Chapter 10. Introducing Python Statements

Now that you're familiar with Python's built-in objects, this chapter begins our exploration of its fundamental statement forms. Much like the previous part's approach, we'll begin here with a general introduction to statement syntax and follow up with more details about specific statements in the next few chapters.

In simple terms, *statements* are the code we write to tell Python what our programs should do. If, as suggested in Chapter 4, programs "do things with stuff," then statements are the way we specify what sort of *things* a program does. Less informally, Python is a procedural, statement-based language; by combining statements, we specify a *procedure* that Python performs to satisfy a program's goals.

The Python Conceptual Hierarchy Revisited

Another way to understand the role of statements is to revisit the concept hierarchy introduced in Chapter 4, which talked about built-in objects and the expressions used to manipulate them. This chapter climbs the hierarchy to the next level of Python program structure:

1. Programs are composed of modules.
2. Modules contain statements.
3. *Statements contain expressions*.
4. Expressions create and process objects.

At their base, programs written in the Python language are composed of statements and expressions. Expressions process objects and are embedded in statements. Statements code the larger *logic* of a program's operation—they use and direct expressions to process the objects we studied in the preceding chapters. Moreover, statements are where objects spring into existence (e.g.,

in expressions within assignment statements), and some statements create entirely new kinds of objects (functions, classes, and so on). At the top, statements always exist in modules, which themselves are managed with statements.

Python's Statements

Table 10-1 summarizes Python's statement set. Each statement in Python has its own specific purpose and its own specific *syntax*—the rules that define its structure—though, as you'll see, many share common syntax patterns, and some statements' roles overlap. Table 10-1 also gives examples of each statement, when coded according to its syntax rules. In your programs, these units of code can perform actions, repeat tasks, make choices, build larger program structures, and so on.

This part of the book deals with entries in the table from the top through `break` and `continue`. You've informally been introduced to a few of the statements in Table 10-1 already; this part of the book will fill in details that were skipped earlier, introduce the rest of Python's procedural statement set, and cover the overall syntax model. Statements lower in Table 10-1 that have to do with larger program units—functions, classes, modules, and exceptions—lead to larger programming ideas, so they will each have a section of their own. More focused statements (like `del`, which deletes various components) are covered elsewhere in this book, as well as in Python's standard manuals.

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'python', 3.12` `a = (b := next()) + more`
Calls and other expressions	Running functions	`log.write('app crash')`
`print`	Printing objects	`print(hack, code, file=log)`
`if/elif/else`	Selecting actions	`if 'python' in text: read(text)`
`match/case`	Multiway selections	`match edition: case 6: print(2024)`
`for/else`	Iteration	`for x in myiterable: print(x)`
`while/else`	General loops	`while x := file.readline(): print(x)`
`pass`	Empty placeholder	`if 'python' not in text: pass`
`break`	Loop exit	`while True: if exittest(): break`
`continue`	Loop continue	`while True: if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=2, *more): print(a + b + c)`
`return`	Functions results	`def f(a, b, c=2, *more): return a + b + c`
`yield`	Generator functions	`def gen(n): for i in n: yield i*2`
`global`	Namespaces	`x = 1 def function(): global x; x = 2`
`nonlocal`	Namespaces	`def outer(): x = 1 def inner(): nonlocal x; x = 2`
`async`	Coroutine designator	`async def consumer(a, b): await producer()`
`await`	Coroutine transfer	`await asyncio.sleep(1)`

Statement	Role	Example
`import`	Module access	`import sys`
`from`	Attribute access	`from sys import stdin as f`
`class`	Building objects	`class Subclass(Superclass): classAttr = [] def method(self): pass`
`try/except/finally`	Catching exceptions, termination actions	`try: action() except: print('action error')`
`raise`	Triggering exceptions	`raise EndSearch(location)`
`assert`	Debugging checks	`assert X > Y, 'X too small'`
`with`	Context managers	`with open('data') as file: process(file)`

