

MODULE I

Chapter 1 : Python Basics

Chapter 2 : Flow Control

Chapter 3 : Functions

CSE, HIT, Nidasoshi

CSE, HIT, Nidasoshi

PYTHON BASICS

The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

You will, however, have to learn some basic programming concepts before you can do anything. Like a wizard in training, you might think these concepts seem arcane and tedious, but with some knowledge and practice, you'll be able to command your computer like a magic wand and perform incredible feats.

This chapter has a few examples that encourage you to type into the *interactive shell*, also called the *REPL* (Read-Evaluate-Print Loop), which lets you run (or *execute*) Python instructions one at a time and instantly shows you the results. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

ENTERING EXPRESSIONS INTO THE INTERACTIVE SHELL

You can run the interactive shell by launching the Mu editor, which you should have downloaded when going through the setup instructions in the Preface. On Windows, open the Start menu, type "Mu," and open the Mu app. On macOS, open your Applications folder and double-click **Mu**. Click the **New** button and save an empty file as *blank.py*. When you run this blank file by clicking the **Run** button or pressing F5, it will open the interactive shell, which will open as a new pane that opens at the bottom of the Mu editor's window. You should see a `>>>` prompt in the interactive shell.

Enter `2 + 2` at the prompt to have Python do some simple math. The Mu window should now look like this:

```
>>> 2 + 2
4
>>>
```

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as 2) and *operators* (such as +), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, 4. A single value with no operators is also considered an expression, though it evaluates only to itself, as

shown here:

```
>>> 2
```

```
2
```

You can use plenty of other operators in Python expressions, too. For example, Table 1-1 lists all the math operators in Python.

Table 1-1: Math Operators from Highest to Lowest Precedence

Operator	Operation	Example	Evaluates to . . .
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

The *order of operations* (also called *precedence*) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Whitespace in between the operators and values doesn't matter for Python (except for the indentation at the beginning of the line), but a single space is convention. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
```

```
20
```

```
>>> (2 + 3) * 6
```

```
30
```

```
>>> 48565878 * 578453
```

```
28093077826734
```

```
>>> 2 ** 8
```



```
>>> 23 / 7
```

```
3.2857142857142856
```

```
>>> 23 // 7
```

```
3
```

```
>>> 23 % 7
```

```
2
```

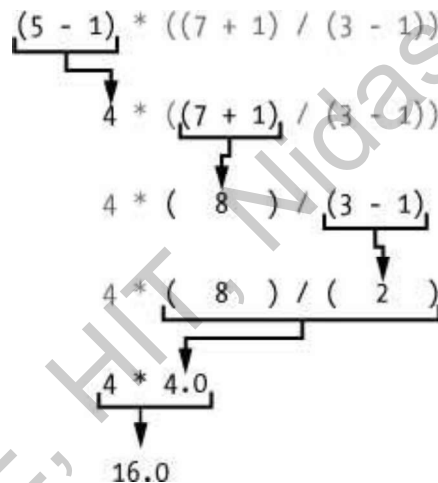
```
>>> 2 + 2
```

```
4
```

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
```

```
16.0
```

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it down to a single value. Python will keep evaluating parts of the expression until it becomes a single value, as shown here:



These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you enter a bad Python instruction, Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```
>>> 5 +
```

```
File "<stdin>", line 1
```

```
5 +
```

```
^
```

```
SyntaxError: invalid syntax
```

```
>>> 42 + 5 + * 2
```

```
File "<stdin>", line 1
```

```
42 + 5 + * 2
```

```
^
```

```
SyntaxError: invalid syntax
```

You can always test to see whether an instruction works by entering it into the interactive shell. Don't worry about breaking the computer: the worst that could happen is that Python responds with an error message. Professional software developers get error messages while writing code all the time.

THE INTEGER, FLOATING-POINT, AND STRING DATA TYPES

Remember that expressions are just values combined with operators, and they always evaluate down to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most common data types in Python are listed in Table 1-2. The values -2 and 30, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as 3.14, are called *floating-point numbers* (or *floats*). Note that even though the value 42 is an integer, the value 42.0 would be a floating-point number.

Table 1-2: Common Data Types

Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

Python programs can also have text values called *strings*, or *strs* (pronounced “stirs”). Always surround your string in single quote (') characters (as in 'Hello' or 'Goodbye cruel world!') so Python knows where the string begins and ends. You can even have a string with no characters in it, "", called a *blank string* or an *empty string*. Strings are explained in greater detail in Chapter 4.

If you ever see the error message `SyntaxError: EOL while scanning string literal`, you probably forgot the final single quote character at the end of the string, such as in this example:

```
>>> 'Hello, world!
```

```
SyntaxError: EOL while scanning string literal
```

STRING CONCATENATION AND REPLICATION

The meaning of an operator may change based on the data types of the values next to it. For example, `+` is the addition operator when it operates on two integers or floating-point values. However, when `+` is used on two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the `+` operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Alice' + 42
TypeError: can only concatenate str (not "int") to str
```

The error message `can only concatenate str (not "int") to str` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string because Python cannot do this automatically. (Converting data types will be explained in “Dissecting Your Program” on page 13 when we talk about the `str()`, `int()`, and `float()` functions.)

