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:

`while` / `else`

: The most general looping statement, which can handle repetitive tasks of all kinds

`for` / `else`

: A specialized loop designed for stepping through the items in any "iterable" object easily

We've met and used 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`, the loop `else`, and cover some built-ins commonly used with loops, such as `range`, `zip`, and `enumerate`.

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* and functional tools like `map`, `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 an associated block of 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 does become 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                  #     Repeated loop body
else:                           # Optional else
    statements                  #     Run if didn't exit l
```

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 it:

```
>>> 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 (and begins omitting the REPL's `...` prompts for easier emedia copy and paste where possible). It's typical to test an object directly like this instead of using the more verbose equivalent

(`while x != '':`), though later in this chapter you'll see other ways to step through the items in a string more easily with a `for` loop:

```
>>> x = 'code'
>>> while x:                    # While x is not empty
        print(x, end=' ')       # Print next character
        x = x[1:]               # Strip first character c

code ode de e
```

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 will probably leave your REPL's input prompt at the end of the output's line; type Enter (or your keyboard's or app's equivalent) to reset if desired.

The following code *counts* from the value of `a` up to, but not including, `b`. Loops like this are often used to generate object indexes. You'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
        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…                 # Always run loop body at
    if test: break              # Test for loop exit at t
```

To fully understand how this structure works, we need to move on to the next section's coverage of `break`.

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`, as well as Python's empty placeholder statement `pass`, which is not tied to loops but falls into the 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 e
    if test: continue           # Go to test at top of
else:
    statements                  # Run on exit if didn't
```

`break` and `continue` statements can appear anywhere inside the `while` (or per ahead, `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 coded 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 stc
```

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 works only if the body isn't a compound statement (and doesn't contain one).

This example does nothing forever. It probably isn't the most useful Python program ever written (unless you want to warm up your laptop or phone on a cold winter's day), but it's tough to come up with a better `pass` example at this point in the book; it's not a commonly used tool.

You'll see other places where `pass` makes a bit 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. Though partly a preview, 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.

The ellipsis-literal alternative

Despite the limited roles, there's a similar, if more obscure, way to achieve the same effect as `pass`. Python allows an *ellipsis*, coded as `...` (literally, three consecutive dots, not the Unicode character), to appear any place an expression can. Because ellipses do nothing by themselves, they 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

func1()                     # Does nothing if called
```

This works because any expression can appear as a statement (as we learned in Chapter 11), and the `...` literal qualifies as an expression. Ellipses can also appear on a statement header by itself, and may be used to initialize variable names if no specific type is required—which also makes it an alternative to `None`:

```
def func1(): ...            # Works on same line too

>>> tbd = ...               # Alternative to both pass ar
>>> tbd                     # Ellipsis is a real (if oddl
Ellipsis
```

This goes well beyond the original intent of `...` in slicing extensions (which, like the `@` operator and type hinting, is unused by Python itself), so time will tell if it rises to challenge `pass` and `None` in these inane and vacuous roles.

continue

The `continue` statement causes an immediate jump to the top of a loop. It's often used to avoid statement nesting, as in the next example that uses `continue` to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Recall that 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—and prints `8 6 4 2 0`:

```
x = 10
while x:
    x -= 1                          # Or, x = 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.

The nested-code alternative

If all 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 coding. For instance, the last example might be clearer if the `print` were nested under the `if`:

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

Later in this book, you'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 you 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—technically, from the closest enclosing loop, when loops are nested. Because the code that follows it in the loop is not executed if the `break` is reached, it can sometimes avoid nesting much like `continue`. For example, here is a

simple interactive loop (a takeoff on code we studied in Chapter 10) that inputs data with `input` and exits when the user enters a "stop" line:

```
>>> num = 1
>>> while True:
        tool = input(f'{num}) What\'s your favorite lar
        if tool == 'stop': break
        print('Bravo!' if tool == 'Python' else 'Try ag
        num += 1

1) What's your favorite language? Java
Try again...
2) What's your favorite language? Python
Bravo!
3) What's your favorite language? stop
```

Because the `break` in this terminates the `while` immediately, there's no reason to nest code below it in an `else`.

The named-assignment alternative

That said, it's also possible to use the newer `:=` expression we met in "Named Assignment Expressions" to crunch this example—albeit, at the expense of its role as a `break` demo. While you should judge for yourself, the net effect is concise but may at least flirt with unreadability:

```
>>> num = 1
>>> while (tool := input(f'{num}) What\'s your favorite
        print('Bravo!' if tool == 'Python' else 'Try ag
        num += 1

1) What's your favorite language? Python
Bravo!
```

Nesting `:=` within `:=` as in the following, however, could easily incite pitchforks and torches (in fact, the full one-liner here is too wide for this book!). Unless you can defend this in a court of your code-reuse peers, just say no:

```
num = 0
while (tool := input(f'{(num := num + 1)}) What\'s your
```

Preview: in Chapter 36, you'll see that `input` also raises an exception at end-of-file (e.g., if the user enters Ctrl+Z on Windows or Ctrl+D on Unix); wrapping `input` in `try` statements allows users to respond this way too.

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. In abstract terms the `break` in the following skips the `else` on the way out of the loop:

```
while continuing:
    if found:
        found code
        break
    else advance
else:
    not-found code
```

As a more concrete example, the following piece of code determines whether a positive integer `num` is prime—has no factors other than 1 and itself—by searching for factors greater than 1 (to run live, assign `num` before pasting):

```
x = num // 2                                # For some nu
while x > 1:
    if num % x == 0:                        # Remainder 0
        print(num, 'has factor', x)
        break                               # Exit now ar
    x -= 1
else:                                       # Normal exit
    print(num, '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 this code never hits 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 `num` is 2).

NOTE

Subprime code: This example determines primes, but only informally so. Numbers less than 2 are not considered prime by the strict mathematical definition, but 1 and 0 are classified as such here. To be really picky, this code also fails for negative numbers and succeeds for floating-point numbers with all-zero decimal digits. Also note that its code must use `//` instead of `/` because we need the initial division to truncate remainders, not retain them. If you want to experiment with this code further, watch for its associated exercise at the end of Part IV, which wraps it in a function for reuse.

Why the loop else?

Because the loop `else` clause is unique to Python, it tends to perplex some newcomers (and even some veterans; in fact, a few either pointlessly code the loop `else` without a `break` or don't know that the loop `else` exists at all!). 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 need to know whether the value was found after you 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('Found')
        found = True
    else:
        x = x[1:]                   # Slice off front
if not found:
    print('Not found')
```

Here, we initialize, set, and later test a `found` 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 meant to handle. Here's an `else` equivalent:

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

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.

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. All told, the `for` statement works on strings, lists, tuples, sets, dictionaries, and all other built-in iterables, as well as new user-defined objects that you'll learn how to create later with classes. We met `for` briefly in Chapter 4 and have used it 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 t
    statements                      #     Repeated loop boc
else:                               # Optional else
    statements                      #     Run if didn't exi
```

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.

While *`target`* can be any assignment target we met in Chapter 11, it's often just a simple name. This name is a possibly new variable that lives in the scope where the `for` statement itself 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 *`object`* 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 early 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` . Given all that, the `for` loop's complete format can be described this way:

```
for target in object:               # Assign object items t
    statements
    if test: break                  # Exit loop now, skip e
    if test: continue               # Go to top of loop now
else:
    statements                      # Run on exit if didn't
```

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 ['app', 'script', 'program']:
        print(x, end=' ')

app script program
```

The next two examples compute the sum and product of all the items in a list. Later in this chapter and later in this book you'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 also work on strings and tuples:

```
>>> S = 'Python'
>>> T = ('web', 'num', 'app')

>>> for x in S: print(x, end=' ')      # Iterate over c

P y t h o n

>>> for x in T: print(x, end=' ')      # Iterate over c

web num app
```

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

Tuple (sequence) 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 assignme
        print(a, b)

1 2
3 4
5 6
```

Here, the first time through the loop is like running `(a,b) = (1,2)`, the second time is like `(a,b) = (3,4)`, and so on. The net effect is to automatically *unpack* the current tuple on each iteration. List syntax works as a `for` target too because tuple and list assignment are both *sequence* assignment, and tuple parentheses are optional:

```
>>> for [a, b] in T:                   # List assignmer
```

```
>>> for a, b in T:                    # Tuple sans par
```

This list-of-tuples data format is commonly used in conjunction with the `zip` call you'll meet later in this chapter, to implement parallel traversals. It also crops up 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 (i.e., sequence) assignment extracts columns.

