

---

# Introducing Python Statements

Now that you're familiar with Python's core built-in object types, this chapter begins our exploration of its fundamental statement forms. As in the previous part, 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 things you write to tell Python what your programs should do. If programs “do things with stuff,” statements are the way you specify what sort of things a program does. Python is a procedural, statement-based language; by combining statements, you specify a procedure that Python performs to satisfy a program's goals.

## Python Program Structure 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:

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

At its core, Python syntax is 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). Statements always exist in modules, which themselves are managed with statements.

## Python's Statements

Table 10-1 summarizes Python's statement set.\* 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 exotic statements like `exec` (which compiles and executes code constructed as strings) are covered later in the book, or in Python standard documentation.

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	<code>a, b, c = 'good', 'bad', 'ugly'</code>
Calls	Running functions	<code>log.write("spam, ham\n")</code>
<code>print</code>	Printing objects	<code>print 'The Killer', joke</code>
<code>if/elif/else</code>	Selecting actions	<code>if "python" in text:     print text</code>
<code>for/else</code>	Sequence iteration	<code>for x in mylist:     print x</code>
<code>while/else</code>	General loops	<code>while X &gt; Y:     print 'hello'</code>
<code>pass</code>	Empty placeholder	<code>while True:     pass</code>
<code>break, continue</code>	Loop jumps	<code>while True:     if not line: break</code>
<code>try/except/finally</code>	Catching exceptions	<code>try:     action() except:     print 'action error'</code>
<code>raise</code>	Triggering exceptions	<code>raise endSearch, location</code>
<code>import, from</code>	Module access	<code>import sys from sys import stdin</code>
<code>def, return, yield</code>	Building functions	<code>def f(a, b, c=1, *d):     return a+b+c+d[0] def gen(n):     for i in n, yield i*2</code>

\* Technically speaking, in Python 2.5, `yield` became an expression instead of a statement, and the `try/except` and `try/finally` statements were merged (the two were formerly separate statements, but we can now say both `except` and `finally` in the same `try` statement). Also, a new `with/as` statement is to be added in Python 2.6 to encode context managers—roughly speaking, it's an alternative to `try/finally` exception-related operations (in 2.5, `with/as` is an optional extension, and is not available unless you explicitly turn it on by running the statement from `__future__ import with_statement`). See Python manuals for more details. Further in the future, in 3.0, `print` and `exec` will become function calls instead of statements, and a new `nonlocal` statement will have a purpose much like that of today's `global`.

Table 10-1. Python statements (continued)

Statement	Role	Example
<code>class</code>	Building objects	<code>class subclass(Superclass):</code> <code>staticData = []</code>
<code>global</code>	Namespaces	<code>def function():</code> <code>global x, y</code> <code>x = 'new'</code>
<code>del</code>	Deleting references	<code>del data[k]</code> <code>del data[i:j]</code> <code>del obj.attr</code> <code>del variable</code>
<code>exec</code>	Running code strings	<code>exec "import " + modName</code> <code>exec code in gdict, ldict</code>
<code>assert</code>	Debugging checks	<code>assert X &gt; Y</code>
<code>with/as</code>	Context managers (2.6)	<code>with open('data') as myfile:</code> <code>process(myfile)</code>

## A Tale of Two ifs

Before we delve into the details of any of the concrete statements in Table 10-1, I want to begin our look at Python statement syntax by showing you what you are *not* going to type in Python code so you can compare and contrast it with other syntax models you might have seen in the past.

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 Perl. 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, well, 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.

More specifically, when you compare the two syntax models, you’ll notice that Python adds one new thing to the mix, and that three items that are present in the C-like language are not present in Python code.

## What Python Adds

The one new syntax component in Python is the colon character (:). All Python *compound statements* (i.e., statements that have 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 I’ve witnessed thousands of times in Python training classes. In fact, if you are new to Python, you’ll almost certainly forget the colon character very soon. Most Python-friendly editors make this mistake easy to spot, and including it eventually becomes an unconscious habit (so much so that you may start typing colons into your C++ code, too, generating many entertaining error messages from your C++ 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 in Python.

### Parentheses are optional

The first of these is the set of parentheses around the tests at the top of the statement:



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

The parentheses here are required by the syntax of many C-like languages. In Python, 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 an ex-C programmer still learning Python (I was once, too). 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. 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 (if such a state is possible...), you can continue to use semicolons at the end of each statement—the language lets you get away with them if they are present. *But don't do that either* (really!); again, doing so tells the world that you're still a C programmer who hasn't quite made the switch to Python coding. The Pythonic style is to leave off the semicolons altogether.