The `*` operator multiplies two integer or floating-point values. But when the `*` operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original string a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The `*` operator can be used with only two numeric values (for multiplication), or one string value and one integer value (for string replication). Otherwise, Python will just display an error message, like the following:

```
>>> 'Alice' * 'Bob'
```

Traceback (most recent call last):

```
File "<pyshell#32>", line 1, in <module>
```

```
'Alice' * 'Bob'
```

TypeError: can't multiply sequence by non-int of type 'str'

```
>>> 'Alice' * 5.0
```

Traceback (most recent call last):

```
File "<pyshell#33>", line 1, in <module>
```

```
'Alice' * 5.0
```

TypeError: can't multiply sequence by non-int of type 'float'

It makes sense that Python wouldn't understand these expressions: you can't multiply two words, and it's hard to replicate an arbitrary string a fractional number of times.

STORING VALUES IN VARIABLES

A *variable* is like a box in the computer's memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, then a variable named `spam` will have the integer value 42 stored in it.

Think of a variable as a labeled box that a value is placed in, as in Figure 1-1.

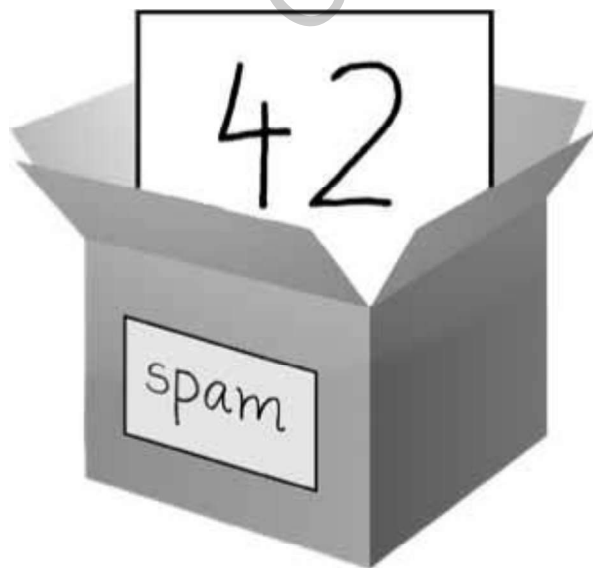


Figure 1-1: `spam = 42` is like telling the program, “The variable `spam` now has the integer value 42 in it.”

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
>>> spam
40
>>> eggs = 2
❷ >>> spam + eggs
42
>>> spam + eggs + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to 42 instead of 40 at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in Figure 1-2, the `spam` variable in this example stores 'Hello' until you replace the string with 'Goodbye'.



Figure 1-2: When a new value is assigned to a variable, the old one is forgotten.

Variable Names

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! Most of this book's examples (and Python's documentation) use generic variable names like `spam`, `eggs`, and `bacon`, which come from the Monty Python "Spam" sketch. But in your programs, a descriptive name will help make your code more readable.

Though you can name your variables almost anything, Python does have some naming restrictions. Table 1-3 has examples of legal variable names. You can name a variable anything as long as it obeys the following three rules:

- It can be only one word with no spaces.
- It can use only letters, numbers, and the underscore (`_`) character.
- It can't begin with a number.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
<code>current_balance</code>	<code>current-balance</code> (hyphens are not allowed)
<code>currentBalance</code>	<code>current balance</code> (spaces are not allowed)
<code>account4</code>	<code>4account</code> (can't begin with a number)

Valid variable names**Invalid variable names**

_42

42 (can't begin with a number)

TOTAL_SUM

TOTAL_\$UM (special characters like \$ are not allowed)

hello

'hello' (special characters like ' are not allowed)

Variable names are case-sensitive, meaning that `spam`, `SPAM`, `Spam`, and `sPaM` are four different variables. Though `Spam` is a valid variable you can use in a program, it is a Python convention to start your variables with a lowercase letter.

This book uses *camelcase* for variable names instead of underscores; that is, variables lookLikeThis instead of looking_like_this. Some experienced programmers may point out that the official Python code style, PEP 8, says that underscores should be used. I unapologetically prefer camelcase and point to the “A Foolish Consistency Is the Hobgoblin of Little Minds” section in PEP 8 itself:

Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment.

YOUR FIRST PROGRAM

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs, you'll type the instructions into the file editor. The *file editor* is similar to text editors such as Notepad or TextMate, but it has some features specifically for entering source code. To open a new file in Mu, click the **New** button on the top row.

The window that appears should contain a cursor awaiting your input, but it's different from the interactive shell, which runs Python instructions as soon as you press `ENTER`. The file editor lets you type in many instructions, save the file, and run the program. Here's how you can tell the difference between the two:


- The interactive shell window will always be the one with the `>>>` prompt.
- The file editor window will not have the `>>>` prompt.

