CHAPTER 13

while and for Loops

This chapter concludes our tour of Python procedural statements by presenting the language's two main *looping* constructs—statements that repeat an action over and over. The first of these, the `while` statement, provides a way to code general loops. The second, the `for` statement, is designed for stepping through the items in a sequence or other iterable object and running a block of code for each.

We've seen both of these informally already, but we'll fill in additional usage details here. While we're at it, we'll also study a few less prominent statements used within loops, such as `break` and `continue`, and cover some built-ins commonly used with loops, such as `range`, `zip`, and `map`.

Although the `while` and `for` statements covered here are the primary syntax provided for coding repeated actions, there are additional looping operations and concepts in Python. Because of that, the iteration story is continued in the next chapter, where we'll explore the related ideas of Python's *iteration protocol* (used by the `for` loop) and *list comprehensions* (a close cousin to the `for` loop). Later chapters explore even more exotic iteration tools such as *generators*, `filter`, and `reduce`. For now, though, let's keep things simple.

while Loops

Python's `while` statement is the most general iteration construct in the language. In simple terms, it repeatedly executes a block of (normally indented) statements as long as a test at the top keeps evaluating to a true value. It is called a "loop" because control keeps looping back to the start of the statement until the test becomes false. When the test becomes false, control passes to the statement that follows the `while` block. The net effect is that the loop's body is executed repeatedly while the test at the top is true. If the test is false to begin with, the body never runs and the `while` statement is skipped.

General Format

In its most complex form, the `while` statement consists of a header line with a test expression, a body of one or more normally indented statements, and an optional `else` part that is executed if control exits the loop without a `break` statement being encountered. Python keeps evaluating the test at the top and executing the statements nested in the loop body until the test returns a false value:

```
while test:                          # Loop test
    statements                       # Loop body
else:                                # Optional else
    statements                       # Run if didn't exit loop with break
```

Examples

To illustrate, let's look at a few simple `while` loops in action. The first, which consists of a `print` statement nested in a `while` loop, just prints a message forever. Recall that `True` is just a custom version of the integer `1` and always stands for a Boolean true value; because the test is always true, Python keeps executing the body forever, or until you stop its execution. This sort of behavior is usually called an *infinite loop*—it's not really immortal, but you may need a Ctrl-C key combination to forcibly terminate one:

```
>>> while True:
...    print('Type Ctrl-C to stop me!')
```

The next example keeps slicing off the first character of a string until the string is empty and hence false. It's typical to test an object directly like this instead of using the more verbose equivalent (`while x != '':`). Later in this chapter, we'll see other ways to step through the items in a string more easily with a `for` loop.

```
>>> x = 'spam'
>>> while x:                        # While x is not empty
...     print(x, end=' ')           # In 2.X use print x,
...     x = x[1:]                   # Strip first character off x
...
spam pam am m
```

Note the `end=' '` keyword argument used here to place all outputs on the same line separated by a space; see Chapter 11 if you've forgotten why this works as it does. This may leave your input prompt in an odd state at the end of your output; type Enter to reset. Python 2.X readers: also remember to use a trailing comma instead of `end` in the `print`s like this.

The following code counts from the value of `a` up to, but not including, `b`. We'll also see an easier way to do this with a Python `for` loop and the built-in `range` function later:

```
>>> a=0; b=10
>>> while a < b:                    # One way to code counter loops
...     print(a, end=' ')
...     a += 1                      # Or, a = a + 1
```

```
...
0 1 2 3 4 5 6 7 8 9
```

Finally, notice that Python doesn't have what some languages call a "do until" loop statement. However, we can simulate one with a test and `break` at the bottom of the loop body, so that the loop's body is always run at least once:

```
while True:
    ...loop body...
    if exitTest(): break
```

To fully understand how this structure works, we need to move on to the next section and learn more about the `break` statement.

break, continue, pass, and the Loop else

Now that we've seen a few Python loops in action, it's time to take a look at two simple statements that have a purpose only when nested inside loops—the `break` and `continue` statements. While we're looking at oddballs, we will also study the loop `else` clause here because it is intertwined with `break`, and Python's empty placeholder statement, `pass` (which is not tied to loops per se, but falls into the general category of simple one-word statements). In Python:

`break`
: Jumps out of the closest enclosing loop (past the entire loop statement)

`continue`
: Jumps to the top of the closest enclosing loop (to the loop's header line)

`pass`
: Does nothing at all: it's an empty statement placeholder

`Loop else block`
: Runs if and only if the loop is exited normally (i.e., without hitting a `break`)

General Loop Format

Factoring in `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    statements
    if test: break                 # Exit loop now, skip else if present
    if test: continue              # Go to top of loop now, to test1
else:
    statements                     # Run if we didn't hit a 'break'
```

`break` and `continue` statements can appear anywhere inside the `while` (or `for`) loop's body, but they are usually coded further nested in an `if` test to take action in response to some condition.

Let's turn to a few simple examples to see how these statements come together in practice.

pass

Simple things first: the `pass` statement is a no-operation placeholder that is used when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a `pass`:

```
while True: pass                    # Type Ctrl-C to stop me!
```

Because the body is just an empty statement, Python gets stuck in this loop. `pass` is roughly to statements as `None` is to objects—an explicit nothing. Notice that here the `while` loop's body is on the same line as the header, after the colon; as with `if` statements, this only works if the body isn't a compound statement.

This example does nothing forever. It probably isn't the most useful Python program ever written (unless you want to warm up your laptop computer on a cold winter's day!); frankly, though, I couldn't think of a better `pass` example at this point in the book.

We'll see other places where `pass` makes more sense later—for instance, to ignore exceptions caught by `try` statements, and to define empty `class` objects with attributes that behave like "structs" and "records" in other languages. A `pass` is also sometime coded to mean "to be filled in later," to stub out the bodies of functions temporarily:

```
def func1():
    pass                            # Add real code here later

def func2():
    pass
```

We can't leave the body empty without getting a syntax error, so we say `pass` instead.

> *Version skew note*: Python 3.X (but not 2.X) allows *ellipses* coded as `...` (literally, three consecutive dots) to appear any place an expression can. Because ellipses do nothing by themselves, this can serve as an alternative to the `pass` statement, especially for code to be filled in later—a sort of Python "TBD":
>
> ```
> def func1():
>     ...                     # Alternative to pass
>
> def func2():
>     ...
>
> func1()                     # Does nothing if called
> ```
>
> Ellipses can also appear on the same line as a statement header and may be used to initialize variable names if no specific type is required:
>
> ```
> def func1(): ...            # Works on same line too
> def func2(): ...
> ```

```
>>> X = ...                    # Alternative to None
>>> X
Ellipsis
```

This notation is new in Python 3.X—and goes well beyond the original intent of `...` in slicing extensions—so time will tell if it becomes widespread enough to challenge `pass` and `None` in these roles.

continue

The `continue` statement causes an immediate jump to the top of a loop. It also sometimes lets you avoid statement nesting. The next example uses `continue` to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Remember, 0 means false and `%` is the remainder of division (modulus) operator, so this loop counts down to 0, skipping numbers that aren't multiples of 2—it prints `8 6 4 2 0`:

```
x = 10
while x:
    x = x-1                    # Or, x -= 1
    if x % 2 != 0: continue    # Odd? -- skip print
    print(x, end=' ')
```

Because `continue` jumps to the top of the loop, you don't need to nest the `print` statement here inside an `if` test; the `print` is only reached if the `continue` is not run. If this sounds similar to a "go to" in other languages, it should. Python has no "go to" statement, but because `continue` lets you jump about in a program, many of the warnings about readability and maintainability you may have heard about "go to" apply. `continue` should probably be used sparingly, especially when you're first getting started with Python. For instance, the last example might be clearer if the `print` were nested under the `if`:

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:             # Even? -- print
        print(x, end=' ')
```

Later in this book, we'll also learn that raised and caught exceptions can also emulate "go to" statements in limited and structured ways; stay tuned for more on this technique in Chapter 36 where we will learn how to use it to break out of multiple nested loops, a feat not possible with the next section's topic alone.

break

The `break` statement causes an immediate exit from a loop. Because the code that follows it in the loop is not executed if the `break` is reached, you can also sometimes avoid nesting by including a `break`. For example, here is a simple interactive loop (a variant of a larger example we studied in Chapter 10) that inputs data with `input` (known as `raw_input` in Python 2.X) and exits when the user enters “stop” for the name request:

```
>>> while True:
...     name = input('Enter name:')            # Use raw_input() in 2.X
...     if name == 'stop': break
...     age  = input('Enter age: ')
...     print('Hello', name, '=>', int(age) ** 2)
...
Enter name:bob
Enter age: 40
Hello bob => 1600
Enter name:sue
Enter age: 30
Hello sue => 900
Enter name:stop
```

Notice how this code converts the `age` input to an integer with `int` before raising it to the second power; as you’ll recall, this is necessary because `input` returns user input as a string. In Chapter 36, you’ll see that `input` also raises an exception at end-of-file (e.g., if the user types Ctrl-Z on Windows or Ctrl-D on Unix); if this matters, wrap `input` in `try` statements.

Loop else

When combined with the loop `else` clause, the `break` statement can often eliminate the need for the search status flags used in other languages. For instance, the following piece of code determines whether a positive integer `y` is prime by searching for factors greater than 1:

```
x = y // 2                                # For some y > 1
while x > 1:
    if y % x == 0:                        # Remainder
        print(y, 'has factor', x)
        break                             # Skip else
    x -= 1
else:                                     # Normal exit
    print(y, 'is prime')
```

Rather than setting a flag to be tested when the loop is exited, it inserts a `break` where a factor is found. This way, the loop `else` clause can assume that it will be executed only if no factor is found; if you don’t hit the `break`, the number is prime. Trace through this code to see how this works.

The loop `else` clause is also run if the body of the loop is never executed, as you don’t run a `break` in that event either; in a `while` loop, this happens if the test in the header is false to begin with. Thus, in the preceding example you still get the “is prime” message if `x` is initially less than or equal to 1 (for instance, if `y` is 2).

This example determines primes, but only informally so. Numbers less than 2 are not considered prime by the strict mathematical definition. To be really picky, this code also fails for negative numbers and succeeds for floating-point numbers with no decimal digits. Also note that its code must use `//` instead of `/` in Python 3.X because of the migration of `/` to "true division," as described in Chapter 5 (we need the initial division to truncate remainders, not retain them!). If you want to experiment with this code, be sure to see the exercise at the end of Part IV, which wraps it in a function for reuse.

More on the loop else

Because the loop `else` clause is unique to Python, it tends to perplex some newcomers (and go unused by some veterans; I've met some who didn't even know there *was* an `else` on loops!). In general terms, the loop `else` simply provides explicit syntax for a common coding scenario—it is a coding structure that lets us catch the "other" way out of a loop, without setting and checking flags or conditions.

Suppose, for instance, that we are writing a loop to search a list for a value, and we need to know whether the value was found after we exit the loop. We might code such a task this way (this code is intentionally abstract and incomplete; `x` is a sequence and `match` is a tester function to be defined):

```
found = False
while x and not found:
    if match(x[0]):                 # Value at front?
        print('Ni')
        found = True
    else:
        x = x[1:]                   # Slice off front and repeat
if not found:
    print('not found')
```

Here, we initialize, set, and later test a flag to determine whether the search succeeded or not. This is valid Python code, and it does work; however, this is exactly the sort of structure that the loop `else` clause is there to handle. Here's an `else` equivalent:

```
while x:                            # Exit when x empty
    if match(x[0]):
        print('Ni')
        break                       # Exit, go around else
    x = x[1:]
else:
    print('Not found')              # Only here if exhausted x
```

This version is more concise. The flag is gone, and we've replaced the `if` test at the loop end with an `else` (lined up vertically with the word `while`). Because the `break` inside the main part of the `while` exits the loop and goes around the `else`, this serves as a more structured way to catch the search-failure case.