`del`	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

`type`	Type hinting alias	`type vector = list[float]`

Technically, Table 10-1 is sufficient as a quick preview and reference, but it's not quite complete as is. Here are a few fine points about its content:

- Assignment statements come in a wide variety of syntax flavors, described in Chapter 11: basic, sequence, augmented, and more; and named assignment (`:=`) is used as an expression, not a statement.
- `print` is really a built-in function call, and neither a reserved word nor a statement. Because it will nearly always be run as an expression statement, though, and usually on a line by itself, it's generally thought of as a statement type, and will be treated separately in Chapter 11.
- `yield` and `await` are also expressions instead of statements. Like `print` , they're often used as expression statements and so are included in this table, but scripts may also assign or otherwise use their result, as you'll see in Chapter 20. As expressions, `yield` and `await` are also reserved words, unlike `print` .

- Most of the words used in statements and expressions are *reserved* and cannot be used as variables in your code; this includes `and`, `in`, `if`, `for`, `while`, and others. Newer statements use "soft" reserved words that are reserved only when used in the statements to which they belong; this includes `match`, `case`, and `type` (though not `async` and `await`). We'll formalize the full list of reserved words in Chapter 11.

A Tale of Two ifs

Before we delve into the details of any of the concrete statements in Table 10-1, this book wishes to begin our look at Python statement syntax by showing you what you are *not* going to type in Python code.

Consider the following `if` statement, coded in a C-like language:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

This might be a statement in C, C++, Java, JavaScript, or similar. Now, look at the equivalent statement in the Python language:

```
if x > y:
    x = 1
    y = 2
```

The first thing that may pop out at you is that the equivalent Python statement is less cluttered—that is, there are fewer syntactic components. This is by design; as a scripting language, one of Python's goals is to make programmers' lives easier by requiring less typing. And less typing also means less room for mistakes.

More specifically, when you compare the two syntax models, you'll notice that Python adds one new thing to the mix but removes three things that are required in C-like languages.

What Python Adds

The one new syntax component in Python is the colon character (`:`). All Python *compound statements*—statements that have other statements nested inside them—follow the same general pattern of a header line terminated in a colon, followed by a nested block of code usually indented underneath the header line, like this:

```
Header line:
    Nested statement block
```

The colon is required, and omitting it is probably the most common coding mistake among new Python programmers (it's certainly one witnessed thousands of times live in Python training classes). In fact, if you are new to Python, you'll almost certainly forget the colon character very soon, if you haven't already. You'll get an error message if you do, and most Python-friendly editors make this mistake easy to spot:

```
>>> if x
    if x
        ^
SyntaxError: expected ':'
```

Including the colon eventually becomes an unconscious habit—so much so that you may start typing colons in your C-like language code, too (generating reams of entertaining error messages from that language's compiler!).

What Python Removes

Although Python requires the extra colon character, there are three things programmers in C-like languages must include that you don't generally have to code in Python.

Parentheses are optional

The first of these is the set of *parentheses* around the tests at the top of some statements:

```
if (x < y)
```

The parentheses here are required by the syntax of many C-like languages. In Python, though, they are not—we simply omit the parentheses, and the statement works the same way:

```
if x < y
```

Technically speaking, because every expression can be enclosed in parentheses, including them will not hurt in this Python code, and they are not treated as an error if present.

But don't do that. You'll be wearing out your keyboard needlessly, and broadcasting to the world that you're a programmer of a C-like language still learning Python (it happens). The "Python way" is to simply omit the parentheses in these kinds of statements altogether.

End-of-line is end of statement

The second and more significant syntax component you won't find in Python code is the *semicolon*. You don't need to terminate statements with semicolons in Python the way you do in C-like languages:

```
x = 1;
```

In Python, the general rule is that the end of a line automatically terminates the statement that appears on that line. In other words, you can leave off the semicolons, and it works the same way:

```
x = 1
```

There are some ways to work around this rule, as you'll see in a moment (for instance, wrapping code in a bracketed structure allows it to span lines). But, in general, you write one statement per line for the vast majority of Python code, and no semicolon is required.