Now it's time to create your first program! When the file editor window opens, enter the following into it:

❶ # This program says hello and asks for my name.

```
❷ print('Hello, world!')
   print('What is your name?') # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
   print(len(myName))
❻ print('What is your age?') # ask for their age
   myAge = input()
   print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Once you've entered your source code, save it so that you won't have to retype it each time you start Mu. Click the **Save** button, enter *hello.py* in the File Name field, and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit Mu, you won't lose the code. As a shortcut, you can press CTRL-S on Windows and Linux or  on macOS to save your file.

Once you've saved, let's run our program. Press the **F5** key. Your program should run in the interactive shell window. Remember, you have to press **F5** from the file editor window, not the interactive shell window. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Python 3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018, 19:02:22) [MSC v.1913 64 bit
(AMD64)] on win32
```

```
Type "copyright", "credits" or "license()" for more information.
```

```
>>> ===== RESTART
```

```
=====
```

```
>>>
```

```
Hello, world!
```

```
What is your name?
```

```
Al
```

```
It is good to meet you, Al
```

```
The length of your name is:
```

```
2
```

```
What is your age?
```

```
4
```

```
You will be 5 in a year.
```

```
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.)

You can close the file editor by clicking the X at the top of the window. To reload a saved program, select **File►Open...** from the menu. Do that now, and in the window that appears, choose *hello.py* and click the **Open** button. Your previously saved *hello.py* program should open in the file editor window.

You can view the execution of a program using the Python Tutor visualization tool at <http://pythontutor.com/>. You can see the execution of this particular program at <https://autbor.com/hello.py/>. Click the forward button to move through each step of the program's execution. You'll be able to see how the variables' values and the output change.

DISSECTING YOUR PROGRAM

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*.

```
❶ # This program says hello and asks for my name.
```

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program isn't working. You can remove the # later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This can make your code easier to read, like paragraphs in a book.

The print() Function

The `print()` function displays the string value inside its parentheses on the screen.

```
❷ print('Hello, world!')  
print('What is your name?') # ask for their name
```

The line `print('Hello, world!')` means “Print out the text in the string 'Hello, world!'.” When Python executes this line, you say that Python is *calling* the `print()` function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

NOTE

You can also use this function to put a blank line on the screen; just call `print()` with nothing in between the parentheses.

When you write a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book, you’ll see `print()` rather than `print`. Chapter 3 describes functions in more detail.

The input() Function

The `input()` function waits for the user to type some text on the keyboard and press ENTER.

```
❸ myName = input()
```

This function call evaluates to a string equal to the user’s text, and the line of code assigns the `myName` variable to this string value.

You can think of the `input()` function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to `myName = 'Al'`.

If you call `input()` and see an error message, like `NameError: name 'Al' is not defined`, the problem is that you’re running the code with Python 2 instead of Python 3.

Printing the User’s Name

The following call to `print()` actually contains the expression `'It is good to meet you, ' + myName` between the parentheses.

```
❹ print('It is good to meet you, ' + myName)
```

Remember that expressions can always evaluate to a single value. If 'Al' is the value stored in `myName` on line ❸, then this expression evaluates to `'It is good to meet you, Al'`. This single string value is then passed to `print()`, which prints it on the screen.

The len() Function

You can pass the `len()` function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

```
❶ print('The length of your name is:')  
print(len(myName))
```

Enter the following into the interactive shell to try this:

```
>>> len('hello')  
5  
>>> len('My very energetic monster just scarfed nachos.')46  
>>> len('')  
0
```

Just like those examples, `len(myName)` evaluates to an integer. It is then passed to `print()` to be displayed on the screen. The `print()` function allows you to pass it either integer values or string values, but notice the error that shows up when you type the following into the interactive shell:

```
>>> print('I am ' + 29 + ' years old.')
```

Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
print('I am ' + 29 + ' years old.')

TypeError: can only concatenate str (not "int") to str

The `print()` function isn't causing that error, but rather it's the expression you tried to pass to `print()`. You get the same error message if you type the expression into the interactive shell on its own.

```
>>> 'I am ' + 29 + ' years old.'
```

Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module>
'I am ' + 29 + ' years old.'

TypeError: can only concatenate str (not "int") to str

Python gives an error because the `+` operator can only be used to add two integers together or concatenate two strings. You can't add an integer to a string, because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

The str(), int(), and float() Functions

If you want to concatenate an integer such as 29 with a string to pass to print(), you'll need to get the value '29', which is the string form of 29. The str() function can be passed an integer value and will evaluate to a string value version of the integer, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because str(29) evaluates to '29', the expression 'I am ' + str(29) + ' years old.' evaluates to 'I am ' + '29' + ' years old.', which in turn evaluates to 'I am 29 years old.'. This is the value that is passed to the print() function.

The str(), int(), and float() functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions and watch what happens.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the str(), int(), and float() functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The str() function is handy when you have an integer or float that you want to concatenate to a string. The int() function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the input() function always

returns a string, even if the user enters a number. Enter `spam = input()` into the interactive shell and enter `101` when it waits for your text.

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside `spam` isn't the integer `101` but the string `'101'`. If you want to do math using the value in `spam`, use the `int()` function to get the integer form of `spam` and then store this as the new value in `spam`.

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the `spam` variable as an integer instead of a string.

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to `int()` that it cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
```

```
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

The `int()` function is also useful if you need to round a floating-point number down.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

You used the `int()` and `str()` functions in the last three lines of your program to get a value of the appropriate data type for the code.

