concatenated'
>>> x = 1
>>> if x > 0:
...     string1 = "this string broken by a backslash will end up \
...               with the indentation tabs in it"
...
>>> string1
'this string broken by a backslash will end up \t\t\twith
  the indentation tabs in it'
>>> if x > 0:
...     string1 = "this can be easily avoided by splitting the " \
...               "string in this way"
...
>>> string1
'this can be easily avoided by splitting the string in this way'
```

8.6 Boolean values and expressions

The previous examples of control flow use conditional tests in a fairly obvious manner but never really explain what constitutes true or false in Python or what expressions can be used where a conditional test is needed. This section describes these aspects of Python.

Python has a Boolean object type that can be set to either `True` or `False`. Any expression with a Boolean operation will return `True` or `False`.

8.6.1 Most Python objects can be used as Booleans

In addition, Python is similar to C with respect to Boolean values, in that C uses the integer 0 to mean false and any other integer to mean true. Python generalizes this idea; 0 or empty values are `False`, and any other values are `True`. In practical terms, this means the following:

- The numbers 0, 0.0, and 0+0j are all `False`; any other number is `True`.
- The empty string `""` is `False`; any other string is `True`.
- The empty list `[]` is `False`; any other list is `True`.
- The empty dictionary `{}` is `False`; any other dictionary is `True`.
- The empty set `set()` is `False`; any other set is `True`.
- The special Python value `None` is always `False`.

There are some Python data structures we haven't looked at yet, but generally the same rule applies. If the data structure is empty or 0, it's taken to mean false in a Boolean context; otherwise it's taken to mean true. Some objects, such as file objects and code objects, don't have a sensible definition of a 0 or empty element, and these objects shouldn't be used in a Boolean context.

8.6.2 *Comparison and Boolean operators*

You can compare objects using normal operators: `<`, `<=`, `>`, `>=`, and so forth. `==` is the equality test operator, and either `!=` or `<>` may be used as the “not equal to” test. There are also `in` and `not in` operators to test membership in sequences (lists, tuples, strings, and dictionaries) as well as `is` and `is not` operators to test whether two objects are the same.

Expressions that return a Boolean value may be combined into more complex expressions using the `and`, `or`, and `not` operators. This code snippet checks to see if a variable is within a certain range:

```
if 0 < x and x < 10:
    ...
```

Python offers a nice shorthand for this particular type of compound statement; you can write it as you would in a math paper:

```
if 0 < x < 10:
    ...
```

Various rules of precedence apply; when in doubt, you can use parentheses to make sure Python interprets an expression the way you want it to. This is probably a good idea for complex expressions, regardless of whether it's necessary, because it makes it clear to future maintainers of the code exactly what's happening. See the appendix for more details on precedence.

The rest of this section provides more advanced information. If this is your first read through this book as you're learning the language, you may want to skip over it.

The `and` and `or` operators return objects. The `and` operator returns either the first false object (that an expression evaluates to) or the last object. Similarly, the `or` operator returns either the first true object or the last object. As with many other languages, evaluation stops as soon as a true expression is found for the `or` operator or as soon as a false expression is found for the `and` operator:

```
>>> [2] and [3, 4]
[3, 4]
>>> [] and 5
[]
>>> [2] or [3, 4]
[2]
>>> [] or 5
5
>>>
```

The `==` and `!=` operators test to see if their operands contains the same values. They are used in most situations. The `is` and `is not` operators test to see if their operands are the same object:

```
>>> x = [0]
>>> y = [x, 1]
>>> x is y[0]
True
>>> x = [0]
>>> x is y[0]
False
>>> x == y[0]
True
```

They reference the same object

x has been assigned to a different object

Revisit “Nested lists and deep copies” (section 5.6) of “Lists, tuples, and sets” (chapter 5) if this example isn’t clear to you.

8.7 Writing a simple program to analyze a text file

To give you a better sense of how a Python program works, let’s look a small sample that roughly replicates the UNIX `wc` utility and reports the number of lines, words, and characters in a file. The sample in listing 8.1 is deliberately written to be clear to programmers new to Python and as simple as possible.

Listing 8.1 word_count.py

```
#!/usr/bin/env python3.1

""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

infile = open('word_count.tst')
lines = infile.read().split("\n")

line_count = len(lines)

word_count = 0
char_count = 0

for line in lines:
    words = line.split()
    word_count += len(words)
    char_count += len(line)

print("File has {0} lines, {1} words, {2} characters".format
      count, word_count, char_count))
```

Opens file

Reads file; splits into lines

Gets number of lines with len()

Initializes other counts

Iterates through lines

Splits into words

Returns number of characters

Prints answers

To test, you can run this against a sample file containing the first paragraph of this chapter's summary, like listing 8.2.

Listing 8.2 word_count.tst

```
Python provides a complete set of control flow elements,  
including while and for loops, and conditionals.  
Python uses the level of indentation to group blocks  
of code with control elements.
```

On running `word_count.py`, you'll get the following output:

```
vern@mac:~/quickpythonbook/code $ python3.1 word_count.py  
File has 4 lines, 30 words, 189 characters
```

This code can give you an idea of a Python program. There isn't much code, and most of the work gets done in three lines of code in the `for` loop. Most Pythonistas see this conciseness as one of Python's great strengths.

8.8 Summary

Python provides a complete set of control-flow elements, including `while` and `for` loops and conditionals. Python uses the level of indentation to group blocks of code with control elements.

Python has the Boolean values `True` and `False`, which can be referenced by variables, but it also considers any 0 or empty value to be false and any nonzero or non-empty value to be true.

Control flow is an important part of programming, but just as important is the ability to package and reuse blocks of code. In the next few chapters, we'll look at ways to do that in Python, beginning with functions.

9 *Functions*

This chapter covers

- Defining functions
- Using function parameters
- Passing mutable objects as parameters
- Understanding local and global variables
- Creating and using lambda expressions
- Using decorators

This chapter assumes you're familiar with function definitions in at least one other computer language and with the concepts that correspond to function definitions, arguments, parameters, and so forth.

9.1 Basic function definitions

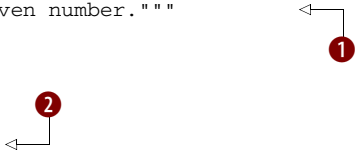
The basic syntax for a Python function definition is