Here, too, if you are pining for your C programming days (and why would you?) you can continue to use semicolons at the end of each statement—the Python language lets you get away with them if they are present, because the semicolon is also a separator when statements are combined.

But don't do that either (really!). Again, doing so tells the world that you're a programmer of a C-like language who still hasn't quite made the switch to Python coding. The Pythonic style is to leave off the semicolons altogether. Judging from students in classes, this seems a tough habit for some veteran programmers to break. But you'll get there; semicolons are useless noise in this role in Python.

End of indentation is end of block

The third and final syntax component that Python removes…and the one that may seem the most unusual to soon-to-be-ex-programmers of C-like languages, until they've used it for 10 minutes and realize it's actually a feature...is that you do not type *anything* explicit in your code to syntactically mark the beginning and end of a nested block of code. You don't need to include `begin` / `end` , `then` / `endif` , or `{` / `}` around the nested block, as you do in C-like languages:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Instead, in Python, we consistently indent all the statements in a given single nested block the same distance to the right, and Python uses the statements' physical indentation to determine where the block starts and stops:

```
if x > y:
    x = 1
    y = 2
```

This *indentation* means the blank whitespace all the way to the left of the two nested statements here. Python doesn't care *how* you indent (you may use either spaces or tabs), or *how much* you indent (you may use any number of spaces or tabs). In fact, the indentation of one nested block can be totally different from that of another. The syntax rule is only that for a given single nested block, all of its statements must be indented the same distance to the right. If this is not the case, you will get a syntax error, and your code will not run until you repair its indentation to be consistent:

```
>>> if True:
...     a = 1
...   b = 2
    b = 2
        ^
IndentationError: unindent does not match any outer inc
```

Why Indentation Syntax?

The indentation rule may seem unusual at first glance to programmers accustomed to C-like languages, but it is an intentional feature of Python: it's one of the main ways that Python almost forces programmers to produce uniform, regular, and readable code. It essentially means that you must line up your code vertically, in *columns*, according to its logical structure. The net effect is to make your code more consistent and readable—unlike much of the code written in C-like languages.

To put that more strongly, aligning your code according to its logical structure is a major part of making it readable, and thus reusable and maintainable, by yourself and others. In fact, even if you never use Python after reading this book, you should get into the habit of aligning your code for readability in *any* block-structured language. Python underscores the issue by making this a part of its syntax, but it's an important thing to do in any programming language, and it has a huge impact on the usefulness of your code.

Your experience may vary, but there's a common phenomenon in large, old C++ programs that have been worked on by many programmers over the years. Almost invariably, each programmer has his or her own style for indenting code. For example, a `while` loop coded in the C++ language may have begun its tenure like this:

```
while (x > 0) {
```

Before we even get into indentation, there are three or four ways that programmers can arrange the `{}` braces in a C-like language, and organizations often suffer political battles and standards docs to address the options (which seems more than a little off-topic for the problem to be solved by programming). Be that as it may, here's the scenario often encountered in

C++ code. The first person who worked on the code indented the loop four spaces:

```
while (x > 0) {
    --------;
    --------;
```

That person eventually moved on to other projects (or, sadly, management), only to be replaced by someone who liked to indent further to the right:

```
while (x > 0) {
    --------;
    --------;
           --------;
           --------;
```

That person later moved on to other opportunities (ending their reign of coding terror), and someone else picked up the code who liked to indent less:

```
while (x > 0) {
    --------;
    --------;
           --------;
           --------;
--------;
--------;
}
```

And so on. Eventually, the block is terminated by a closing brace (}), which of course makes this “block-structured code” (yes, sarcasm). No: in any block-structured language, Python or otherwise, if nested blocks are not indented consistently, they become very difficult for the reader to interpret, change, or reuse, because the code no longer visually reflects its logical meaning. *Readability matters*, and indentation is a major component of readability.