```
⑥ print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

TEXT AND NUMBER EQUIVALENCE

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

```
>>> 42 == '42'
```

```
False
```

```
>>> 42 == 42.0
```

```
True
```

```
>>> 42.0 == 0042.000
```

```
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.

The `myAge` variable contains the value returned from `input()`. Because the `input()` function always returns a string (even if the user typed in a number), you can use the `int(myAge)` code to return an integer value of the string in `myAge`. This integer value is then added to 1 in the expression `int(myAge) + 1`.

The result of this addition is passed to the `str()` function: `str(int(myAge) + 1)`. The string value returned is then concatenated with the strings 'You will be ' and ' in a year.' to evaluate to one large string value. This large string is finally passed to `print()` to be displayed on the screen.

Let's say the user enters the string '4' for `myAge`. The string '4' is converted to an integer, so you can add one to it. The result is 5. The `str()` function converts the result back to a string, so you can concatenate it with the second string, 'in a year.', to create the final message. These evaluation steps would look something like the following:

```

print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str( 4 + 1 ) + ' in a year.')
print('You will be ' + str( 5 ) + ' in a year.')
print('You will be ' + '5' + ' in a year.')
print('You will be 5' + ' in a year.')
print('You will be 5 in a year.')

```

SUMMARY

You can compute expressions with a calculator or enter string concatenations with a word processor. You can even do string replication easily by copying and pasting text. But expressions, and their component values—operators, variables, and function calls—are the basic building blocks that make programs. Once you know how to handle these elements, you will be able to instruct Python to operate on large amounts of data for you.

It is good to remember the different types of operators (+, -, *, /, //, %, and ** for math operations, and + and * for string operations) and the three data types (integers, floating-point numbers, and strings) introduced in this chapter.

I introduced a few different functions as well. The `print()` and `input()` functions handle simple text output (to the screen) and input (from the keyboard). The `len()` function takes a string and evaluates to an `int` of the number of characters in the string. The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, or floating-point number form of the value they are passed.

In the next chapter, you'll learn how to tell Python to make intelligent decisions about what code to run, what code to skip, and what code to repeat based on the values it has. This is known as *flow control*, and it allows you to write programs that make intelligent decisions.

PRACTICE QUESTIONS

1. Which of the following are operators, and which are values?

*

'hello'

-88.8

-

/
+
5

2. Which of the following is a variable, and which is a string?

spam
'spam'

3. Name three data types.

4. What is an expression made up of? What do all expressions do?

5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?

6. What does the variable `bacon` contain after the following code runs?

```
bacon = 20  
bacon + 1
```

7. What should the following two expressions evaluate to?

```
'spam' + 'spamsam'  
'spam' * 3
```

8. Why is `eggs` a valid variable name while `100` is invalid?

9. What three functions can be used to get the integer, floating-point number, or string version of a value?

10. Why does this expression cause an error? How can you fix it?

```
'I have eaten ' + 99 + ' burritos.'
```

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled “Built-in Functions.” Skim the list of other functions Python has, look up what the `round()` function does, and experiment with it in the interactive shell.

FLOW CONTROL

So, you know the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another like a weekend errand list. Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

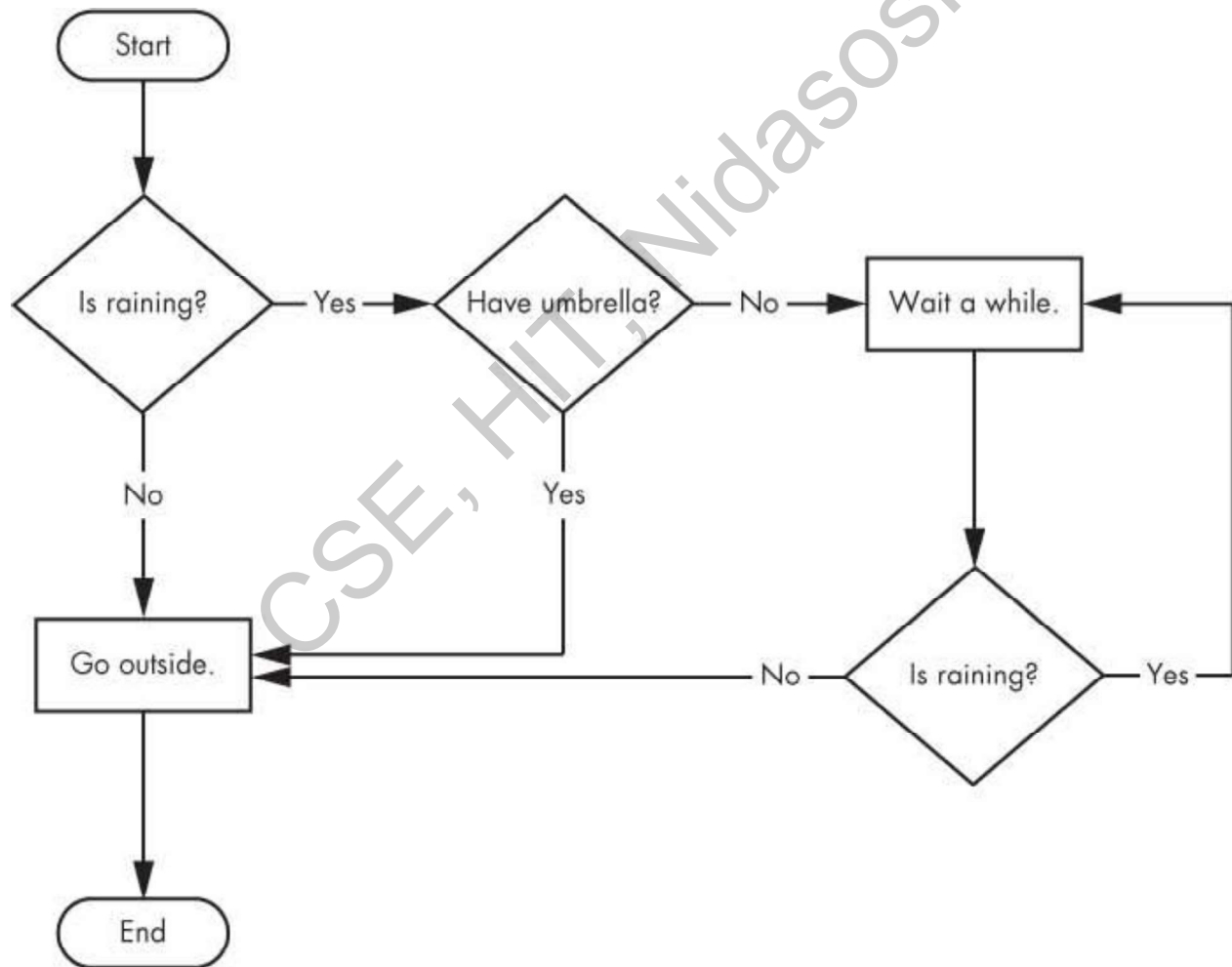


Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

BOOLEAN VALUES

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: True and False. (Boolean is capitalized because the data type is named after mathematician George Boole.) When entered as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Enter the following into the interactive shell. (Some of these instructions are intentionally incorrect, and they'll cause error messages to appear.)

❶ `>>> spam = True`

`>>> spam`

True

❷ `>>> true`

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

true

NameError: name 'true' is not defined

❸ `>>> True = 2 + 2`

SyntaxError: can't assign to keyword

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use True and False for variable names ❸, Python will give you an error message.

COMPARISON OPERATORS

Comparison operators, also called *relational operators*, compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1: Comparison Operators

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal to
<code>>=</code>	Greater than or equal to

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, `==` (equal to) evaluates to `True` when the values on both sides are the same, and `!=` (not equal to) evaluates to `True` when the two values are different. The `==` and `!=` operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
```