As we've seen in earlier chapters, 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}
>>> for key in D:
        print(key, '=>', D[key])        # Use dict keys

a => 1
b => 2

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

>>> for (key, value) in D.items():
        print(key, '=>', value)         # Iterate over b

a => 1
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` . For example, 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 assignm
        print(a, b)
```

```
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 sequenc
>>> a, b, c
(1, 2, 3)

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

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(

1 2 3
X Y 6
```

Extended-unpacking assignment in for loops

In fact, because the loop variable in a `for` loop can be *any* assignment target, we can also use the starred names and other targets of extended-unpacking assignment here to extract both items and sections of sequences within sequences. Because this works in assignment statements, it automatically works in `for` loops too.

This topic was introduced in Chapter 11, but here's a quick refresher to reinforce the technique. 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)                #
>>> 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
```

Because sequence assignment supports a more general set of names with a starred target 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)                #
>>> 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. As usual in Python, you can achieve similar effects with more basic tools—in this case by slicing. The only difference is that slicing returns a type-specific result, whereas starred targets always receive lists:

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

1 (2, 3) 4
5 (6, 7) 8
```

Finally, all the starred-target forms work in `for` as in `=` assignment statements, including nested sequences, indexes, and slices (though practical

roles for code like the following are probably much more rare than common!):

```
>>> L, M = [1, 2], [3, 4]
>>> pairs = [[(5, 6), (7, 8), (9, 10)]] * 2

>>> for [(a, *X), (b, *L[0]), (c, *M[:0])] in pairs:
        print(f'<{a=} {X=}>  <{b=} {L=}>  <{c=} {M=}>')

<a=5 X=[6]>  <b=7 L=[[8], 2]>  <c=9 M=[10, 3, 4]>
<a=5 X=[6]>  <b=7 L=[[8], 2]>  <c=9 M=[10, 10, 3, 4]>
```

See Chapter 11 for more on the extended-unpacking form of assignment.

Nested for loops

Now let's look at some `for` loops that are a bit more sophisticated than those demoed so far. The first shows what happens when `for` loops are nested—the inner loop is run for every iteration of the outer loop, and the `+` within the inner loop combines their items by using each loop's variable:

```
>>> for x in 'abc':                          # For each
        for y in '123':                       # And for e
            print(x + y, end=' ')             # Concatenc

a1 a2 a3 b1 b2 b3 c1 c2 c3
```

The next example kicks the nesting up a notch, illustrating both three-level 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 list of
>>> tests = [(4, 5), 3.14]                  # Keys to s
>>>
>>> for key in tests:                        # For all k
        for item in items:                   # For all i
            if item == key:                  # Check for
                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 inner loop's `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` .

The preceding 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 k
        if key in items:                        # Let Pytho
            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 both brevity and performance.

Our final 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 (it's nearly *intersection*, unless there are duplicate values). After the loop runs, `res` refers to a list that contains all the items found in `seq1` and `seq2` :

```
>>> seq1 = 'trippy'
>>> seq2 = 'python'
>>>
>>> res = []                                   # Start emp
>>> for x in seq1:                             # Scan firs
        if x in seq2:                           # Common it
```

```
            res.append(x)                    # Add to re
```

```
>>> res
['t', 'p', 'p', 'y']
```

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.

Of course, if you read Chapter 4 or Chapter 5, you know that Python has sets that provide true intersection with the `&` operator—but the result's order is scrambled, duplicates are dropped, and multiple conversions are required to match:

```
>>> list(set(seq1) & set(seq2))             # Real inte
['p', 'y', 't']
```

More usefully, this code also exhibits the classic *list comprehension* pattern—collecting a results list with an iteration and optional filter test—and could be coded much more concisely with this tool:

```
>>> [x for x in seq1 if x in seq2]          # Let Pytho
['t', 'p', 'p', 'y']
```

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

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 produces a series of successively higher integers, which can be used as indexes in a `for`.
- The built-in `zip` function returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a `for`.
- The built-in `enumerate` function generates both the values and indexes of items in an iterable, so we don't need to count manually.

Because `for` loops may run quicker than `while`-based counter loops, 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 roles. As you'll see, some loop-coding alternatives are more valid than others.

Counter Loops: range

Our first loop-related function, `range`, is a general tool that can be used in a variety of contexts. We met it briefly in Chapter 4 and have used it occasionally along the way. Although it's used often to generate indexes in a loop, you can call it anywhere you need a series of integers (see Chapter 11's enumerated-names trick for a prime example).

As we've seen, `range` is an *iterable* that generates items on demand, so we need to wrap it in a `list` call to display all its results at once in a REPL. Surprisingly, `range`'s results support *some* sequence operations, but not all; per Chapter 9, it was reclassified in Python's docs as a sort of sequence object type, though one with less functionality than lists and tuples—and much less basis in reality:

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

>>> range(5)[2], range(5)[1:3], list(range(5)) + [6, 7]
(2, range(1, 3), [0, 1, 2, 3, 4, 6, 7])
```

```
>>> range(5) + [6, 7]
TypeError: unsupported operand type(s) for +: 'range' a
```

Categorization aside, `range` usage is straightforward, With one argument, `range` generates a series 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. And 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 need 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 take a deeper look at iterables in Chapter 14. In this case, Python 2.X had an optimized built-in named `xrange` , which was like its `range` but didn't build a result list in memory all at once; which was later superseded in 3.X by the generator behavior of its `range` ; which was later rebranded a sequence by 3.X docs (confusingly!). The upshot of this long walk is that today's `range` doesn't consume much space, because it produces numbers only on demand.

Although the preceding `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, so we don't need to use a `list` wrapper here. In fact, we *shouldn't*: letting `range` produce its results one at a time uses much less memory than forcing them all at once.

Sequence Scans: while, range, and 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 fastest way to step through a sequence exhaustively is almost always with a simple `for`, because Python handles most of the details for you in quick, internal code:

```
>>> X = 'hack'
>>> for item in X: print(item, end=' ')            # Aut

h a c k
```

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 (and sometimes you may), you can do it with a `while` loop:

```
>>> i = 0
>>> while i < len(X):                             # Mar
        print(X[i], end=' ')
        i += 1

h a c k
>>> i = -1
>>> while (i := i + 1) < len(X):                  # Mar
        print(X[i], end=' ')

h a c k
```

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

```
>>> X
'hack'
>>> len(X)                                        # Ler
4
>>> list(range(len(X)))                           # All
[0, 1, 2, 3]
>>>
```

```
>>> for i in range(len(X)): print(X[i], end=' ')  # Mar

h a c k
```

Importantly, 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 is useful in some roles, it's probably not the best option in most. It may run slower, and it's also more code than we need to write. 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
```

As guidelines, 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 guideline, though, there are plenty of exceptions—as the next section demonstrates.

Sequence Shufflers: range and len

Though not ideal for simple sequence scans, the `range` / `len` 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; its `range`'s integers provide a repeat count in the first, and a position for slicing in the second:

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

ackh ckha khac hack

>>> S
```

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

hack ackh ckha khac
```

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, 4]
>>> for i in range(len(L)):
        X = L[i:] + L[:i]          # Works on any sequer
        print(X, end=' ')

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

The results of `range` itself, however, don't make the grade in either coding (they're not true sequences!):

```
>>> L = range(4)
>>> …same code as prior example…
TypeError: unsupported operand type(s) for +: 'range' a
```

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.

Skipping Items: range and Slices

The prior section showed one 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` statement.

In many or 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, while `range` does not—and hence may save significant memory for very large strings. Naturally, whether your program needs to care depends on what it does.

Changing Lists: range and 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 updating ages at year end). You can try this with a simple `for` loop, but the result may not be what you want or expect:

```
>>> L = [10, 20, 30, 40, 50]

>>> for x in L:
        x += 1                          # Changes x, not L

>>> L                                   # L's objects unch
```

```
[10, 20, 30, 40, 50]
>>> x                                    # x is not a cursc
51
```

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 `10`, taken from `L`, When we then add to `x` in the loop body with `+=`, it sets `x` to a different object, integer `11`, but it does not update the list where `10` originally came from; the new `11` is 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 = [10, 20, 30, 40, 50]

>>> for i in range(len(L)):              # Add one to each
        L[i] += 1                        # Or L[i] = L[i] +

>>> L
[11, 21, 31, 41, 51]
```

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`, because such a loop iterates through actual *items*, not their 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, your host device, and perhaps the alignment of planets (you'll see how to check such claims in Chapter 21):

```
>>> i = 0
>>> while i < len(L):                   # And similar with
        L[i] += 1
        i += 1

>>> L
[12, 22, 32, 42, 52]
```

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

```
>>> [x + 1 for x in L]
[13, 23, 33, 43, 53]
```

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 tell the rest of the statements story of loops there.