Here is another example that may have burned you in the past if you’ve done much programming in a C-like language. Consider the following statement in C:

```
if (x)
     if (y)
          statement1;
else
     statement2;
```

Which `if` does the `else` here go with? Surprisingly, the `else` is paired with the nested `if` statement (`if (y)`) in C, even though it looks visually as though it is associated with the outer test (`if (x)`). This is a classic pitfall in the C language, and it can lead to the reader completely misinterpreting the code and changing it incorrectly in ways that might not be uncovered until the Mars rover crashes into a giant rock!

This cannot happen in Python—because indentation is significant, the way the code looks is the way it will work. Consider an equivalent Python statement:

```
if x:
     if y:
          statement1
else:
     statement2
```

In this example, the `if` that the `else` lines up with vertically and visually is the one it is associated with logically and behaviorally (the outer `if x`). In a sense, Python is a *WYSIWYG* language—what you see is what you get—because the way code looks is the way it runs, regardless of who coded it.

If this still isn't enough to underscore the benefits of Python's syntax, here's another anecdote. Some *companies* that develop systems software in the C language, where consistent indentation is not required, enforce it anyhow. It's not too uncommon for such groups to run automated tools that analyze the indentation used in code, when it is checked into source control systems at the end of the day. If the tools notice that you've indented your code inconsistently, you might just receive an automated email about it the next morning—and so might your manager!

The point is that even when a language doesn't require it, good programmers know that consistent use of indentation has a huge impact on code readability and quality. The fact that Python promotes this to the level of syntax is seen by most as a feature of the language.

Also keep in mind that nearly every programmer-friendly *text editor* has built-in support for Python's syntax model. In the IDLE GUI, for example, lines of code are automatically indented when you are typing a nested block; pressing the Backspace key backs up one level of indentation, and you can customize how far to the right IDLE indents statements in a nested block. There is no requirement on this: *four spaces* is very common, but it's up to you to decide how and how much you wish to indent (unless your company has endured politics and docs to standardize this too). Indent further to the right for further nested blocks, and less to close the prior block.

As a caution, though, you probably shouldn't *mix* tabs and spaces in the same block in Python, unless you do so consistently; use tabs or spaces in a given block, but not both (in fact, Python issues an error for inconsistent use of tabs and spaces, as you'll see in Chapter 12). Then again, you probably shouldn't mix tabs or spaces in indentation in *any* structured language—such code can cause major readability issues if the next programmer has her or his editor set to display tabs differently than yours. C-like languages might let coders get away with this, but they really shouldn't: the result can be a mangled mess.

Regardless of which language you code in, you should be indenting consistently for readability. In fact, if you weren't taught to do this earlier in your career, your teachers did you a disservice. Most programmers—especially those who must read others' code—consider it an asset that Python elevates this to the level of syntax. Moreover, generating tabs instead of braces is no more difficult in practice for tools that must output Python code, and the page breaks that can obscure code nesting in the print versions of books (including this one!) will not be present in the real world of coding.

In sum, if you do what you should be doing in a C-like language anyhow, but get rid of the braces, your code will satisfy Python's syntax rules.

A Few Special Cases

As mentioned previously, in Python's syntax model:

- The end of a line terminates the statement on that line (without semicolons).
- Nested statements are blocked and associated by their physical indentation (without braces).

Those rules cover almost all Python code you'll write or see in practice. However, Python also provides some special-purpose rules that allow for flexibility in both statements and nested statement blocks. They're not required and should be used sparingly, but programmers have found them useful in practice.

Statement rule special cases

There are three special rules for statements, two of which have already been introduced and used in this book. First of all, although statements normally appear one per line, it is possible to *squeeze* more than one statement onto a single line in Python by separating them with semicolons:

```
a = 1; b = 2; print(a + b)                  # Three statem
```

This is the only place in Python where semicolons are required: as statement separators. This only works, though, if the statements thus combined are not *themselves* compound statements. In other words, you can chain together only simple statements, like assignments, and calls to `print` and other functions and methods. Compound statements like `if` tests and `while` loops must still appear on lines of their own (otherwise, you could squeeze an entire program onto one line, which probably would not make you very popular among your coworkers!).