True

❶ `>>> 42 == '42'`

False

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` ❶ evaluates to `False` because Python considers the integer `42` to be different from the string `'42'`.

The `<`, `>`, `<=`, and `>=` operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
```

True

```
>>> 42 > 100
```

False

```
>>> 42 < 42
```

False

```
>>> eggCount = 42
```

❶ `>>> eggCount <= 42`

True

```
>>> myAge = 29
```

❷ `>>> myAge >= 10`

True

THE DIFFERENCE BETWEEN THE `==` AND `=` OPERATORS

You might have noticed that the `==` operator (equal to) has two equal signs, while the `=` operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The `==` operator (equal to) asks whether two values are the same as each other.
- The `=` operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the `==` operator (equal to) consists of two characters, just like the `!=` operator (not equal to) consists of two characters.

You'll often use comparison operators to compare a variable's value to some other value, like in the `eggCount <= 42` ❶ and `myAge >= 10` ❷ examples. (After all, instead of entering `'dog' != 'cat'` in your code, you could have just entered `True`.) You'll see more examples of this later when you learn about flow control statements.

BOOLEAN OPERATORS

The three Boolean operators (`and`, `or`, and `not`) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the `and` operator.

Binary Boolean Operators

The `and` and `or` operators always take two Boolean values (or expressions), so they're considered *binary* operators. The `and` operator evaluates an expression to `True` if *both* Boolean values are `True`; otherwise, it evaluates to `False`. Enter some expressions using `and` into the interactive shell to see it in action.

```
>>> True and True
```

```
True
```

```
>>> True and False
```

```
False
```

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the `and` operator.

Table 2-2: The `and` Operator's Truth Table

Expression	Evaluates to . . .
True and True	True
True and False	False
False and True	False
False and False	False

On the other hand, the `or` operator evaluates an expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`.

```
>>> False or True
```

```
True
```

```
>>> False or False
```

```
False
```

You can see every possible outcome of the `or` operator in its truth table, shown in Table 2-3.

Table 2-3: The `or` Operator's Truth Table

Expression	Evaluates to . . .
True or True	True
True or False	True
False or True	True
False or False	False

The not Operator

Unlike `and` or `or`, the `not` operator operates on only one Boolean value (or expression). This makes it a *unary* operator. The `not` operator simply evaluates to the opposite Boolean value.

```
>>> not True
```

```
False
```

```
❶ >>> not not not not True
```

```
True
```

Much like using double negatives in speech and writing, you can nest `not` operators ❶, though there's never not no reason to do this in real programs. Table 2-4 shows the truth table for `not`.

Table 2-4: The `not` Operator's Truth Table

Expression	Evaluates to . . .
not True	False
not False	True

MIXING BOOLEAN AND COMPARISON OPERATORS

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the `and`, `or`, and `not` operators are called Boolean operators because they always operate on the Boolean values `True` and `False`. While expressions like `4 < 5` aren't Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

```
>>> (4 < 5) and (5 < 6)
```

```
True
```

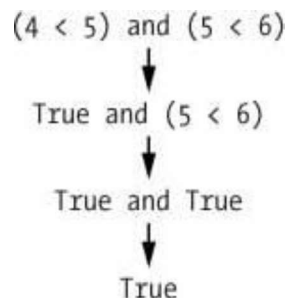
```
>>> (4 < 5) and (9 < 6)
```

```
False
```

```
>>> (1 == 2) or (2 == 2)
```

```
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for `(4 < 5) and (5 < 6)` as the following:



You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
```

```
True
```

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the `not` operators first, then the `and` operators, and then the `or` operators.

ELEMENTS OF FLOW CONTROL

Flow control statements often start with a part called the *condition* and are always followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, True or False. A flow control statement decides what to do based on whether its condition is True or False, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

- Blocks begin when the indentation increases.
- Blocks can contain other blocks.
- Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
    ❶ print('Hello, Mary')
    if password == 'swordfish':
        ❷ print('Access granted.')
    else:
        ❸ print('Wrong password.')
```

You can view the execution of this program at <https://autbor.com/blocks/>. The first block of code ❶ starts at the line `print('Hello, Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access Granted.)`. The third block ❸ is also one line long: `print('Wrong password.)`.