Some readers might have noticed that the prior example's `else` clause could be replaced with a test for an empty `x` after the loop (e.g., `if not x:`). Although that's true in this example, the `else` provides explicit syntax for this coding pattern (it's more obviously a search-failure clause here), and such an explicit empty test may not apply in some cases. The loop `else` becomes even more useful when used in conjunction with the `for` loop—the topic of the next section—because sequence iteration is not under your control.

Why You Will Care: Emulating C while Loops

The section on expression statements in Chapter 11 stated that Python doesn't allow statements such as assignments to appear in places where it expects an expression. That is, each statement must generally appear on a line by itself, not nested in a larger construct. That means this common C language coding pattern won't work in Python:

```
while ((x = next(obj)) != NULL) {...process x...}
```

C assignments return the value assigned, but Python assignments are just statements, not expressions. This eliminates a notorious class of C errors: you can't accidentally type `=` in Python when you mean `==`. If you need similar behavior, though, there are at least three ways to get the same effect in Python `while` loops without embedding assignments in loop tests. You can move the assignment into the loop body with a `break`:

```
while True:
    x = next(obj)
    if not x: break
    ...process x...
```

or move the assignment into the loop with tests:

```
x = True
while x:
    x = next(obj)
    if x:
        ...process x...
```

or move the first assignment outside the loop:

```
x = next(obj)
while x:
    ...process x...
    x = next(obj)
```

Of these three coding patterns, the first may be considered by some to be the least structured, but it also seems to be the simplest and is the most commonly used. A simple Python `for` loop may replace such C loops as well and be more Pythonic, but C doesn't have a directly analogous tool:

```
for x in obj: ...process x...
```

for Loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The for statement works on strings, lists, tuples, and other built-in iterables, as well as new user-defined objects that we'll learn how to create later with classes. We met for briefly in Chapter 4 and in conjunction with sequence object types; let's expand on its usage more formally here.

General Format

The Python for loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through. The header is followed by a block of (normally indented) statements that you want to repeat:

```
for target in object:                  # Assign object items to target
    statements                         # Repeated loop body: use target
else:                                  # Optional else part
    statements                         # If we didn't hit a 'break'
```

When Python runs a for loop, it assigns the items in the iterable object to the target one by one and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

The name used as the assignment target in a for header line is usually a (possibly new) variable in the scope where the for statement is coded. There's not much unique about this name; it can even be changed inside the loop's body, but it will automatically be set to the next item in the sequence when control returns to the top of the loop again. After the loop this variable normally still refers to the last item visited, which is the last item in the sequence unless the loop exits with a break statement.

The for statement also supports an optional else block, which works exactly as it does in a while loop—it's executed if the loop exits without running into a break statement (i.e., if all items in the sequence have been visited). The break and continue statements introduced earlier also work the same in a for loop as they do in a while. The for loop's complete format can be described this way:

```
for target in object:                  # Assign object items to target
    statements
    if test: break                     # Exit loop now, skip else
    if test: continue                  # Go to top of loop now
else:
    statements                         # If we didn't hit a 'break'
```

Examples

Let's type a few for loops interactively now, so you can see how they are used in practice.

Basic usage

As mentioned earlier, a `for` loop can step across any kind of sequence object. In our first example, for instance, we'll assign the name `x` to each of the three items in a list in turn, from left to right, and the `print` statement will be executed for each. Inside the `print` statement (the loop body), the name `x` refers to the current item in the list:

```
>>> for x in ["spam", "eggs", "ham"]:
...     print(x, end=' ')
...
spam eggs ham
```

The next two examples compute the sum and product of all the items in a list. Later in this chapter and later in the book we'll meet tools that apply operations such as `+` and `*` to items in a list automatically, but it's often just as easy to use a `for`:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24
```

Other data types

Any sequence works in a `for`, as it's a generic tool. For example, `for` loops work on strings and tuples:

```
>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")

>>> for x in S: print(x, end=' ')        # Iterate over a string
...
l u m b e r j a c k

>>> for x in T: print(x, end=' ')        # Iterate over a tuple
...
and I'm okay
```

In fact, as we'll learn in the next chapter when we explore the notion of "iterables," `for` loops can even work on some objects that are not sequences—files and dictionaries work, too.

Tuple assignment in for loops

If you're iterating through a sequence of tuples, the loop target itself can actually be a *tuple* of targets. This is just another case of the tuple-unpacking assignment we studied in Chapter 11 at work. Remember, the `for` loop assigns items in the sequence object to the target, and assignment works the same everywhere:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                    # Tuple assignment at work
...     print(a, b)
...
1 2
3 4
5 6
```

Here, the first time through the loop is like writing `(a,b) = (1,2)`, the second time is like writing `(a,b) = (3,4)`, and so on. The net effect is to automatically unpack the current tuple on each iteration.

This form is commonly used in conjunction with the `zip` call we'll meet later in this chapter to implement parallel traversals. It also makes regular appearances in conjunction with SQL databases in Python, where query result tables are returned as sequences of sequences like the list used here—the outer list is the database table, the nested tuples are the rows within the table, and tuple assignment extracts columns.