```
def name(parameter1, parameter2, . . .):  
    body
```

As it does with control structures, Python uses indentation to delimit the body of the function definition. The following simple example puts the factorial code from

a previous section into a function body, so we can call a `fact` function to obtain the factorial of a number:

```
>>> def fact(n):
...     """Return the factorial of the given number."""
...     r = 1
...     while n > 0:
...         r = r * n
...         n = n - 1
...     return r
... 
```



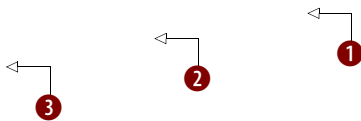
The second line ① is an optional *documentation string* or *docstring*. You can obtain its value by printing `fact.__doc__`. The intention of docstrings is to describe the external behavior of a function and the parameters it takes, whereas comments should document internal information about how the code works. Docstrings are strings that immediately follow the first line of a function definition and are usually triple quoted to allow for multiline descriptions. Browsing tools are available that extract the first line of document strings. It's a standard practice for multiline documentation strings to give a synopsis of the function in the first line, follow this with a blank second line, and end with the rest of the information. This line shows the value after the return is sent back to the code calling the function ②.

Procedure or function?

In some languages, a function that doesn't return a value is called a *procedure*. Although you can (and will) write functions that don't have a `return` statement, they aren't really procedures. All Python procedures are functions; if no explicit `return` is executed in the procedure body, then the special Python value `None` is returned, and if `return arg` is executed, then the value `arg` is immediately returned. Nothing else in the function body is executed once a `return` has been executed. Because Python doesn't have true procedures, we'll refer to both types as *functions*.

Although all Python functions return values, it's up to you whether a function's return value is used:

```
>>> fact(4)
24
>>> x = fact(4)
>>> x
24
>>> 
```



The return value isn't associated with a variable ①. The `fact` function's value is printed in the interpreter only ②. The return value is associated with the variable `x` ③.

9.2 Function parameter options

Most functions need parameters, and each language has its own specifications for how function parameters are defined. Python is flexible and provides three options for defining function parameters. These are outlined in this section.

9.2.1 Positional parameters

The simplest way to pass parameters to a function in Python is by position. In the first line of the function, you specify definition variable names for each parameter; when the function is called, the parameters used in the calling code are matched to the function's parameter variables based on their order. The following function computes x to the power of y :

```
>>> def power(x, y):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
>>> power(3, 3)
27
```

This method requires that the number of parameters used by the calling code exactly match the number of parameters in the function definition, or a `TypeError` exception will be raised:

```
>>> power(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() takes exactly 2 positional arguments (1 given)
>>>
```

Default values

Function parameters can have default values, which you declare by assigning a default value in the first line of the function definition, like so:

```
def fun(arg1, arg2=default2, arg3=default3, . . .)
```

Any number of parameters can be given default values. Parameters with default values must be defined as the last parameters in the parameter list. This is because Python, like most languages, pairs arguments with parameters on a positional basis. There must be enough arguments to a function that the last parameter in that function's parameter list that doesn't have a default value gets an argument. See the next section, "Passing arguments by parameter name," for a more flexible mechanism.

The following function also computes x to the power of y . But if y isn't given in a call to the function, the default value of 2 is used, and the function is just the square function:

```
>>> def power(x, y=2):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
```

You can see the effect of the default argument in the following interactive session:

```
>>> power(3, 3)
27
>>> power(3)
9
```

9.2.2 Passing arguments by parameter name

You can also pass arguments into a function using the name of the corresponding function parameter, rather than its position. Continuing with the previous interactive example, we can type

```
>>> power(2, 3)
8
>>> power(3, 2)
9
>>> power(y=2, x=3)
9
```

Because the arguments to `power` in the final invocation of it are named, their order is irrelevant; the arguments are associated with the parameters of the same name in the definition of `power`, and we get back 3^2 . This type of argument passing is called *keyword passing*.

Keyword passing, in combination with the default argument capability of Python functions, can be highly useful when you're defining functions with large numbers of possible arguments, most of which have common defaults. For example, consider a function that's intended to produce a list with information about files in the current directory and that uses Boolean arguments to indicate whether that list should include information such as file size, last modified date, and so forth, for each file. We can define such a function along these lines

```
def list_file_info(size=False, create_date=False, mod_date=False, ...):
    ...get file names...
    if size:
        # code to get file sizes goes here
    if create_date:
        # code to get create dates goes here
    .
    .
    .
    return fileinfostructure
```

and then call it from other code using keyword argument passing to indicate that we want only certain information (in this example, the file size and modification date but *not* the creation date):

```
fileinfo = list_file_info(size=True, mod_date=True)
```

This type of argument handling is particularly suited for functions with very complex behavior, and one place such functions occur is in graphical user interfaces. If you ever use the Tkinter package to build GUIs in Python, you'll find that the use of optional, keyword-named arguments like this is invaluable.

9.2.3 Variable numbers of arguments

Python functions can also be defined to handle variable numbers of arguments. You can do this two different ways. One way handles the relatively familiar case where you wish to collect an unknown number of arguments at the end of the argument list into a list. The other method can collect an arbitrary number of keyword-passed arguments, which have no correspondingly named parameter in the function parameter list, into a dictionary. These two mechanisms are discussed next.

DEALING WITH AN INDEFINITE NUMBER OF POSITIONAL ARGUMENTS

Prefixing the final parameter name of the function with a `*` causes all excess non-keyword arguments in a call of a function (that is, those positional arguments not assigned to another parameter) to be collected together and assigned as a tuple to the given parameter. Here's a simple way to implement a function to find the maximum in a list of numbers.

First, implement the function:

```
>>> def maximum(*numbers):
...     if len(numbers) == 0:
...         return None
...     else:
...         maxnum = numbers[0]
...         for n in numbers[1:]:
...             if n > maxnum:
...                 maxnum = n
...         return maxnum
... 
```

Now, test out the behavior of the function:

```
>>> maximum(3, 2, 8)
8
>>> maximum(1, 5, 9, -2, 2)
9
```

DEALING WITH AN INDEFINITE NUMBER OF ARGUMENTS PASSED BY KEYWORD

An arbitrary number of keyword arguments can also be handled. If the final parameter in the parameter list is prefixed with `**`, it will collect all excess *keyword-passed* arguments into a dictionary. The index for each entry in the dictionary will be the keyword (parameter name) for the excess argument. The value of that entry is the argument

itself. An argument passed by keyword is excess in this context if the keyword by which it was passed doesn't match one of the parameter names of the function.

For example:

```
>>> def example_fun(x, y, **other):
...     print("x: {0}, y: {1}, keys in 'other': {2}".format(x,
...     y, list(other.keys())))
...     other_total = 0
...     for k in other.keys():
...         other_total = other_total + other[k]
...     print("The total of values in 'other' is {0}".format(other_total))
```

Trying out this function in an interactive session reveals that it can handle arguments passed in under the keywords `foo` and `bar`, even though these aren't parameter names in the function definition:

```
>>> example_fun(2, y="1", foo=3, bar=4)
x: 2, y: 1, keys in 'other': ['foo', 'bar']
The total of values in 'other' is 7
```

9.2.4 Mixing argument-passing techniques

It's possible to use all of the argument-passing features of Python functions at the same time, although it can be confusing if not done with care. Rules govern what you can do. See the documentation for the details.

9.3 Mutable objects as arguments

Arguments are passed in by object reference. The parameter becomes a new reference to the object. For immutable objects (such as tuples, strings, and numbers), what is done with a parameter has no effect outside the function. But if you pass in a mutable object (for example, a list, dictionary, or class instance), any change made to the object will change what the argument is referencing outside the function. Reassigning the parameter doesn't affect the argument, as shown in figures 9.1 and 9.2:

```
>>> def f(n, list1, list2):
...     list1.append(3)
...     list2 = [4, 5, 6]
...     n = n + 1
...
>>> x = 5
>>> y = [1, 2]
>>> z = [4, 5]
>>> f(x, y, z)
>>> x, y, z
(5, [1, 2, 3], [4, 5])
```

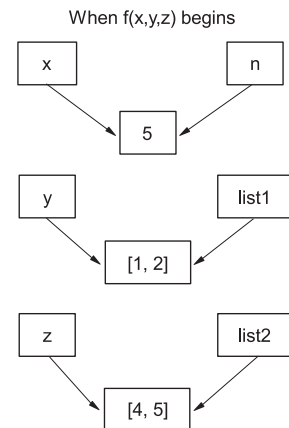
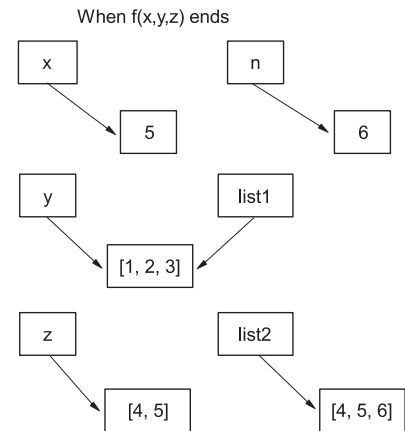


Figure 9.1 At the beginning of function `f()`, both the initial variables and the function parameters refer to the same objects.

Figures 9.1 and 9.2 illustrate what happens when function `f` is called. The variable `x` isn't changed because it's immutable. Instead, the function parameter `n` is set to refer to the new value of 6. Likewise, variable `z` is unchanged because inside function `f`, its corresponding parameter `list2` was set to refer to a new object, `[4, 5, 6]`. Only `y` sees a change because the actual list it points to was changed.

Figure 9.2 At the end of function `f()`, `y` (`list1` inside the function) has been changed internally, whereas `n` and `list2` refer to different objects.



9.4 Local, nonlocal, and global variables

Let's return to our definition of `fact` from the beginning of this chapter:

```
def fact(n):
    """Return the factorial of the given number."""
    r = 1
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

Both the variables `r` and `n` are *local* to any particular call of the factorial function; changes to them made when the function is executing have no effect on any variables outside the function. Any variables in the parameter list of a function, and any variables created within a function by an assignment (like `r = 1` in `fact`), are local to the function.

You can explicitly make a variable global by declaring it so before the variable is used, using the `global` statement. Global variables can be accessed and changed by the function. They exist outside the function and can also be accessed and changed by other functions that declare them global or by code that's not within a function. Let's look at an example to see the difference between local and global variables:

```
>>> def fun():
...     global a
...     a = 1
...     b = 2
... 
```

This defines a function that treats `a` as a global variable and `b` as a local variable and attempts to modify both `a` and `b`.

Now, test this function:

```
>>> a = "one"
>>> b = "two"
```

```
>>> fun()
>>> a
1
>>> b
'two'
```

The assignment to `a` within `fun` is an assignment to the global variable `a` also existing outside of `fun`. Because `a` is designated `global` in `fun`, the assignment modifies that global variable to hold the value `1` instead of the value `"one"`. The same isn't true for `b`—the local variable called `b` inside `fun` starts out referring to the same value as the variable `b` outside of `fun`, but the assignment causes `b` to point to a new value that's local to the function `fun`.

Similar to the `global` statement is the `nonlocal` statement, which causes an identifier to refer to a previously bound variable in the closest enclosing scope. We'll discuss scopes and namespaces in more detail in the next chapter, but the point is that `global` is used for a top-level variable, whereas `nonlocal` can refer to any variable in an enclosing scope, as the example in listing 9.1 illustrates.

Listing 9.1 File `nonlocal.py`

```
g_var = 0
nl_var = 0
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
def test():
    nl_var = 2
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
    def inner_test():
        global g_var
        nonlocal nl_var
        g_var = 1
        nl_var = 4
        print("in inner_test-> g_var: {0} nl_var: {1}".format(g_var,
                                                                nl_var))

    inner_test()
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))

test()
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
```

Diagram illustrating variable binding in Listing 9.1:

- `g_var` in `inner_test` binds top-level `g_var` (indicated by an arrow from the `global g_var` line to the `g_var` in the print statement).
- `nl_var` in `inner_test` binds to `nl_var` in `test` (indicated by an arrow from the `nonlocal nl_var` line to the `nl_var` in the print statement).
- `nl_var` in `inner_test` binds to `nl_var` in `test` (indicated by an arrow from the `nl_var` in the print statement to the `nl_var` in the `test` function definition).