PROGRAM EXECUTION

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. The *program execution* (or simply, *execution*) is a term for the current instruction being executed. If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find yourself jumping around the source code based on conditions, and you'll probably skip entire clauses.

FLOW CONTROL STATEMENTS

Now, let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

if Statements

The most common type of flow control statement is the *if* statement. An *if* statement's clause (that is, the block following the *if* statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an *if* statement could be read as, "If this condition is true, execute the code in the clause." In Python, an *if* statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the *if* clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend `name` was assigned some value earlier.)

```
if name == 'Alice':  
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This *if* statement's clause is the block with `print('Hi, Alice.')`. Figure 2-2 shows what a flowchart of this code would look like.

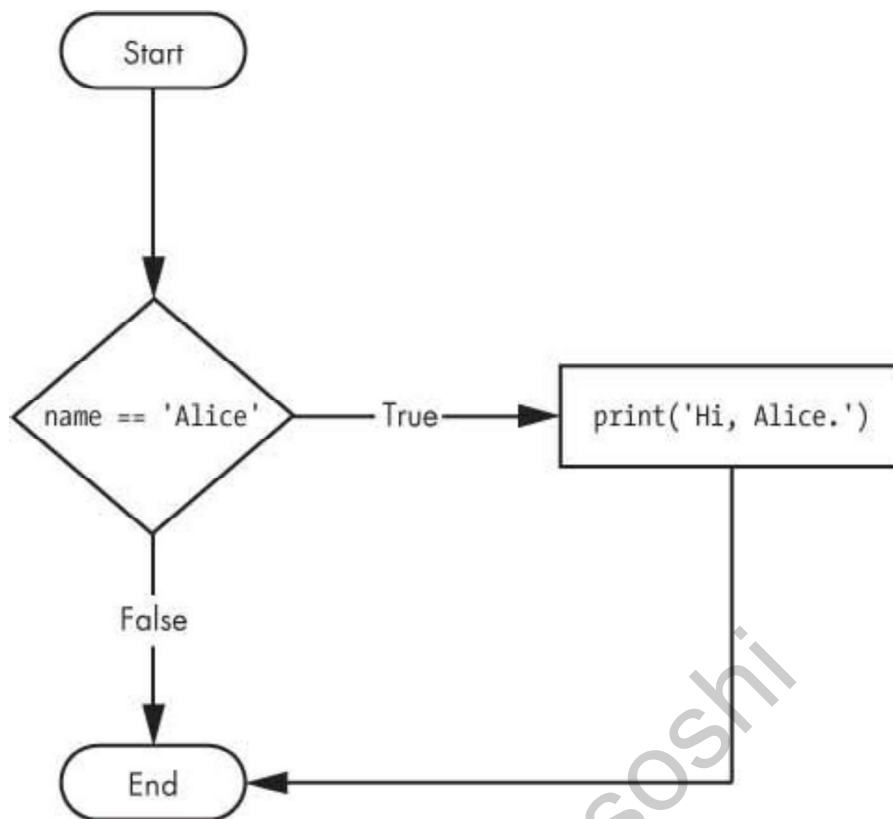


Figure 2-2: The flowchart for an if statement

else Statements

An if clause can optionally be followed by an else statement. The else clause is executed only when the if statement's condition is False. In plain English, an else statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An else statement doesn't have a condition, and in code, an else statement always consists of the following:

- The else keyword
- A colon
- Starting on the next line, an indented block of code (called the else clause)

Returning to the Alice example, let's look at some code that uses an else statement to offer a different greeting if the person's name isn't Alice.