Tuples in `for` loops also come in handy to iterate through *both* keys and values in dictionaries using the `items` method, rather than looping through the keys and indexing to fetch the values manually:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...    print(key, '=>', D[key])            # Use dict keys iterator and index
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...    print(key, '=>', value)             # Iterate over both keys and values
...
a => 1
c => 3
b => 2
```

It's important to note that tuple assignment in `for` loops isn't a special case; any assignment target works syntactically after the word `for`. We can always assign manually within the loop to unpack:

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both                        # Manual assignment equivalent
...     print(a, b)                        # 2.X: prints with enclosing tuple "()"
...
```

```
1 2
3 4
5 6
```

But tuples in the loop header save us an extra step when iterating through sequences of sequences. As suggested in Chapter 11, even *nested* structures may be automatically unpacked this way in a `for`:

```
>>> ((a, b), c) = ((1, 2), 3)               # Nested sequences work too
>>> a, b, c
(1, 2, 3)

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

Even this is not a special case, though—the `for` loop simply runs the sort of assignment we ran just before it, on each iteration. Any nested sequence structure may be unpacked this way, simply because *sequence assignment* is so generic:

```
>>> for ((a, b), c) in [([1, 2], 3), ['XY', 6]]: print(a, b, c)
...
1 2 3
X Y 6
```

Python 3.X extended sequence assignment in for loops

In fact, because the loop variable in a `for` loop can be any assignment target, we can also use Python 3.X's extended sequence-unpacking assignment syntax here to extract items and sections of sequences within sequences. Really, this isn't a special case either, but simply a new assignment form in 3.X, as discussed in Chapter 11; because it works in assignment statements, it automatically works in `for` loops.

Consider the tuple assignment form introduced in the prior section. A tuple of values is assigned to a tuple of names on each iteration, exactly like a simple assignment statement:

```
>>> a, b, c = (1, 2, 3)                        # Tuple assignment
>>> a, b, c
(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:       # Used in for loop
...     print(a, b, c)
...
1 2 3
4 5 6
```

In Python 3.X, because a sequence can be assigned to a more general set of names with a starred name to collect multiple items, we can use the same syntax to extract parts of nested sequences in the `for` loop:

```
>>> a, *b, c = (1, 2, 3, 4)                    # Extended seq assignment
>>> a, b, c
```

```
(1, [2, 3], 4)

>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

In practice, this approach might be used to pick out multiple columns from rows of data represented as nested sequences. In Python 2.X starred names aren't allowed, but you can achieve similar effects by slicing. The only difference is that slicing returns a type-specific result, whereas starred names always are assigned lists:

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:          # Manual slicing in 2.X
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
...
1 (2, 3) 4
5 (6, 7) 8
```

See Chapter 11 for more on this assignment form.

Nested for loops

Now let's look at a `for` loop that's a bit more sophisticated than those we've seen so far. The next example illustrates statement nesting and the loop `else` clause in a `for`. Given a list of objects (`items`) and a list of keys (`tests`), this code searches for each key in the objects list and reports on the search's outcome:

```
>>> items = ["aaa", 111, (4, 5), 2.01]       # A set of objects
>>> tests = [(4, 5), 3.14]                   # Keys to search for
>>>
>>> for key in tests:                        # For all keys
...     for item in items:                   # For all items
...         if item == key:                  # Check for match
...             print(key, "was found")
...             break
...     else:
...         print(key, "not found!")
...
(4, 5) was found
3.14 not found!
```

Because the nested `if` runs a `break` when a match is found, the loop `else` clause can assume that if it is reached, the search has failed. Notice the nesting here. When this code runs, there are two loops going at the same time: the outer loop scans the keys list, and the inner loop scans the items list for each key. The nesting of the loop `else` clause is critical; it's indented to the same level as the header line of the inner `for` loop, so it's associated with the inner loop, not the `if` or the outer `for`.

This example is illustrative, but it may be easier to code if we employ the `in` operator to test membership. Because `in` implicitly scans an object looking for a match (at least logically), it replaces the inner loop:

```
>>> for key in tests:                    # For all keys
...     if key in items:                 # Let Python check for a match
...         print(key, "was found")
...     else:
...         print(key, "not found!")
...
(4, 5) was found
3.14 not found!
```

In general, it's a good idea to let Python do as much of the work as possible (as in this solution) for the sake of brevity and performance.

The next example is similar, but builds a list as it goes for later use instead of printing. It performs a typical data-structure task with a `for`—collecting common items in two sequences (strings)—and serves as a rough set intersection routine. After the loop runs, `res` refers to a list that contains all the items found in `seq1` and `seq2`:

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = []                             # Start empty
>>> for x in seq1:                       # Scan first sequence
...     if x in seq2:                    # Common item?
...         res.append(x)                # Add to result end
...
>>> res
['s', 'a', 'm']
```

Unfortunately, this code is equipped to work only on two specific variables: `seq1` and `seq2`. It would be nice if this loop could somehow be generalized into a tool you could use more than once. As you'll see, that simple idea leads us to *functions*, the topic of the next part of the book.

This code also exhibits the classic *list comprehension* pattern—collecting a results list with an iteration and optional filter test—and could be coded more concisely too:

```
>>> [x for x in seq1 if x in seq2]       # Let Python collect results
['s', 'a', 'm']
```

But you'll have to read on to the next chapter for the rest of this story.

Why You Will Care: File Scanners

In general, loops come in handy anywhere you need to repeat an operation or process something more than once. Because *files* contain multiple characters and lines, they are one of the more typical use cases for loops. To load a file's contents into a string all at once, you simply call the file object's `read` method:

```
file = open('test.txt', 'r')      # Read contents into a string
print(file.read())
```

But to load a file in smaller pieces, it's common to code either a `while` loop with breaks on end-of-file, or a `for` loop. To read by *characters*, either of the following codings will suffice:

```
file = open('test.txt')
while True:
    char = file.read(1)          # Read by character
    if not char: break           # Empty string means end-of-file
    print(char)

for char in open('test.txt').read():
    print(char)
```

The `for` loop here also processes each character, but it loads the file into memory all at once (and assumes it fits!). To read by *lines* or *blocks* instead, you can use `while` loop code like this:

```
file = open('test.txt')
while True:
    line = file.readline()       # Read line by line
    if not line: break
    print(line.rstrip())         # Line already has a \n

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)        # Read byte chunks: up to 10 bytes
    if not chunk: break
    print(chunk)
```

You typically read binary data in blocks. To read text files *line by line*, though, the `for` loop tends to be easiest to code and the quickest to run:

```
for line in open('test.txt').readlines():
    print(line.rstrip())

for line in open('test.txt'):    # Use iterators: best for text input
    print(line.rstrip())
```

Both of these versions work in both Python 2.X and 3.X. The first uses the file `readlines` method to load a file all at once into a line-string list, and the last example here relies on file *iterators* to automatically read one line on each loop iteration.