When run, this code prints the following:

```
top level-> g_var: 0 nl_var: 0
in test-> g_var: 0 nl_var: 2
in inner_test-> g_var: 1 nl_var: 4
in test-> g_var: 1 nl_var: 4
top level-> g_var: 1 nl_var: 0
```

Note that the value of the top-level `nl_var` hasn't been affected, which would happen if `inner_test` contained the line `global nl_var`.

The bottom line is that if you want to assign to a variable existing outside a function, you must explicitly declare that variable to be nonlocal or global. But if you're accessing a variable that exists outside the function, you don't need to declare it nonlocal or global. If Python can't find a variable name in the local function scope, it will attempt to look up the name in the global scope. Hence, accesses to global variables will automatically be sent through to the correct global variable. Personally, I don't recommend using this shortcut. It's much clearer to a reader if all global variables are explicitly declared as global. Further, you probably want to limit the use of global variables within functions to only rare occasions.

9.5 Assigning functions to variables

Functions can be assigned, like other Python objects, to variables, as shown in the following example:

```
>>> def f_to_kelvin(degrees_f):
...     return 273.15 + (degrees_f - 32) * 5 / 9
...
>>> def c_to_kelvin(degrees_c):
...     return 273.15 + degrees_c
...
>>> abs_temperature = f_to_kelvin
>>> abs_temperature(32)
273.14999999999998
>>> abs_temperature = c_to_kelvin
>>> abs_temperature(0)
273.14999999999998
```

← Defines a function

← Defines a function

← Assigns function to variable

← Assigns function to variable

You can place them in lists, tuples, or dictionaries:

```
>>> t = {'FtoK': f_to_kelvin, 'CtoK': c_to_kelvin}
>>> t['FtoK'](32)
273.14999999999998
>>> t['CtoK'](0)
273.14999999999998
```

← ①

← Accesses a function as value in dictionary

← Accesses a function as value in dictionary

A variable that refers to a function can be used in exactly the same way as the function ①. This last example shows how you can use a dictionary to call different functions by the value of the strings used as keys. This is a common pattern in situations where different functions need to be selected based on a string value, and in many cases it takes the place of the `switch` structure found in languages like C and Java.

9.6 lambda expressions

Short functions like those you just saw can also be defined using `lambda` expressions of the form

```
lambda parameter1, parameter2, . . . : expression
```

`lambda` expressions are anonymous little functions that you can quickly define inline. Often, a small function needs to be passed to another function, like the key function

used by a list's sort method. In such cases, a large function is usually unnecessary, and it would be awkward to have to define the function in a separate place from where it's used. Our dictionary in the previous subsection can be defined all in one place with

```
>>> t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,
...       'CtoK': lambda deg_c: 273.15 + deg_c}
>>> t2['FtoK'](32)
273.14999999999998
```

This defines `lambda` expressions as values of the dictionary ❶. Note that `lambda` expressions don't have a `return` statement, because the value of the expression is automatically returned.

9.7 Generator functions

A *generator* function is a special kind of function that you can use to define your own iterators. When you define a generator function, you return each iteration's value using the `yield` keyword. When there are no more iterations, an empty `return` statement or flowing off the end of the function ends the iterations. Local variables in a generator function are saved from one call to the next, unlike in normal functions:

```
>>> def four():
...     x = 0
...     while x < 4:
...         print("in generator, x =", x)
...         yield x
...         x += 1
...
>>> for i in four():
...     print(i)
...
in generator, x = 0
0
in generator, x = 1
1
in generator, x = 2
2
in generator, x = 3
3
```

← Sets initial value of x to 0

← Returns current value of x

← Increments value of x

Note that this generator function has a `while` loop that limits the number of times the generator will execute. Depending on how it's used, a generator that doesn't have some condition to halt it could cause an endless loop when called.

You can also use generator functions with `in` to see if a value is in the series that the generator produces:

```
>>> 2 in four()
in generator, x = 0
in generator, x = 1
in generator, x = 2
True
>>> 5 in four()
```

```

in generator, x = 0
in generator, x = 1
in generator, x = 2
in generator, x = 3
False

```

9.8 Decorators

Because functions are first-class objects in Python, they can be assigned to variables, as you’ve seen. Functions can be passed as arguments to other functions and passed back as return values from other functions.

For example, it’s possible to write a Python function that takes another function as its parameter, wrap it in another function that does something related, and then return the new function. This new combination can be used instead of the original function:

```

>>> def decorate(func):
...     print("in decorate function, decorating", func.__name__)
...     def wrapper_func(*args):
...         print("Executing", func.__name__)
...         return func(*args)
...     return wrapper_func
...
>>> def myfunction(parameter):
...     print(parameter)
...
>>> myfunction = decorate(myfunction)
in decorate function, decorating myfunction
>>> myfunction("hello")
Executing myfunction
hello

```

A decorator is syntactic sugar for this process and lets you wrap one function inside another with a one-line addition. This still gives you exactly the same effect as the previous code, but the resulting code is much cleaner and easier to read.

Very simply, using a decorator involves two parts: defining the function that will be wrapping or “decorating” other functions and then using an `@` followed by the decorator immediately before the wrapped function is defined. The decorator function should take a function as a parameter and return a function, as follows:

```

>>> def decorate(func):
...     print("in decorate function, decorating", func.__name__)
...     def wrapper_func(*args):
...         print("Executing", func.__name__)
...         return func(*args)
...     return wrapper_func
...
>>> @decorate
... def myfunction(parameter):
...     print(parameter)
...
in decorate function, decorating myfunction
>>> myfunction("hello")
Executing myfunction
hello

```

The `decorate` function prints the name of the function it's wrapping when the function is defined ❶. When it's finished, the decorator returns the wrapped function ❷. `myfunction` is decorated using `@decorate` ❸. The wrapped function is called after the decorator function has completed ❹.

Using a decorator to wrap one function in another can be handy for a number of purposes. In web frameworks such as Django, decorators are used to make sure a user is logged in before executing a function; and in graphics libraries, decorators can be used to register a function with the graphics framework.

9.9 **Summary**

Defining functions in Python is simple but highly flexible. Although all variables created during the execution of a function body are local to that function, external variables can easily be accessed using the `global` statement.