The other special rule for statements is essentially the inverse: you can make a single statement *span* across multiple lines. To make this work, you simply have to enclose part of your statement in a bracketed pair—parentheses (`()`), square brackets (`[]`), or curly braces (`{}`). Any code enclosed in these constructs can cross multiple lines: your statement doesn't end until Python reaches the line containing the closing part of the pair. For instance, to continue a list literal:

```
mylist = [1111,            # Continuation lines
          2222,            # Any code in (), [], {}
          3333]
```

Because this code is enclosed in a square brackets pair, Python simply keeps reading on the next line until it encounters the closing bracket. The curly braces surrounding dictionaries (as well as set literals and dictionary and set

comprehensions) allow them to span lines this way, too, and parentheses handle tuples, function calls, and expressions. The *indentation* of the continuation lines does not matter, though common sense dictates that the lines should be aligned somehow for readability. Any `#` *comments* are ignored as usual in continuation lines too, and *nested* brackets must all be closed before the continuation-line run ends.

Parentheses are the catchall device—because *any* expression can be wrapped in them, simply inserting a left *parenthesis* allows you to drop down to the next line and continue your statement:

```
X = (A + B +
     C + D)
```

This technique works within *compound* statements, too, by the way. Anywhere you need to code a large expression, simply wrap it in parentheses to continue it on the next line:

```
if (A == 1 and
    B == 2 and
    C == 3):
        print('hack' * 3)
```

An older rule also allows for continuation lines when the prior line ends in a *backslash*:

```
X = A + B + \
      C + D                 # An error-prone older alterr
```

This alternative technique is somewhat discouraged today because it's difficult to notice and maintain the backslashes. It's also fairly *brittle* and *error-prone* —there can be no spaces or # comments after the backslash, and accidentally omitting it can have unexpected effects if the next line is mistaken to be a new statement (in this example, "C + D" is a valid statement by itself if it's not indented and would silently be run as such). This rule is also a throwback to the C language, where it is commonly used in "#define" macros. While `\` may be occasionally useful, when in Pythonland, do as Pythoneers do: use bracketed pairs instead of `\` as a rule.

Block rule special case

As mentioned previously, statements in a nested block of code are normally associated by being indented the same amount to the right. As one special case here, and another we met in earlier chapters, the body of a compound statement can instead appear on the same line as the header in Python, after the colon:

```
if x > y: print(x)
```

This allows us to code single-line `if` statements, single-line `while` and `for` loops, and so on. Much like `;` separators, though, this will work only if the body of the compound statement itself does not *contain* any compound statements. That is, only simple statements—assignments, calls to `print` and others, and the like—are allowed after the colon. Larger statements must still appear on lines by themselves. Extra parts of compound statements (such as the `else` part of an `if`, which you'll meet in the next section) must also be on separate lines of their own. Compound statement bodies can also consist of multiple simple statements separated by semicolons, but this tends to be frowned upon.

In general, even though it's not always required, if you keep most of your statements on individual lines and indent your nested blocks as a norm, your code will be easier to read and change in the future. Moreover, some code profiling and coverage tools may not be able to distinguish between multiple statements squeezed onto a single line, or the header and body of a one-line compound statement. It is almost always to your advantage to keep things simple in Python. You can use the special-case exceptions to write Python code that's hard to read, but it takes a lot of work, and there are probably better ways to spend your time.

To see a prime and common exception to one of these rules in action, however (the use of a single-line `if` statement to `break` out of a loop), and to introduce more of Python's syntax, let's move on to the next section and write some real code.