The last example is also generally the *best* option for text files—besides its simplicity, it works for arbitrarily large files because it doesn't load the entire file into memory all at once. The iterator version may also be the quickest, though I/O performance may vary per Python line and release.

File `readlines` calls can still be useful, though—to *reverse* a file's lines, for example, assuming its content can fit in memory. The `reversed` built-in accepts a sequence, but not an arbitrary iterable that generates values; in other words, a list works, but a file object doesn't:

```
for line in reversed(open('test.txt').readlines()): ...
```

In some 2.X Python code, you may also see the name `open` replaced with `file` and the file object's older `xreadlines` method used to achieve the same effect as the file's automatic line iterator (it's like `readlines` but doesn't load the file into memory all at once). Both `file` and `xreadlines` are removed in Python 3.X, because they are redundant. You should generally avoid them in new 2.X code too—use file iterators and `open` call in recent 2.X releases—but they may pop up in older code and resources.

See the library manual for more on the calls used here, and Chapter 14 for more on file line iterators. Also watch for the sidebar "Why You Will Care: Shell Commands and More" on page 411 in this chapter; it applies these same file tools to the `os.popen` command-line launcher to read program output. There's more on reading files in Chapter 37 too; as we'll see there, text and binary files have slightly different semantics in 3.X.

Loop Coding Techniques

The `for` loop we just studied subsumes most counter-style loops. It's generally simpler to code and often quicker to run than a `while`, so it's the first tool you should reach for whenever you need to step through a sequence or other iterable. In fact, as a general rule, you should *resist the temptation to count things in Python*—its iteration tools automate much of the work you do to loop over collections in lower-level languages like C.

Still, there are situations where you will need to iterate in more specialized ways. For example, what if you need to visit every second or third item in a list, or change the list along the way? How about traversing more than one sequence in parallel, in the same `for` loop? What if you need indexes too?

You can always code such unique iterations with a `while` loop and manual indexing, but Python provides a set of built-ins that allow you to specialize the iteration in a `for`:

- The built-in `range` function (available since Python 0.X) produces a series of successively higher integers, which can be used as indexes in a `for`.
- The built-in `zip` function (available since Python 2.0) returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a `for`.
- The built-in `enumerate` function (available since Python 2.3) generates both the values and indexes of items in an iterable, so we don't need to count manually.
- The built-in `map` function (available since Python 1.0) can have a similar effect to `zip` in Python 2.X, though this role is removed in 3.X.

Because `for` loops may run quicker than `while`-based counter loops, though, it's to your advantage to use tools like these that allow you to use `for` whenever possible. Let's look at each of these built-ins in turn, in the context of common use cases. As we'll see, their usage may differ slightly between 2.X and 3.X, and some of their applications are more valid than others.

Counter Loops: range

Our first loop-related function, `range`, is really a general tool that can be used in a variety of contexts. We met it briefly in Chapter 4. Although it's used most often to generate indexes in a `for`, you can use it anywhere you need a series of integers. In

Python 2.X range creates a physical *list*; in 3.X, range is an *iterable* that generates items on demand, so we need to wrap it in a list call to display its results all at once in 3.X only:

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

With one argument, range generates a list of integers from zero up to but not including the argument's value. If you pass in two arguments, the first is taken as the lower bound. An optional third argument can give a *step*; if it is used, Python adds the step to each successive integer in the result (the step defaults to +1). Ranges can also be nonpositive and nonascending, if you want them to be:

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

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

We'll get more formal about iterables like this one in Chapter 14. There, we'll also see that Python 2.X has a cousin named xrange, which is like its range but doesn't build the result list in memory all at once. This is a space optimization, which is subsumed in 3.X by the generator behavior of its range.

Although such range results may be useful all by themselves, they tend to come in most handy within for loops. For one thing, they provide a simple way to repeat an action a specific number of times. To print three lines, for example, use a range to generate the appropriate number of integers:

```
>>> for i in range(3):
...     print(i, 'Pythons')
...
0 Pythons
1 Pythons
2 Pythons
```

Note that for loops force results from range automatically in 3.X, so we don't need to use a list wrapper here in 3.X (in 2.X we get a temporary list unless we call xrange instead).

Sequence Scans: while and range Versus for

The range call is also sometimes used to iterate over a sequence indirectly, though it's often not the best approach in this role. The easiest and generally fastest way to step through a sequence exhaustively is always with a simple for, as Python handles most of the details for you:

```
>>> X = 'spam'
>>> for item in X: print(item, end=' ')          # Simple iteration
...
s p a m
```

Internally, the `for` loop handles the details of the iteration automatically when used this way. If you really need to take over the indexing logic explicitly, you can do it with a `while` loop:

```
>>> i = 0
>>> while i < len(X):                           # while loop iteration
...     print(X[i], end=' ')
...     i += 1
...
s p a m
```

You can also do manual indexing with a `for`, though, if you use `range` to generate a list of indexes to iterate through. It's a multistep process, but it's sufficient to generate offsets, rather than the items at those offsets:

```
>>> X
'spam'
>>> len(X)                                      # Length of string
4
>>> list(range(len(X)))                         # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ') # Manual range/len iteration
...
s p a m
```

Note that because this example is stepping over a list of *offsets* into `X`, not the actual *items* of `X`, we need to index back into `X` within the loop to fetch each item. If this seems like overkill, though, it's because it is: there's really no reason to work this hard in this example.