Python functions provide exceedingly powerful argument-passing features:

- Arguments may be passed by position or by parameter name.
- Default values may be provided for function parameters.
- Functions can collect arguments into tuples, giving you the ability to define functions that take an indefinite number of arguments.
- Functions can collect arguments into dictionaries, giving you the ability to define functions that take an indefinite number of arguments passed by parameter name.

Functions are first-class objects in Python, which means that they can be assigned to variables, accessed by way of variables, and decorated. Functions are essential building blocks for writing readable, structured code. By packaging code that performs a particular function, they make reusing that code easier, and they also make the rest of your code simpler and easier to understand. The next step along this path is packaging functions (and other objects) into `modules`, which is the topic of the next chapter.

10

Modules and scoping rules

This chapter covers:

- Defining a module
- Writing a first module
- Using the `import` statement
- Modifying the module search path
- Making names private in modules
- Importing standard library and third-party modules
- Understanding Python scoping rules and namespaces

Modules are used to organize larger Python projects. The Python standard library is split into modules to make it more manageable. You don't need to organize your own code into modules, but if you're writing any programs that are more than a few pages long, or any code that you want to reuse, you should probably do so.

10.1 What is a module?

A *module* is a file containing code. A module defines a group of Python functions or other objects, and the name of the module is derived from the name of the file.

Modules most often contain Python source code, but they can also be compiled C or C++ object files. Compiled modules and Python source modules are used the same way.

As well as grouping related Python objects, modules help avoid name-clash problems. For example, you might write a module for your program called `mymodule`, which defines a function called `reverse`. In the same program, you might also wish to use somebody else's module called `othermodule`, which also defines a function called `reverse`, but which does something different from your `reverse` function. In a language without modules, it would be impossible to use two different functions named `reverse`. In Python, it's trivial—you refer to them in your main program as `mymodule.reverse` and `othermodule.reverse`.

This is because Python uses *namespaces*. A namespace is essentially a dictionary of the identifiers available to a block, function, class, module, and so on. We'll discuss namespaces a bit more at the end of this chapter, but be aware that each module has its own namespace, and this helps avoid naming conflicts.

Modules are also used to make Python itself more manageable. Most standard Python functions aren't built into the core of the language but instead are provided via specific modules, which you can load as needed.

10.2 A first module

The best way to learn about modules is probably to make one, so let's get started.

Create a text file called `mymath.py`, and in that text file enter the Python code in listing 10.1. (If you're using IDLE, select New Window from the File menu and start typing, as shown in figure 10.1.)

Listing 10.1 File `mymath.py`

```
"""mymath - our example math module"""
pi = 3.14159
def area(r):
    """area(r): return the area of a circle with radius r."""
    global pi
    return(pi * r * r)
```

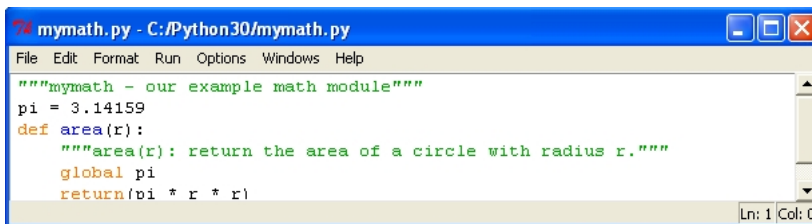


Figure 10.1 An IDLE edit window provides the same editing functionality as the shell window, including automatic indentation and colorization.

Save this for now in the directory where your Python executable is. This code merely assigns `pi` a value and defines a function. The `.py` filename suffix is strongly suggested for all Python code files. It identifies that file to the Python interpreter as consisting of Python source code. As with functions, you have the option of putting in a document string as the first line of your module.

Now, start up the Python Shell and type the following:

```
>>> pi
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> area(2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

In other words, Python doesn't have the constant `pi` or the function `area` built in.

Now, type

```
>>> import mymath
>>> pi
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> mymath.pi
3.1415899999999999
>>> mymath.area(2)
12.56636
>>> mymath.__doc__
'mymath - our example math module'
>>> mymath.area.__doc__
'area(r): return the area of a circle with radius r.'
```

We've brought in the definitions for `pi` and `area` from the `mymath.py` file, using the `import` statement (which automatically adds on the `.py` suffix when it searches for the file defining the module named `mymath`). But the new definitions aren't directly accessible; typing `pi` by itself gave an error, and typing `area(2)` by itself would give an error. Instead, we access `pi` and `area` by *prepending* them with the name of the module that contains them. This guarantees name safety. There may be another module out there that also defines `pi` (maybe the author of that module thinks that `pi` is 3.14 or 3.14159265), but that is of no concern. Even if that other module is imported, its version of `pi` will be accessed by `othermodulename.pi`, which is different from `mymath.pi`. This form of access is often referred to as *qualification* (that is, the variable `pi` is being qualified by the module `mymath`). We may also refer to `pi` as an *attribute* of `mymath`.

Definitions within a module can access other definitions within that module, without prepending the module name. The `mymath.area` function accesses the `mymath.pi` constant as just `pi`.

If you want to, you can also specifically ask for names from a module to be imported in such a manner that you don't have to prepend it with the module name. Type

```
>>> from mymath import pi
>>> pi
3.1415899999999999
>>> area(2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

The name `pi` is now directly accessible because we specifically requested it using `from module import name`.

The function `area` still needs to be called as `mymath.area`, though, because it wasn't explicitly imported.

You may want to use the basic interactive mode or IDLE's Python shell to incrementally test a module as you're creating it. But if you change your module on disk, retyping the `import` command won't cause it to load again. You need to use the `reload` function from the `imp` module for this. The `imp` module provides an interface to the mechanisms behind importing modules:

```
>>> import mymath, imp
>>> imp.reload(mymath)
<module 'mymath' from '/home/doc/quickpythonbook/code/mymath.py'>
```

When a module is reloaded (or imported for the first time), all of its code is parsed. A syntax exception is raised if an error is found. On the other hand, if everything is okay, a `.pyc` file (for example, `mymath.pyc`) containing Python byte code is created.