A Quick Example: Interactive Loops

You'll see all these syntax rules in action when we tour Python's specific compound statements in the next few chapters, but they work the same everywhere in the Python language. To get started, let's work through a brief but realistic example that demos the way that statement syntax and nesting come together and introduces a few statements along the way. To work along, either copy and paste this section's examples from emedia into your REPL or run them from the file *interact.py* located in this book's examples package using any of the launch tools we studied in Chapter 3.

A Simple Interactive Loop

Suppose you're asked to write a Python program that interacts with a user in a console window. Maybe you're accepting inputs to send to a database or reading numbers to be used in a calculation. Regardless of the purpose, you need to code a loop that reads one or more inputs from a user typing on a keyboard and prints back a result for each. In other words, you need to write a classic read/evaluate/print loop program, similar to Python's standard REPL.

In Python, typical boilerplate code for such an interactive loop might look like this:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

This code makes use of a few new ideas and some we've already explored:

- The code leverages the Python `while` loop, Python's most general looping statement. We'll study the `while` statement in more detail later, but in short, it consists of the word `while`, followed by an expression that is interpreted as a true or false result, followed by a nested block of code that is repeated while the test at the top is true. The word `True` here is considered always true, so this loop continues forever, unless somehow stopped.
- The `input` built-in function we met earlier in the book (to keep windows open after clicks in Chapter 3 and the Appendix A coverage it referenced)

is used here for general console input—it prints its optional argument string as a prompt and returns the user's typed reply as a string.

- A single-line `if` statement that makes use of the special rule for nested blocks also appears here: the body of the `if` appears on the header line after the colon instead of being indented on a new line underneath it. This would work either way, but as it's coded, we've saved an extra line.
- Finally, the Python `break` statement is used to exit the loop immediately—it simply jumps out of the loop statement altogether, and the program continues after the loop. Without this exit statement, the `while` would loop forever, as its test is always true.

In effect, this combination of statements essentially means "read a line from the user and print it in uppercase until the user enters the word 'stop.'" There are other ways to code such a loop (e.g., see the note ahead), but the form used here is very common in Python code and serves to illustrate syntax basics.

Notice that all three lines nested under the `while` header line are indented the same amount: because they line up vertically in a column this way, they are the block of code that is associated with the `while` test and repeated. Either a lesser-indented statement or the end of the source file (as here) will suffice to terminate the loop body block.

When this code is run, either interactively or as a script file, here is the sort of interaction we get:

```
Enter text:python
PYTHON
Enter text:312
312
Enter text:stop
```

NOTE

Or crunch code with the := operator: Spoiler alert—this section's code is traditional and simple and works as a syntax demo, but it's possible to reduce it from four lines to two with the `:=` named-assignment expression added in Python 3.8 and covered in the next chapter. This expression assigns a name to another expression's result, but also returns the assigned value as its overall result. The net effect lets us fetch, assign, and test input on the same line—and all in the loop's header:

```
while (reply := input('Enter text:')) != 'stop':
    print(reply.upper())
```

While useful in narrow roles, this expression also requires nested parentheses in this context and is arguably more implicit than traditional forms (though ex-C programmers' mileage may vary!).

Doing Math on User Inputs

Our script works, but now suppose that instead of converting a text string to uppercase, we want to do some math with numeric input—squaring it, for example (perhaps in some misguided effort of an age-input program to tease its users). We might try statements like these to achieve the desired effect:

```
>>> reply = '40'
>>> reply ** 2
TypeError: unsupported operand type(s) for ** or pow():
```

This won't quite work in our script, though: input from a user is always returned as a *string*, and as discussed in the prior part of this book, Python won't convert object types in expressions unless they are all numeric. We cannot raise a string of digits to a power unless we convert it manually to an integer:

```
>>> int(reply) ** 2
1600
```

Armed with this information, we can now recode our loop to perform the necessary math. Type the following in a file to run it with command line, IDLE menu options, or any other technique we met in Chapter 3 (the REPL

both requires a blank line after the `while`, and runs just one statement at a time—the final `print` here wouldn't make sense):