Although the `range`/`len` combination suffices in this role, it's probably not the best option. It may run slower, and it's also more work than we need to do. Unless you have a special indexing requirement, you're better off using the simple `for` loop form in Python:

```
>>> for item in X: print(item, end=' ')          # Use simple iteration if you can
```

As a general rule, use `for` instead of `while` whenever possible, and don't use `range` calls in `for` loops except as a last resort. This simpler solution is almost always better. Like every good rule, though, there are plenty of exceptions—as the next section demonstrates.

Sequence Shufflers: range and len

Though not ideal for simple sequence scans, the coding pattern used in the prior example does allow us to do more specialized sorts of traversals when required. For example, some algorithms can make use of sequence reordering—to generate alternatives in searches, to test the effect of different value orderings, and so on. Such cases may require offsets in order to pull sequences apart and put them back together, as in the following; the range's integers provide a repeat count in the first, and a position for slicing in the second:

```
>>> S = 'spam'
>>> for i in range(len(S)):          # For repeat counts 0..3
...     S = S[1:] + S[:1]            # Move front item to end
...     print(S, end=' ')
...
pams amsp mspa spam

>>> S
'spam'
>>> for i in range(len(S)):          # For positions 0..3
...     X = S[i:] + S[:i]            # Rear part + front part
...     print(X, end=' ')
...
spam pams amsp mspa
```

Trace through these one iteration at a time if they seem confusing. The second creates the same results as the first, though in a different order, and doesn't change the original variable as it goes. Because both slice to obtain parts to concatenate, they also work on any type of sequence, and return sequences of the same type as that being shuffled—if you shuffle a list, you create reordered lists:

```
>>> L = [1, 2, 3]
>>> for i in range(len(L)):
...     X = L[i:] + L[:i]            # Works on any sequence type
...     print(X, end=' ')
...
[1, 2, 3] [2, 3, 1] [3, 1, 2]
```

We'll make use of code like this to test functions with different argument orderings in Chapter 18, and will extend it to functions, generators, and more complete permutations in Chapter 20—it's a widely useful tool.

Nonexhaustive Traversals: range Versus Slices

Cases like that of the prior section are valid applications for the `range`/`len` combination. We might also use this technique to skip items as we go:

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

Here, we visit every *second* item in the string `S` by stepping over the generated `range` list. To visit every third item, change the third `range` argument to be `3`, and so on. In effect, using `range` this way lets you skip items in loops while still retaining the simplicity of the `for` loop construct.

In most cases, though, this is also probably not the "best practice" technique in Python today. If you really mean to skip items in a sequence, the extended three-limit form of the *slice expression*, presented in Chapter 7, provides a simpler route to the same goal. To visit every second character in `S`, for example, slice with a stride of 2:

```
>>> S = 'abcdefghijk'
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

The result is the same, but substantially easier for you to write and for others to read. The potential advantage to using `range` here instead is space: slicing makes a copy of the string in both 2.X and 3.X, while `range` in 3.X and `xrange` in 2.X do not create a list; for very large strings, they may save memory.

Changing Lists: range Versus Comprehensions

Another common place where you may use the `range`/`len` combination with `for` is in loops that change a list as it is being traversed. Suppose, for example, that you need to add 1 to every item in a list (maybe you're giving everyone a raise in an employee database list). You can try this with a simple `for` loop, but the result probably won't be exactly what you want:

```
>>> L = [1, 2, 3, 4, 5]

>>> for x in L:
...     x += 1                          # Changes x, not L
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

This doesn't quite work—it changes the loop variable `x`, not the list `L`. The reason is somewhat subtle. Each time through the loop, `x` refers to the next integer already pulled out of the list. In the first iteration, for example, `x` is integer `1`. In the next iteration, the loop body sets `x` to a different object, integer `2`, but it does not update the list where `1` originally came from; it's a piece of memory separate from the list.

To really change the list as we march across it, we need to use indexes so we can assign an updated value to each position as we go. The `range`/`len` combination can produce the required indexes for us:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):             # Add one to each item in L
...     L[i] += 1                       # Or L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

When coded this way, the list is changed as we proceed through the loop. There is no way to do the same with a simple `for x in L:`–style loop, because such a loop iterates through actual items, not list positions. But what about the equivalent `while` loop? Such a loop requires a bit more work on our part, and might run more slowly depending on your Python (it does on 2.7 and 3.3, though less so on 3.3—we'll see how to verify this in Chapter 21):

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

Here again, though, the `range` solution may not be ideal either. A list comprehension expression of the form:

```
[x + 1 for x in L]
```

likely runs faster today and would do similar work, albeit without changing the original list in place (we could assign the expression's new list object result back to `L`, but this would not update any other references to the original list). Because this is such a central looping concept, we'll save a complete exploration of list comprehensions for the next chapter, and continue this story there.

Parallel Traversals: zip and map

Our next loop coding technique extends a loop's scope. As we've seen, the `range` built-in allows us to traverse sequences with `for` in a nonexhaustive fashion. In the same spirit, the built-in `zip` function allows us to use `for` loops to visit multiple sequences *in parallel*—not overlapping in time, but during the same loop. In basic operation, `zip` takes one or more sequences as arguments and returns a series of tuples that pair up parallel items taken from those sequences. For example, suppose we're working with two lists (a list of names and addresses paired by position, perhaps):

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

To combine the items in these lists, we can use `zip` to create a list of tuple pairs. Like `range`, `zip` is a list in Python 2.X, but an iterable object in 3.X where we must wrap it in a `list` call to display all its results at once (again, there's more on iterables coming up in the next chapter):