Reloading a module doesn't put you back into exactly the same situation as when you start a new session and import it for the first time. But the differences won't normally cause you any problems. If you're interested, you can look up `reload` in the section on the `imp` module in the *Python Language Reference* to find the details.

Of course, modules don't need to be used from the interactive Python shell. You can also import them into scripts, or other modules for that matter; enter suitable `import` statements at the beginning of your program file. Internally to Python, the interactive session and a script are considered modules as well.

To summarize:

- A module is a file defining Python objects.
- If the name of the module file is `modulename.py`, then the Python name of the module is `modulename`.
- You can bring a module named `modulename` into use with the `import modulename` statement. After this statement is executed, objects defined in the module can be accessed as `modulename.objectname`.
- Specific names from a module can be brought directly into your program using the `from modulename import objectname` statement. This makes `objectname` accessible to your program without needing to prepend it with `modulename`, and it's useful for bringing in names that are often used.

10.3 The import statement

The `import` statement takes three different forms. The most basic,

```
import modulename
```

searches for a Python module of the given name, parses its contents, and makes it available. The importing code can use the contents of the module, but any references by that code to names within the module must still be prepended with the module name. If the named module isn't found, an error will be generated. Exactly where Python looks for modules will be discussed shortly.

The second form permits specific names from a module to be explicitly imported into the code:

```
from modulename import name1, name2, name3, . . .
```

Each of `name1`, `name2`, and so forth from within `modulename` is made available to the importing code; code after the `import` statement can use any of `name1`, `name2`, `name3`, and so on without prepending the module name.

Finally, there's a general form of the `from . . . import . . .` statement:

```
from modulename import *
```

The `*` stands for all the exported names in `modulename`. This imports all public names from `modulename`—that is, those that don't begin with an underscore, and makes them available to the importing code without the necessity of prepending the module name. But if a list of names called `__all__` exists in the module (or the package's `__init__.py`), then the names are the ones imported, whether they begin with an underscore or not.

You should take care when using this particular form of importing. If two modules both define a name, and you import both modules using this form of importing, you'll end up with a name clash, and the name from the second module will replace the name from the first. It also makes it more difficult for readers of your code to determine where names you're using originate. When you use either of the two previous forms of the import statement, you give your reader explicit information about where they're from.

But some modules (such as `tkinter`, which will be covered later) name their functions to make it obvious where they originate and to make it unlikely that name clashes will occur. It's also common to use the general import to save keystrokes when using an interactive shell.

10.4 The module search path

Exactly where Python looks for modules is defined in a variable called `path`, which you can access through a module called `sys`. Enter the following:

```
>>> import sys
>>> sys.path
_list of directories in the search path_
```

The value shown in place of `_list of directories in the search path_` will depend on the configuration of your system. Regardless of the details, the string indicates a list of directories that Python searches (in order) when attempting to execute an `import` statement. The first module found that satisfies the `import` request is used. If there's no satisfactory module in the module search path, an `ImportError` exception is raised.

If you're using IDLE, you can graphically look at the search path and the modules on it using the Path Browser window, which you can start from File menu of the Python Shell window.

The `sys.path` variable is initialized from the value of the environment (operating system) variable `PYTHONPATH`, if it exists, or from a default value that's dependent on your installation. In addition, whenever you run a Python script, the `sys.path` variable for that script has the directory containing the script inserted as its first element—this provides a convenient way of determining where the executing Python program is located. In an interactive session such as the previous one, the first element of `sys.path` is set to the empty string, which Python takes as meaning that it should first look for modules in the current directory.

10.4.1 Where to place your own modules

In the example that started this chapter, the `mymath` module was accessible to Python because (1) when you execute Python interactively, the first element of `sys.path` is `"`, telling Python to look for modules in the current directory; and (2) you were executing Python in the directory that contained the `mymath.py` file. In a production environment, neither of these conditions will typically be true. You won't be running Python interactively, and Python code files won't be located in your current directory. In order to ensure that your programs can use modules you coded, you need to do one of the following:

- Place your modules into one of the directories that Python normally searches for modules.
- Place all the modules used by a Python program into the same directory as the program.
- Create a directory (or directories) that will hold your modules, and modify the `sys.path` variable so that it includes this new directory.

Of these three options, the first is apparently the easiest and is also an option that you should *never* choose unless your version of Python includes local code directories in its default module search path. Such directories are specifically intended for site-specific code and aren't in danger of being overwritten by a new Python install because they're not part of the Python installation. If your `sys.path` refers to such directories, you can put your modules there.

The second option is a good choice for modules that are associated with a particular program. Just keep them with the program.

The third option is the right choice for site-specific modules that will be used in more than one program at that site. You can modify `sys.path` in various ways. You can

assign to it in your code, which is easy, but doing so hard-codes directory locations into your program code; you can set the `PYTHONPATH` environment variable, which is relatively easy, but it may not apply to all users at your site; or you can add to the default search path using a `.pth` file.

See the section on environment variables in the appendix for examples of how to set `PYTHONPATH`. The directory or directories you set it to are prepended to the `sys.path` variable. If you use it, be careful that you don't define a module with the same name as one of the existing library modules that you're using or is being used for you. Your module will be found before the library module. In some cases, this may be what you want, but probably not often.

You can avoid this issue using the `.pth` method. In this case, the directory or directories you added will be appended to `sys.path`. The last of these mechanisms is best illustrated by a quick example. On Windows, you can place this in the directory pointed to by `sys.prefix`. Assume your `sys.prefix` is `c:\program files\python`, and place the file in listing 10.2 in that directory.

Listing 10.2 File `myModules.pth`

```
mymodules
c:\My Documents\python\modules
```

The next time a Python interpreter is started, `sys.path` will have `c:\program files\python\mymodules` and `c:\My Documents\python\modules` added to it, if they exist. You can now place your modules in these directories. Note that the `mymodules` directory still runs the danger of being overwritten with a new installation. The `modules` directory is safer. You also may have to move or create a `mymodules.pth` file when you upgrade Python. See the description of the `site` module in the *Python Library Reference* if you want more details on using `.pth` files.

10.5 Private names in modules

We mentioned that you can enter `from module import *` to import *almost* all names from a module. The exception to this is that names in the module beginning with an underscore can't be imported in this manner so that people can write modules that are intended for importation with `from module import *`. By starting all internal names (that is, names that shouldn't be accessed outside the module) with an underscore, you can ensure that `from module import *` brings in only those names that the user will want to access.