```
if name == 'Alice':  
    print('Hi, Alice.')  
else:  
    print('Hello, stranger.')
```

Figure 2-3 shows what a flowchart of this code would look like.

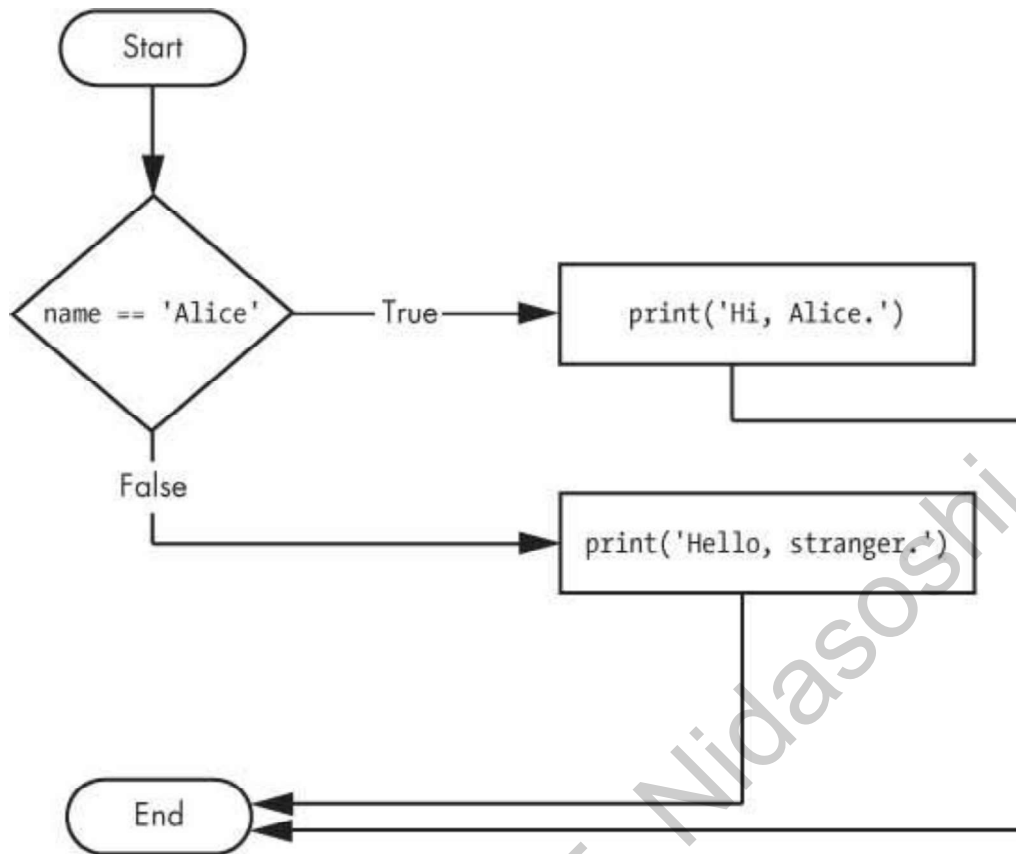


Figure 2-3: The flowchart for an else statement

elif Statements

While only one of the if or else clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The elif statement is an “else if” statement that always follows an if or another elif statement. It provides another condition that is checked only if all of the previous conditions were False. In code, an elif statement always consists of the following:

- The elif keyword
- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the elif clause)

Let's add an elif to the name checker to see this statement in action.