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                    # list() required in 3.X, not 2.X
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Such a result may be useful in other contexts as well, but when wedded with the `for` loop, it supports parallel iterations:

```
>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Here, we step over the result of the `zip` call—that is, the pairs of items pulled from the two lists. Notice that this `for` loop again uses the tuple assignment form we met earlier to unpack each tuple in the `zip` result. The first time through, it's as though we ran the assignment statement `(x, y) = (1, 5)`.

The net effect is that we scan both `L1` *and* `L2` in our loop. We could achieve a similar effect with a `while` loop that handles indexing manually, but it would require more typing and would likely run more slowly than the `for`/`zip` approach.

Strictly speaking, the `zip` function is more general than this example suggests. For instance, it accepts any type of sequence (really, any iterable object, including files), and it accepts more than two arguments. With three arguments, as in the following example, it builds a list of three-item tuples with items from each sequence, essentially projecting by columns (technically, we get an N-ary tuple for N arguments):

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))                    # Three tuples for three arguments
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Moreover, `zip` truncates result tuples at the length of the shortest sequence when the argument lengths differ. In the following, we zip together two strings to pick out characters in parallel, but the result has only as many tuples as the length of the shortest sequence:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))                        # Truncates at len(shortest)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

map equivalence in Python 2.X

In Python 2.X only, the related built-in `map` function pairs items from sequences in a similar fashion when passed `None` for its function argument, but it pads shorter sequences with `None` if the argument lengths differ instead of truncating to the shortest length:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'

>>> map(None, S1, S2)                        # 2.X only: pads to len(longest)
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None,'3')]
```

This example is using a degenerate form of the map built-in, which is no longer supported in 3.X. Normally, map takes a function and one or more sequence arguments and collects the results of calling the function with parallel items taken from the sequence(s).

We'll study map in detail in Chapter 19 and Chapter 20, but as a brief example, the following maps the built-in ord function across each item in a string and collects the results (like zip, map is a value generator in 3.X and so must be passed to list to collect all its results at once in 3.X only):

```
>>> list(map(ord, 'spam'))
[115, 112, 97, 109]
```

This works the same as the following loop statement, but map is often quicker, as Chapter 21 will show:

```
>>> res = []
>>> for c in 'spam': res.append(ord(c))
>>> res
[115, 112, 97, 109]
```

Version skew note: The degenerate form of map using a function argument of None is no longer supported in Python 3.X, because it largely overlaps with zip (and was, frankly, a bit at odds with map's function-application purpose). In 3.X, either use zip or write loop code to pad results yourself. In fact, we'll see how to write such loop code in Chapter 20, after we've had a chance to study some additional iteration concepts.

Dictionary construction with zip

Let's look at another zip use case. Chapter 8 suggested that the zip call used here can also be handy for generating dictionaries when the sets of keys and values must be computed at runtime. Now that we're becoming proficient with zip, let's explore more fully how it relates to dictionary construction. As you've learned, you can always create a dictionary by coding a dictionary literal, or by assigning to keys over time:

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'eggs': 3, 'toast': 5, 'spam': 1}

>>> D1 = {}
>>> D1['spam']  = 1
>>> D1['eggs']  = 3
>>> D1['toast'] = 5
```

What to do, though, if your program obtains dictionary keys and values in *lists* at runtime, after you've coded your script? For example, say you had the following keys and values lists, collected from a user, parsed from a file, or obtained from another dynamic source:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

One solution for turning those lists into a dictionary would be to zip the lists and step through them in parallel with a for loop:

```
>>> list(zip(keys, vals))
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'eggs': 3, 'toast': 5, 'spam': 1}
```

It turns out, though, that in Python 2.2 and later you can skip the for loop altogether and simply pass the zipped keys/values lists to the built-in dict constructor call:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

>>> D3 = dict(zip(keys, vals))
>>> D3
{'eggs': 3, 'toast': 5, 'spam': 1}
```

The built-in name dict is really a type name in Python (you'll learn more about type names, and subclassing them, in Chapter 32). Calling it achieves something like a list-to-dictionary conversion, but it's really an object construction request.

In the next chapter we'll explore the related but richer concept, the list comprehension, which builds lists in a single expression; we'll also revisit Python 3.X and 2.7 dictionary comprehensions, an alternative to the dict call for zipped key/value pairs:

```
>>> {k: v for (k, v) in zip(keys, vals)}
{'eggs': 3, 'toast': 5, 'spam': 1}
```

Generating Both Offsets and Items: enumerate

Our final loop helper function is designed to support dual usage modes. Earlier, we discussed using range to generate the offsets of items in a string, rather than the items at those offsets. In some programs, though, we need both: the item to use, plus an offset as we go. Traditionally, this was coded with a simple for loop that also kept a counter of the current offset:

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print(item, 'appears at offset', offset)
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

This works, but in all recent Python 2.X and 3.X releases (since 2.3) a new built-in named `enumerate` does the job for us—its net effect is to give loops a counter "for free," without sacrificing the simplicity of automatic iteration:

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print(item, 'appears at offset', offset)
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

The `enumerate` function returns a *generator object*—a kind of object that supports the iteration protocol that we will study in the next chapter and will discuss in more detail in the next part of the book. In short, it has a method called by the `next` built-in function, which returns an `(index, value)` tuple each time through the loop. The `for` steps through these tuples automatically, which allows us to unpack their values with tuple assignment, much as we did for `zip`:

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x0000000002A8B900>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

We don't normally see this machinery because all iteration contexts—including list comprehensions, the subject of Chapter 14—run the iteration protocol automatically:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']

>>> for (i, l) in enumerate(open('test.txt')):
...     print('%s) %s' % (i, l.rstrip()))
...
0) aaaaaa
1) bbbbbb
2) cccccc
```

To fully understand iteration concepts like `enumerate`, `zip`, and list comprehensions, though, we need to move on to the next chapter for a more formal dissection.

Why You Will Care: Shell Commands and More

An earlier sidebar showed loops applied to files. As briefly noted in Chapter 9, Python's related `os.popen` call also gives a file-like interface, for reading the outputs of spawned *shell commands*. Now that we've studied looping statements in full, here's an example of this tool in action—to run a shell command and read its standard output text, pass the command as a string to `os popen`, and read text from the file-like object it returns (if this triggers a Unicode encoding issue on your computer, Chapter 25's discussion of currency symbols may apply):