To see this in action, let's assume we have a file called `modtest.py`, containing the code in listing 10.3.

Listing 10.3 File `modtest.py`

```
"""modtest: our test module"""
def f(x):
    return x
```

```
def _g(x):
    return x
a = 4
_b = 2
```

Now, start up an interactive session, and enter the following:

```
>>> from modtest import *
>>> f(3)
3
>>> _g(3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name '_g' is not defined
>>> a
4
>>> _b
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name '_b' is not defined
```

As you can see, the names `f` and `a` are imported, but the names `_g` and `_b` remain hidden outside of `modtest`. Note that this behavior occurs only with `from ... import *`. We can do the following to access `_g` or `_b`:

```
>>> import modtest
>>> modtest._b
2
>>> from modtest import _g
>>> _g(5)
5
```

The convention of leading underscores to indicate private names is used throughout Python and not just in modules. You'll encounter it in classes and packages, later in the book.

10.6 Library and third-party modules

At the beginning of this chapter, I mentioned that the standard Python distribution is split into modules to make it more manageable. After you've installed Python, all the functionality in these library modules is available to you. All that's needed is to import the appropriate modules, functions, classes, and so forth explicitly, before you use them.

Many of the most common and useful standard modules are discussed throughout this book. But the standard Python distribution includes far more than what this book describes. At the very least, you should browse through the table of contents of the *Python Library Reference*.

In IDLE, you can easily browse to and look at those written in Python using the Path Browser window. You can also search for example code that uses them with the Find in Files dialog box, which you can open from the Edit menu of the Python Shell window. You can search through your own modules as well in this way.

Available third-party modules, and links to them, are identified on the Python home page. You need to download these and place them in a directory in your module search path in order to make them available for import into your programs.

10.7 Python scoping rules and namespaces

Python’s scoping rules and namespaces will become more interesting as your experience as a Python programmer grows. If you’re new to Python, you probably don’t need to do anything more than quickly read through the text to get the basic ideas. For more details, look up “namespaces” in the *Python Language Reference*.

The core concept here is that of a *namespace*. A namespace in Python is a mapping from identifiers to objects and is usually represented as a dictionary. When a block of code is executed in Python, it has three namespaces: *local*, *global*, and *built-in* (see figure 10.2).

When an identifier is encountered during execution, Python first looks in the *local namespace* for it. If it isn’t found, the *global namespace* is looked in next. If it still hasn’t been found, the *built-in namespace* is checked. If it doesn’t exist there, this is considered an error and a `NameError` exception occurs.

For a module, a command executed in an interactive session, or a script running from a file, the global and local namespaces are the same. Creating any variable or function or importing anything from another module results in a new entry, or *binding*, being made in this namespace.

But when a function call is made, a local namespace is created, and a binding is entered in it for each parameter of the call. A new binding is then entered into this local namespace whenever a variable is created within the function. The global namespace of a function is the global namespace of the containing block of the function (that of the module, script file, or interactive session). It’s independent of the dynamic context from which it’s called.

In all of these situations, the built-in namespace is that of the `__builtins__` module. This module contains, among other things, all the built-in functions you’ve encountered (such as `len`, `min`, `max`, `int`, `float`, `long`, `list`, `tuple`, `cmp`, `range`, `str`, and `repr`) and the other built-in classes in Python, such as the exceptions (like `NameError`).

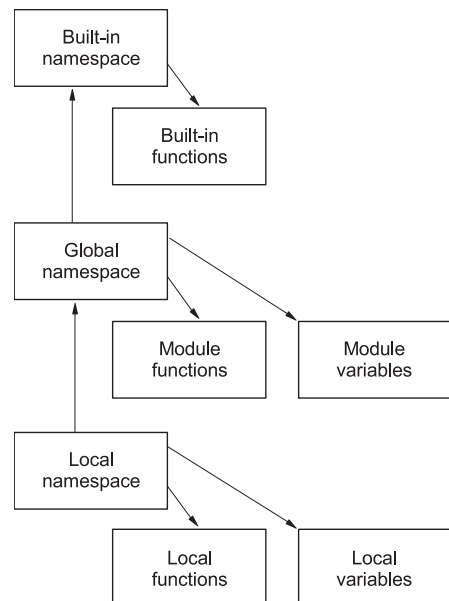


Figure 10.2 The order in which namespaces are checked to locate identifiers

One thing that sometimes catches new Python programmers is the fact that you can override items in the built-in module. If, for example, you create a list in your program and put it in a variable called `list`, you can't subsequently use the built-in `list` function. The entry for your list is found first. There's no differentiation between names for functions and modules and other objects. The most recent occurrence of a binding for a given identifier is used.

Enough talk—it's time to explore this with some examples. We use two built-in functions, `locals` and `globals`. They return dictionaries containing the bindings in the local and global namespaces, respectively.

Start a new interactive session:

```
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
  '__doc__': None, '__package__': None}
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
  '__doc__': None, '__package__': None}>>>
```

The local and global namespaces for this new interactive session are the same. They have three initial key/value pairs that are for internal use: (1) an empty documentation string `__doc__`, (2) the main module name `__name__` (which for interactive sessions and scripts run from files is always `__main__`), and (3) the module used for the built-in namespace `__builtins__` (the module `__builtins__`).

Now, if we continue by creating a variable and importing from modules, we'll see a number of bindings created:

```
>>> z = 2
>>> import math
>>> from cmath import cos
>>> globals()
{'cos': <built-in function cos>, '__builtins__': <module 'builtins'
  (built-in)>, '__package__': None, '__name__': '__main__', 'z': 2,
  '__doc__': None, 'math': <module 'math' from
  '/usr/local/lib/python3.0/libdynload/math.so'>}
>>> locals()
{'cos': <built-in function cos>, '__builtins__':
  <module 'builtins' (built-in)>, '__package__': None, '__name__':
  '__main__', 'z': 2, '__doc__': None, 'math': <module 'math' from
  '/usr/local/lib/python3.0/libdynload/math.so'>}
>>> math.ceil(3.4)
4
```

As expected, the local and global namespaces continue to be equivalent. Entries have been added for `z` as a number, `math` as a module, and `cos` from the `cmath` module as a function.