Parallel Traversals: zip

Our next loop coding technique adds to its bag of tricks. As we've seen, the `range` built-in allows us to traverse sequences with `for` in a nonexhaustive fashion. In a similar 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 arguments (sequences or other iterables) and returns a series of tuples that pair up parallel items taken from those arguments. For example, suppose we're working with two lists of data paired by position:

```
>>> 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 an *iterable* object, so we must wrap it in a `list` call to collect and display all its results at once (again, the next chapter will be more formal about iterables like this):

```
>>> zip(L1, L2)                                # An iterab
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                          # list() re
[(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(f'{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 (a.k.a. sequence) 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)` ; and so on.

The net effect is that we scan both `L1` *and* `L2` in our loop. To be sure, we could achieve a similar effect with a `while` loop that handles indexing manually—like the following that produces the same output as the preceding:

```
>>> i = -1
>>> while (i := i + 1) < len(L1):
       print(f'{L1[i]} + {L2[i]} => {L1[i] + L2[i]}')
```

But this requires noticeably more code, and hence would likely run slower than the `for` / `zip` approach. Moreover, it's no better on space: being an iterable, `zip` makes just one pair per loop, and so does not consume memory needlessly. The clincher, though, is that this is not really equivalent to `zip` — for reasons disclosed in the next section.

More on zip: size and truncation

For the record, the `zip` function is more general than the prior example suggests. For instance, it both *is* an iterable and *accepts* any type of iterable object, including `range` results, input files, and more:

```
>>> list(zip(range(4), 'hack'))
[(0, 'h'), (1, 'a'), (2, 'c'), (3, 'k')]
```

In addition, `zip` is not just for two-item pairs: it accepts any number of *arguments*, of any *size*. The following, for example, builds a list of three-item tuples for three arguments, with items from each sequence—essentially projecting by columns:

```
>>> T1, T2, T3 = (1, 2, 3), (4, 5, 6), (7, 8, 9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))                  # 3 args of
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

And formally speaking, for *N* arguments that contain *M* items, `zip` gives us an *M*-long series of *N*-ary tuples:

```
>>> list(zip(T1, T2))                      # 2 args of
[(1, 4), (2, 5), (3, 6)]
```

When argument lengths differ, `zip` *truncates* the series of result tuples at the length of the shortest sequence. To demo, the following zips two strings to pick out characters in parallel, but the result has only as many tuples as the length of the shortest sequence (formally again, *M* in the prior definition is really the minimum of arguments' lengths):

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

To pad instead of truncating, you can write loop code to pad results yourself—as we will in Chapter 20, after we've had a chance to study some additional iteration concepts that make it a fair fight.

More zip roles: dictionaries

Fine points aside, parallel traversals with `zip` are also useful in *dictionary* construction. We met this technique in Chapter 8, but here's a quick refresher in the context of looping statements. As we learned earlier, you can always

create a dictionary by calling `dict`, assigning to keys over time, or coding a dictionary literal like the following:

```
>>> D1 = {'app': 1, 'script': 3, 'program':5}
>>> D1
{'app': 1, 'script': 3, 'program': 5}
```

What to do, though, if your program obtains dictionary keys and values at runtime, after you've coded your script? For example, the following may be collected from a user, a file, or any other dynamic source:

```
>>> keys = ['app', 'script', 'program']
>>> vals = [1, 3, 5]
```

One way to turn these into a dictionary is to `zip` the lists and step through them in parallel with a `for` loop:

```
>>> list(zip(keys, vals))
[('app', 1), ('script', 3), ('program', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v

>>> D2
{'app': 1, 'script': 3, 'program': 5}
```

As suggested earlier in this book, though, you can skip the `for` loop altogether in this context, and simply pass the zipped keys/values lists to the built-in `dict` constructor call:

```
>>> D3 = dict(zip(keys, vals))
>>> D3
{'app': 1, 'script': 3, 'program': 5}
```

The built-in name `dict` is really a *type* name (you'll learn about type names, and subclassing them, in Chapter 32). Calls to it are object construction requests, but also perform a to-dictionary conversion here. In the next chapter, you'll also learn more about related but richer concepts—list comprehensions, which build lists in expressions, and their dictionary comprehensions kin,

which are an alternative to both `for` statements and `dict` for zipped key/value pairs:

```
>>> {k: v for (k, v) in zip(keys, vals)}
{'app': 1, 'script': 3, 'program': 5}
```

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. This might be coded with a `for` loop that also keeps a counter of the current offset:

```
>>> S = 'hack'
>>> offset = 0
>>> for item in S:
        print(item, 'appears at offset', offset)
        offset += 1

h appears at offset 0
a appears at offset 1
c appears at offset 2
k appears at offset 3
```

This works, but Python has a built-in function named `enumerate` that does the job for us—its net effect is to give loops a counter "for free," without sacrificing the simplicity of automatic iteration:

```
>>> S = 'hack'
>>> for (offset, item) in enumerate(S):
        print(item, 'appears at offset', offset)

h appears at offset 0
a appears at offset 1
c appears at offset 2
k appears at offset 3
```

As for `range` and `zip`, the `enumerate` function's result is an *iterable*—a kind of object that supports the iteration protocol that we will dive into in the

next chapter. 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 0x10ebd7880>
>>> next(E)
(0, 'h')
>>> next(E)
(1, 'a')
>>> next(E)
(2, 'c')
```

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

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'a', 'cc', 'kkk']

>>> for (ix, line) in enumerate(open('data.txt')):
        print(f'{ix}) {line.rstrip()}')

0) Testing file IO
1) Learning Python, 6E
2) Python 3.12
```

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

Chapter Summary

In this chapter, we explored Python's looping statements and their related tools. 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-ins commonly used in `for` loops, including `range`, `zip`, and

`enumerate`, although some of the details regarding their roles as iterables 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 fill in 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 the knowledge you've picked up here with a quiz.

Test Your Knowledge: Quiz

1. What are the main functional differences between `while` and `for` loops?
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 automatically 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 (control flow winds up below the entire `while` or `for` loop statement), and `continue` jumps back to the top of the loop (control flow winds 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 and only if the loop exits normally (i.e., by a false test in `while` or an empty object in `for`), 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 code in Python, if you need to simply step across all the items in a

sequence. Instead, use a simple `for` loop 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` loop 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!).

WHY YOU WILL CARE: FILE SCANNERS

Loops come in handy anywhere you need to repeat an operation or process something more than once. Because *text files* contain multiple characters and lines, they are a typical role for loops. Assuming a file's contents can fit in memory, you can load it all at once with the file object's `read` method of Chapter 9:

```
file = open('data.txt')          # Read contents into c
print(file.read())
```

For more granular access, you can scan by *characters* instead with either of the following—the second of which doesn't load the whole file and has grown terser with the `:=` named assignment of Chapter 11:

```
for char in open('data.txt').read():
    print(char, end='')

file = open('data.txt')
while char := file.read(1):      # Read by character, e
    print(char, end='')          # Don't add a \n after
```

To read by *lines* or *blocks* instead, you can use `while` loops like the following; binary data is often read by blocks using binary file mode `'rb'`, but text should use text mode to avoid splitting character bytes:

```
file = open('data.txt')
while line := file.readline():   # Read line by line
    print(line.rstrip())         # Line already has a \

file = open('data.txt')
while chunk := file.read(10):    # Read block by block:
    print(chunk, end='')         # Keep but don't add r
```

To read text files by *lines*, though, the `for` loop tends to be easiest to code and may be quickest to run:

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

```
for line in open('data.txt'):   # Use iterators: best f
    print(line.rstrip())
```

The first version here uses the file `readlines` method to load a file all at once into a line-string list, but the second example relies on file *iterators* to automatically read one line on each loop iteration.

The second example is also generally best 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 speed can vary per Python release (we'll study ways to time code later in this book).

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 works on sequences, but does not accept iterables that generate values; `sorted`, by contrast, does, so it can order all the lines in a file without loading it in full:

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

for line in sorted(open('data.txt')):
    print(line.rstrip())
```

See Python's documentation for more on the file-object calls used here—as well as its coverage of tools like `os.popen` that returns a file object connected to a *shell command*'s output (by default), and hence supports the same sort of loops. There's also an `os.popen` example in Chapter 21, and more on the distinctions of text and binary files in Chapter 37 when we dive into Unicode more deeply.