```
>>> import os
>>> F = os.popen('dir')              # Read line by line
>>> F.readline()
' Volume in drive C has no label.\n'
>>> F = os.popen('dir')              # Read by sized blocks
>>> F.read(50)
' Volume in drive C has no label.\n Volume Serial Nu'

>>> os.popen('dir').readlines()[0]   # Read all lines: index
' Volume in drive C has no label.\n'
>>> os.popen('dir').read()[:50]      # Read all at once: slice
' Volume in drive C has no label.\n Volume Serial Nu'

>>> for line in os.popen('dir'):     # File line iterator loop
...     print(line.rstrip())
...
 Volume in drive C has no label.
 Volume Serial Number is D093-D1F7
...and so on...
```

This runs a `dir` directory listing on Windows, but any program that can be started with a command line can be launched this way. We might use this scheme, for example, to display the output of the windows `systeminfo` command—`os.system` simply runs a shell command, but `os.popen` also connects to its streams; both of the following show the shell command's output in a simple console window, but the first might not in a GUI interface such as IDLE:

```
>>> os.system('systeminfo')
...output in console, popup in IDLE...
0
>>> for line in os.popen('systeminfo'): print(line.rstrip())

Host Name:                 MARK-VAIO
OS Name:                   Microsoft Windows 7 Professional
OS Version:                6.1.7601 Service Pack 1 Build 7601
...lots of system information text...
```

And once we have a command's output in text form, any string processing tool or technique applies—including display formatting and content parsing:

```
# Formatted, limited display
>>> for (i, line) in enumerate(os.popen('systeminfo')):
...     if i == 4: break
...     print('%05d) %s' % (i, line.rstrip()))
...
00000)
00001) Host Name:                 MARK-VAIO
00002) OS Name:                   Microsoft Windows 7 Professional
00003) OS Version:                6.1.7601 Service Pack 1 Build 7601

# Parse for specific lines, case neutral
>>> for line in os.popen('systeminfo'):
...    parts = line.split(':')
...    if parts and parts[0].lower() == 'system type':
...        print(parts[1].strip())
```

```
...
x64-based PC
```

We'll see `os.popen` in action again in Chapter 21, where we'll deploy it to read the results of a constructed command line that times code alternatives, and in Chapter 25, where it will be used to compare outputs of scripts being tested.

Tools like `os.popen` and `os.system` (and the `subprocess` module not shown here) allow you to leverage every command-line program on your computer, but you can also write emulators with in-process code. For example, simulating the Unix `awk` utility's ability to strip columns out of text files is almost trivial in Python, and can become a reusable function in the process:

```
# awk emulation: extract column 7 from whitespace-delimited file
for val in [line.split()[6] for line in open('input.txt')]:
    print(val)

# Same, but more explicit code that retains result
col7 = []
for line in open('input.txt'):
    cols = line.split()
    col7.append(cols[6])
for item in col7:  print(item)

# Same, but a reusable function (see next part of book)
def awker(file, col):
    return [line.rstrip().split()[col-1] for line in open(file)]

print(awker('input.txt', 7))              # List of strings
print(','.join(awker('input.txt', 7)))    # Put commas between
```

By itself, though, Python provides file-like access to a wide variety of data—including the text returned by *websites* and their pages identified by URL, though we'll have to defer to Part V for more on the package import used here, and other resources for more on such tools in general (e.g., this works in 2.X, but uses `urllib` instead of `urlib.request`, and returns text strings):

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://home.rmi.net/~lutz'):
...     print(line)
...
b'<HTML>\n'
b'\n'
b'<HEAD>\n'
b"<TITLE>Mark Lutz's Book Support Site</TITLE>\n"
...etc...
```

Chapter Summary

In this chapter, we explored Python's looping statements as well as some concepts related to looping in Python. We looked at the `while` and `for` loop statements in depth, and we learned about their associated `else` clauses. We also studied the `break` and `continue` statements, which have meaning only inside loops, and met several built-in tools commonly used in `for` loops, including `range`, `zip`, `map`, and `enumerate`, although some of the details regarding their roles as iterables in Python 3.X were intentionally cut short.

In the next chapter, we continue the iteration story by discussing list comprehensions and the iteration protocol in Python—concepts strongly related to `for` loops. There, we'll also give the rest of the picture behind the iterable tools we met here, such as `range` and `zip`, and study some of the subtleties of their operation. As always, though, before moving on let's exercise what you've picked up here with a quiz.

Test Your Knowledge: Quiz

1. What are the main functional differences between a `while` and a `for`?
2. What's the difference between `break` and `continue`?
3. When is a loop's `else` clause executed?
4. How can you code a counter-based loop in Python?
5. What can a `range` be used for in a `for` loop?

Test Your Knowledge: Answers

1. The `while` loop is a general looping statement, but the `for` is designed to iterate across items in a sequence or other iterable. Although the `while` can imitate the `for` with counter loops, it takes more code and might run slower.
2. The `break` statement exits a loop immediately (you wind up below the entire `while` or `for` loop statement), and `continue` jumps back to the top of the loop (you wind up positioned just before the test in `while` or the next item fetch in `for`).
3. The `else` clause in a `while` or `for` loop will be run once as the loop is exiting, if the loop exits normally (without running into a `break` statement). A `break` exits the loop immediately, skipping the `else` part on the way out (if there is one).
4. Counter loops can be coded with a `while` statement that keeps track of the index manually, or with a `for` loop that uses the `range` built-in function to generate successive integer offsets. Neither is the preferred way to work in Python, if you need to simply step across all the items in a sequence. Instead, use a simple `for` loop instead, without `range` or counters, whenever possible; it will be easier to code and usually quicker to run.
5. The `range` built-in can be used in a `for` to implement a fixed number of repetitions, to scan by offsets instead of items at offsets, to skip successive items as you go, and to change a list while stepping across it. None of these roles requires `range`, and most have alternatives—scanning actual items, three-limit slices, and list comprehensions are often better solutions today (despite the natural inclinations of ex–C programmers to want to count things!).