You can use the `del` statement to remove these new bindings from the namespace (including the module bindings created with the `import` statements):

```
>>> del z, math, cos
>>> locals()
```

```
{'__builtins__': <module 'builtins' (built-in)>, '__package__': None,
 '__name__': '__main__', '__doc__': None}
>>> math.ceil(3.4)
Traceback (innermost last):
  File "<stdin>", line 1, in <module>
NameError: math is not defined
>>> import math
>>> math.ceil(3.4)
4
```

The result isn't drastic, because we're able to import the `math` module and use it again. Using `del` in this manner can be handy when you're in the interactive mode.¹

For the trigger happy, yes, it's also possible to use `del` to remove the `__doc__`, `__main__`, and `__builtins__` entries. But resist doing this, because it wouldn't be good for the health of your session!

Now, let's look at a function created in an interactive session:

```
>>> def f(x):
...     print("global: ", globals())
...     print("Entry local: ", locals())
...     y = x
...     print("Exit local: ", locals())
...
>>> z = 2
>>> globals()
{'f': <function f at 0xb7cbfeac>, '__builtins__': <module 'builtins'
(built-in)>, '__package__': None, '__name__': '__main__', 'z': 2,
 '__doc__': None}
>>> f(z)
global: {'f': <function f at 0xb7cbfeac>, '__builtins__': <module
(built-in)>, '__package__': None, '__name__': '__main__',
 'z': 2, '__doc__': None}
Entry local: {'x': 2}
Exit local: {'y': 2, 'x': 2}
>>>
```

If we dissect this apparent mess, we see that, as expected, upon entry the parameter `x` is the original entry in `f`'s local namespace, but `y` is added later. The global namespace is the same as that of our interactive session, because this is where `f` was defined. Note that it contains `z`, which was defined after `f`.

In a production environment, you normally call functions that are defined in modules. Their global namespace is that of the module they're defined in. Assume that we've created the file in listing 10.4.

Listing 10.4 File `scopetest.py`

```
"""scopetest: our scope test module"""
v = 6
```

¹ Using `del` and then `import` again won't pick up changes made to a module on disk. It isn't removed from memory and then loaded from disk again. The binding is taken out of and then put back into your namespace. You still need to use `imp.reload` if you want to pick up changes made to a file.

```
def f(x):
    """f: scope test function"""
    print("global: ", list(globals().keys()))
    print("entry local:", locals())
    y = x
    w = v
    print("exit local:", list(locals().keys()))
```

Note that we'll be printing only the keys (identifiers) of the dictionary returned by `globals`. This will reduce the clutter in the results. It was necessary in this case due to the fact that in modules as an optimization, the whole `__builtins__` dictionary is stored in the value field for the `__builtins__` key:

```
>>> import scopetest
>>> z = 2
>>> scopetest.f(z)
global:  ['f', '__builtins__', '__file__', '__package__', 'v', '__name__',
➡ '__doc__']
entry local: {'x': 2}
exit local: {'y': 2, 'x': 2, 'w': 6}
```

The global namespace is now that of the `scopetest` module and includes the function `f` and integer `v` (but not `z` from our interactive session). Thus, when creating a module, you have complete control over the namespaces of its functions.

We've now covered local and global namespaces. Next, let's move on to the built-in namespace. We'll introduce another built-in function, `dir`, which, given a module, returns a list of the names defined in it:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__',
 '__import__', '__name__', '__package__', 'abs', 'all', 'any', 'ascii',
 'bin', 'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'cmp',
 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help',
 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
 'vars', 'zip']
```


There are a lot of entries here. Those ending in `Error` and `System Exit` are the names of the exceptions built into Python. These will be discussed in chapter 14, “Exceptions.”

The last group (from `abs` to `zip`) are built-in functions of Python. You’ve already seen many of these in this book and will see more. But they won’t all be covered here. If you’re interested, you can find details on the rest in the *Python Library Reference*. You can also at any time easily obtain the documentation string for any of them, either by using the `help()` function or by printing the docstring directly:

```
>>> print(max.__doc__)
max(iterable[, key=func]) -> value
max(a, b, c, ...[, key=func]) -> value
```

```
With a single iterable argument, return its largest item.
With two or more arguments, return the largest argument.
>>>
```

As mentioned earlier, it’s not unheard of for a new Python programmer to inadvertently override a built-in function:

```
>>> list("Peyto Lake")
['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']
>>> list = [1, 3, 5, 7]
>>> list("Peyto Lake")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: 'list' object is not callable
```

The Python interpreter won’t look beyond the new binding for `list` as a `list`, even though we’re using the built-in `list` function syntax.

The same thing will, of course, happen if we try to use the same identifier twice in a single namespace. The previous value will be overwritten, regardless of its type:

```
>>> import mymath
>>> mymath = mymath.area
>>> mymath.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute 'pi'
```

When you’re aware of this, it isn’t a significant issue. Reusing identifiers, even for different types of objects, wouldn’t make for the most readable code anyway. If you do inadvertently make one of these mistakes when in interactive mode, it’s easy to recover. You can use `del` to remove your binding, to regain access to an overridden built-in, or to import your module again to regain access:

```
>>> del list
>>> list("Peyto Lake")
['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']
>>> import mymath
>>> mymath.pi
3.1415899999999999
```

The `locals` and `globals` functions can be useful as simple debugging tools. The `dir` function doesn't give the current settings; but if you call it without parameters, it returns a sorted list of the identifiers in the local namespace. This helps catch the mistyped variable error that compilers may usually catch for you in languages that require declarations:

```
>>> x1 = 6
>>> x1 = x1 - 2
>>> x1
6
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'x1', 'x1']
```

The debugger that's bundled with IDLE has settings where you can view the local and global variable settings as you step through your code; it displays the output of the `locals` and `globals` functions.

10.8 Summary

Python uses modules to manage Python code and objects but allows you to put related code and objects into a file. Not only does this make managing and reusing larger amounts of code easier, but it also helps avoid conflicting variable names, because each object imported from a module is normally named in association with its module.

Being able to package related functions into modules is the final piece of knowledge you need to write standalone programs and scripts in Python, and that's what we'll discuss in the next chapter.