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

This script uses a single-line `if` statement to exit on "stop" as before, but it also converts inputs to perform the required math. This version also adds an exit message at the bottom. Because the `print` statement in the last line is not indented *as much* as the nested block of code, it is not considered part of the loop body and will run only once, after the loop is exited:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

Handling Errors by Testing Inputs

So far so good, but notice what happens when the input is invalid:

```
Enter text:xxx
ValueError: invalid literal for int() with base 10: 'x›
```

The built-in `int` function raises an *exception* (i.e., flags an error) here in the face of a nonnumber. If we want our script to be robust, we can check the string's content ahead of time with the string object's `isdigit` method:

```
>>> S = '40'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

This also gives us an excuse to further nest the statements in our example. The following new version of our interactive script uses a full-blown `if` statement to work around the exception on conversion errors:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

We'll study the `if` statement in more detail in Chapter 12, but it's a fairly lightweight tool for coding logic in scripts. In its full form, it consists of the word `if` followed by a test and an associated block of code; one or more optional `elif` ("else if") tests and code blocks; and an optional `else` part with a block of code at the bottom to serve as a default. Python runs the block of code associated with the first test that is true, working from top to bottom, or the `else` part if all tests are false.

The `if`, `elif`, and `else` parts in the preceding example are associated as part of the same statement because their opening words all line up vertically (i.e., share the same level of indentation). The `if` statement spans from the word `if` to just before the `print` statement on the last line of the script. In turn, the entire `if` block is part of the `while` loop because all of it is indented under the loop's header line. Statement nesting like this is natural once you start using it.

When we run our new script, its code catches errors before they occur and prints an error message before continuing (which you'll probably want to improve before this code is handed over to the quality-assurance team), but "stop" still gets us out, and valid numbers are still squared:

```
Enter text:5
25
Enter text:xxx
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
```

```
Enter text:stop
Bye
```

Handling Errors with try Statements

The preceding solution works, but as you'll see later in the book, the most general way to handle errors in Python is to catch and recover from them completely using the Python `try` statement. We'll explore this statement in depth in Part VII of this book, but as a preview, using a `try` here can lead to code that some might see as simpler than the prior version:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(num ** 2)
print('Bye')
```

This version works exactly like the previous one, but we've replaced the explicit error check with code that assumes the conversion will work and wraps it in an exception handler for cases when it doesn't. In other words, rather than detecting an error, we simply respond if one occurs.

This `try` statement is another compound statement and follows the same pattern as `if` and `while`. It's composed of the word `try`, followed by the main block of code (the action we are trying to run), followed by an `except` part that gives the exception handler code and an `else` part to be run if no exception is raised in the `try` part. Python first runs the `try` part, then runs either the `except` part (if an exception occurs) or the `else` part (if no exception occurs).

In terms of statement nesting, because the words `try`, `except`, and `else` are all indented to the same level, they are all considered part of the same single `try` statement. Notice that the `else` part is associated with the `try` here, not the `if`. As we've seen, `else` can appear in `if` statements in Python, but it can also appear in `try` statements and loops—its indentation

tells you what statement it is a part of. In this case, the `try` statement spans from the word `try` through the code indented under the word `else`, because the `else` is indented the same as `try`. The `if` statement in this code is a one-liner and ends after the `break`, so the `else` cannot apply to it.

Supporting Floating-Point Numbers

Again, we'll come back to the `try` statement later in this book. For now, be aware that because `try` can be used to intercept any error, it reduces the amount of error-checking code you have to write, and it's a very general approach to dealing with unusual cases. If we're sure that `print` won't fail, for instance, this example could be even more concise:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

And if we wanted to support input of floating-point numbers instead of just integers, for example, using `try` would be much easier than manual error testing—we could simply run a `float` call and catch its exceptions:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(float(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