### End of indentation is end of block

The third and final syntax component that Python removes, and perhaps the most unusual one to soon-to-be-ex-C programmers (until they use 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 braces 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
```

By *indentation*, I mean 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.

## Why Indentation Syntax?

The indentation rule may seem unusual at first glance to programmers accustomed to C-like languages, but it is a deliberate feature of Python, and 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 forces 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 mileage may vary, but when I was still doing development on a full-time basis, I was mostly paid to work on large old C++ programs that had been worked on by many programmers over the years. Almost invariably, each programmer had his or her own style for indenting code. For example, I'd often be asked to change a while loop coded in the C++ language that began like this:

➡ 

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

Before we even get into indentation, there are three or four ways that programmers can arrange these braces in a C-like language, and organizations often have political debates and write standards manuals to address the options (which seems more than a little off-topic for the problem to be solved by programming). Ignoring that, here's the scenario I often encountered in C++ code. The first person who worked on the code intended the loop four spaces:

➡ 


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

That person eventually moved on to 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, 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 (he says, sarcastically). 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. 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)`), even though it looks visually as though it is associated with the outer `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 is the one it is associated with logically (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. Early in my career, I worked at a successful company that developed systems software in the C language, where consistent indentation is not required. Even so, when we checked our code into source control at the end of the day, this company ran an automated script that analyzed the indentation used in the code. If the script noticed that we'd indented our code inconsistently, we received an automated email about it the next morning—and so did our bosses!

My 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.

Finally, keep in mind that nearly every programmer-friendly text editor in use today has built-in support for Python's syntax model. In the IDLE Python 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 absolute standard for how to indent: four spaces or one tab per level is common, but it's up to you to decide how and how much you wish to indent. Indent further to the right for further nested blocks, and less to close the prior block. Moreover, generating tabs instead of braces is no more difficult in practice for tools that must output Python code. In general, do what you should be doing in a C-like language, anyhow, but get rid of the braces, and 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 customization of both statements and nested statement blocks.

### Statement rule special cases

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 statements on one line*

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, prints, and function calls. Compound statements 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



dictionary 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:

```
mlist = [111,
         222,
         333]
```

Because the code is enclosed in a square brackets pair, Python simply drops down to the next line until it encounters the closing bracket. Dictionaries can also span lines this way, 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.

Parentheses are the catchall device—because any expression can be wrapped up 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 with 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 'spam' * 3
```

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

```
X = A + B + \
    C + D
```

But this alternative technique is dated, and somewhat frowned on today because it's difficult to notice and maintain the backslashes, and it's fairly brittle (there can be no spaces after the backslash). It's also another throwback to the C language, where it is commonly used in “`#define`” macros; again, when in Pythonland, do as Pythonistas do, not as C programmers do.

### 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, 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 loops, and so on. Here again, 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,

prints, function calls, 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 we’ll meet later) must also be on separate lines of their own. The body can consist of multiple simple statements separated by semicolons, but this tends to be frowned on.

In general, even though it’s not always required, if you keep all your statements on individual lines, and always indent your nested blocks, your code will be easier to read and change in the future. 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), let’s move on to the next section and write some real code.


## A Quick Example: Interactive Loops

We’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, realistic example that demonstrates the way that statement syntax and statement nesting come together in practice, and introduces a few statements along the way.

### 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.

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



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

This code makes use of a few new ideas:

- 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).

- The `raw_input` built-in function we met earlier in the book 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 means essentially “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, but the form used here is very common in Python code.

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 the end of the source file or a lesser-indented statement will terminate the loop body block.

When run, here is the sort of interaction we get from this code:

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```

## 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 to discourage users who happen to be obsessed with youth. We might try statements like these to achieve the desired effect:

```
>>> reply = '20'
>>> reply ** 2
...error text omitted...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

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

```
>>> int(reply) ** 2
400
```

Armed with this information, we can now recode our loop to perform the necessary math:

```
→ while True:
    reply = raw_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 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
...error text omitted...
ValueError: invalid literal for int() with base 10: 'xxx'
```

The built-in int function raises an exception here in the face of a mistake. 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 = '123'
>>> 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 errors:

```
→ while True:
    reply = raw_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 an associated 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 they all line up vertically (i.e., share the same level of indentation). The `if` statement spans from the word `if` to the start of the `print` statement on the last line if 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 is natural once you get the hang of it.


When we run our new script, its code catches errors before they occur, and prints an (arguably silly) error message to demonstrate:



```
Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
```

## 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 the last part of this book, but, as a preview, using a `try` here can lead to code that some would claim is simpler than the prior version:



```
while True:
    reply = raw_input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print 'Bad!' * 8
    else:
        print int(reply) ** 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 up in an exception handler for cases when it doesn't. This `try` statement is 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, followed by an `else` part to be run if no exception is raised in the `try` part. Python first runs `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'll see, `else` can appear in `if` statements in Python, but also in `try` statements and loops—its indentation tells you what statement it is a part of.

Again, we'll come back to the `try` statement later in the 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 is a very general approach to dealing with unusual cases.


## Nesting Code Three Levels Deep

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



```
while True:
    reply = raw_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 includes 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 the prior versions, but we'll now print “low” for numbers less than 20:



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

## Chapter Summary

That concludes our quick 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.

## Chapter 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 statement span over 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?

## Quiz 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.
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 terminated with semicolons; similarly, if a statement spans many lines, you must terminate it by closing a bracketed syntactic pair.
3. The statements in a nested block are all indented the same number of tabs or spaces.
4. A statement can be made to 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 only works 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 usually an alternative to manually checking for errors in your 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.