There is no `isfloat` method for strings today, so this exception-based approach spares us from having to accommodate all possible floating-point syntax in an up-front error check (a nontrivial task, given the many faces of floats!). When coded this way, we can enter a wider variety of numbers, but errors and exits still work as before:

```
Enter text:50
2500.0
Enter text:40.5
1640.25
Enter text:1.23E-100
1.5129e-200
Enter text:hack
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

NOTE

Or run anything with eval and exec: Python's built-in `eval` call, which we used in Chapters 5 and 9 to convert data in strings and files, would work in place of `float` here, too, and would allow input of arbitrary expressions ("2 ** 100" would be a legal, if curious, input, especially if we're assuming the program is processing ages!). This is a powerful concept that is open to the same security issues mentioned in the prior chapters. If you can't trust the source of a code string, use more focused and restrictive conversion tools like `int` and `float`.

Python's `exec`, used in Chapter 3 to run code read from a file, is similar to `eval` (but assumes the string is a statement instead of an expression and has no result), and its `compile` call precompiles frequently used code strings to bytecode objects for speed. Run a `help` on any of these for more details. We'll also use `exec` to import modules by name string in Chapter 25—an example of its more dynamic roles.

Nesting Code Three Levels Deep

Let's look at one last mutation of our code. Nesting can take us even further if we need it to—we could, for example, extend our prior integer-only script to branch to one of a set of alternatives based on the relative magnitude of a valid input:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
```

```
            num = int(reply)
            if num < 20:
                print('low')
            else:
                print(num ** 2)
    print('Bye')
```

This version adds an `if` statement nested in the `else` clause of another `if` statement, which is in turn nested in the `while` loop. When code is conditional or repeated like this, we simply indent it further to the right. The net effect is like that of prior versions, but we'll now print "low" for numbers less than 20:

```
Enter text:19
low
Enter text:20
400
Enter text:hack
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Chapter Summary

That concludes our first look at Python statement syntax. This chapter introduced the general rules for coding statements and blocks of code. As you've learned, in Python we normally code one statement per line and indent all the statements in a nested block the same amount (indentation is part of Python's syntax). However, we also looked at a few exceptions to these rules, including continuation lines and single-line tests and loops. Finally, we put these ideas to work in an interactive script that demonstrated a handful of statements and showed statement syntax in action.

In the next chapter, we'll start to dig deeper by going over each of Python's basic procedural statements in depth. As you'll see, though, all statements follow the same general rules introduced here.

Test Your Knowledge: Quiz

1. What three things are required in a C-like language but omitted in Python?
2. How is a statement normally terminated in Python?
3. How are the statements in a nested block of code normally associated in Python?
4. How can you make a single statement span multiple lines?
5. How can you code a compound statement on a single line?
6. Is there any valid reason to type a semicolon at the end of a statement in Python?
7. What is a `try` statement for?
8. What is the most common coding mistake among Python beginners?

Test Your Knowledge: Answers

1. C-like languages require parentheses around the tests in some statements, semicolons at the end of each statement, and braces around a nested block of code. Python requires none of these (but adds a `:`).
2. The end of a line terminates the statement that appears on that line. Alternatively, if more than one statement appears on the same line, they can be separated with semicolons; similarly, if a statement spans many lines, you must terminate it by closing a bracketed syntactic pair.
3. The statements (code lines) in a nested block are all indented the same number of tabs or spaces.
4. You can make a statement span many lines by enclosing part of it in parentheses, square brackets, or curly braces; the statement ends when Python sees a line that contains the closing part of the pair.
5. The body of a compound statement can be moved to the header line after the colon, but only if the body consists of only noncompound statements.
6. Only when you need to squeeze more than one statement onto a single line of code. Even then, this works only if all the statements are noncompound, and it's discouraged because it can lead to code that is difficult to read.
7. The `try` statement is used to catch and recover from exceptions (errors) in a Python script. It's often an alternative to manually checking for errors in code.
8. Forgetting to type the colon character at the end of the header line in a compound statement is the most common beginner's mistake. If you're new to Python and haven't made it yet, you probably will soon!