```
if name == 'Alice':  
    print('Hi, Alice.')  
elif age < 12:  
    print('You are not Alice, kiddo.')
```

This time, you check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in Figure 2-4.

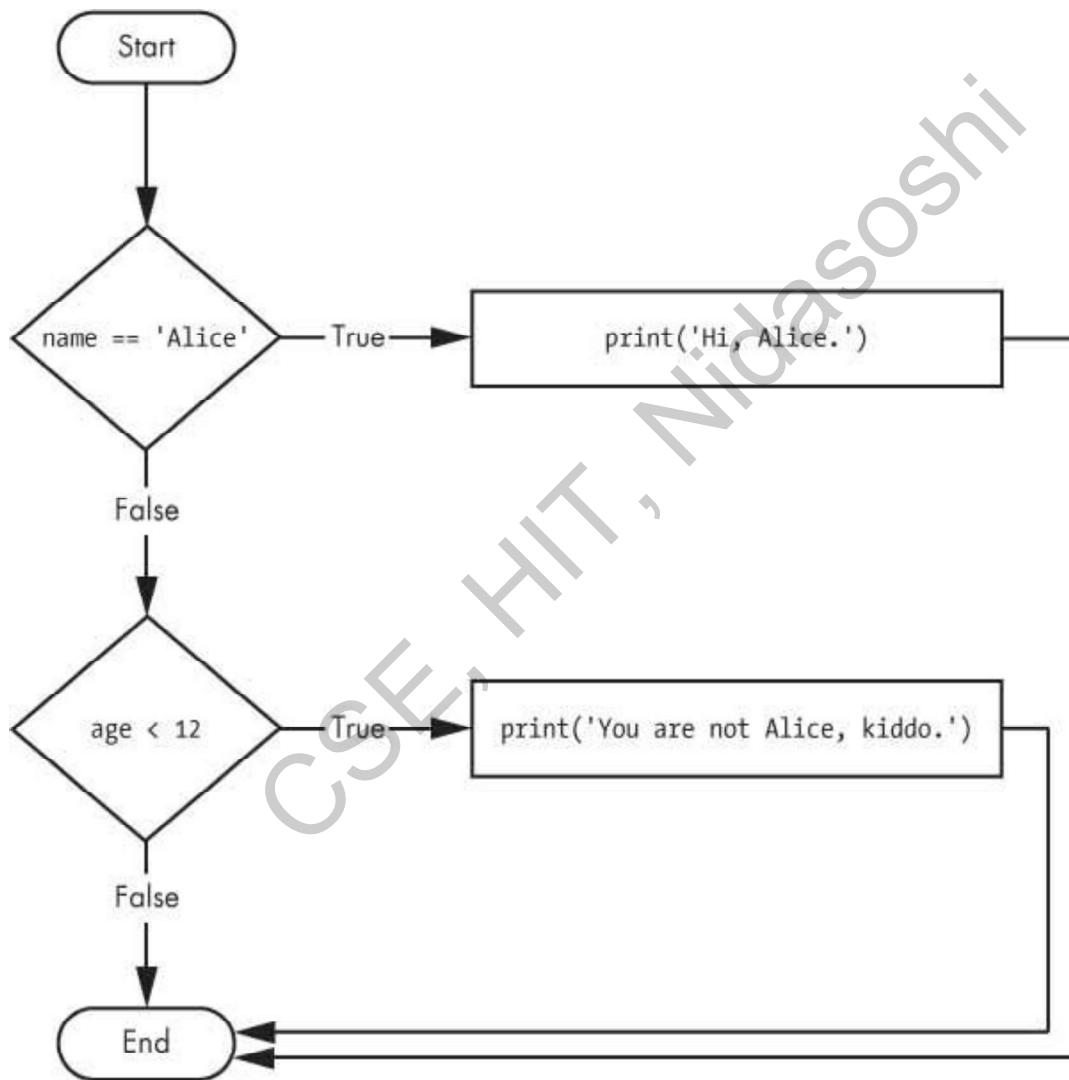


Figure 2-4: The flowchart for an elif statement

The elif clause executes if `age < 12` is True and `name == 'Alice'` is False. However, if both of the conditions are False, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of elif statements, only

one or none of the clauses will be executed. Once one of the statements' conditions is found to be True, the rest of the `elif` clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as `vampire.py`:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

You can view the execution of this program at <https://autbor.com/vampire/>. Here, I've added two more `elif` statements to make the name checker greet a person with different answers based on age. Figure 2-5 shows the flowchart for this.

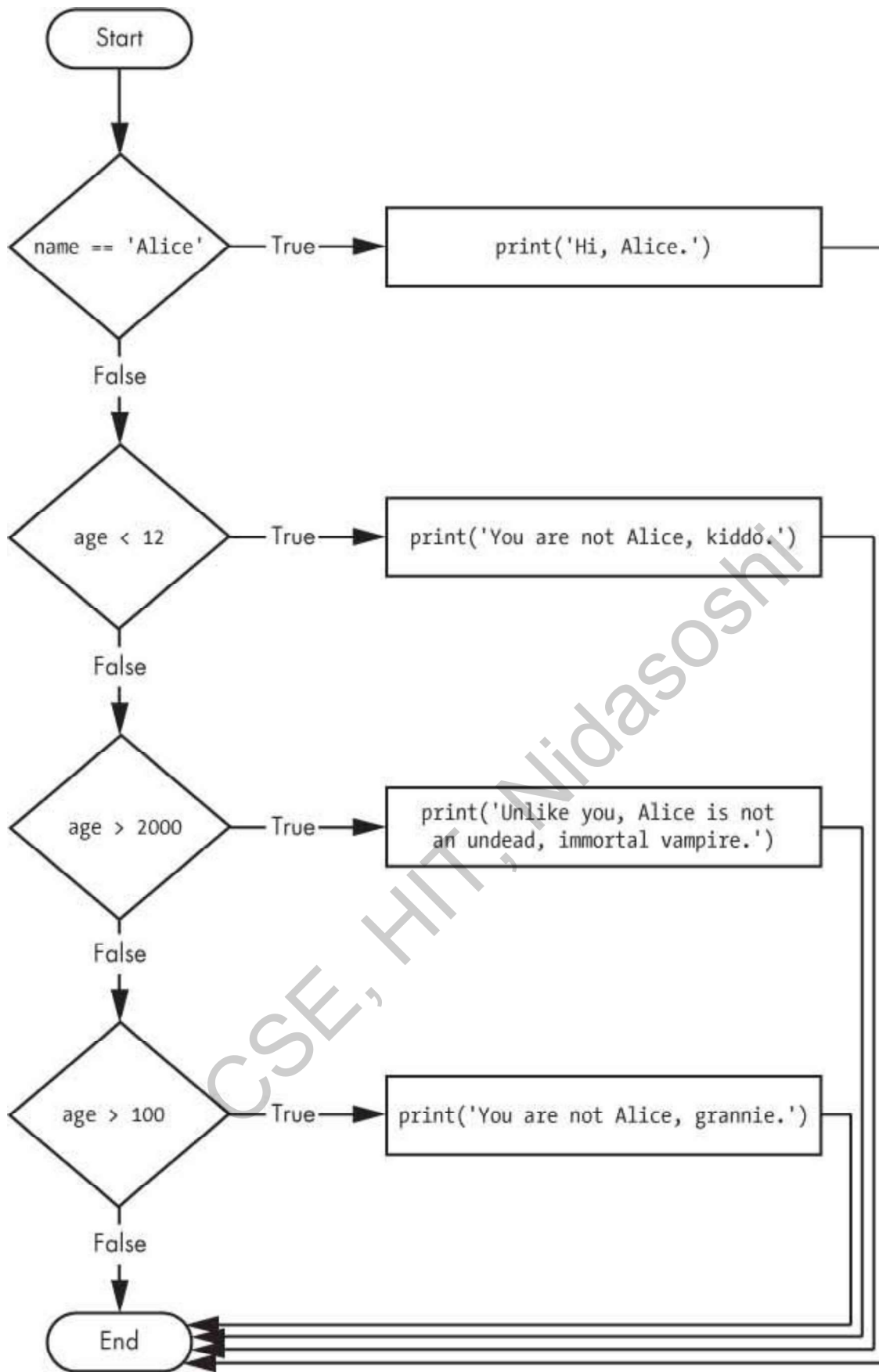


Figure 2-5: The flowchart for multiple elif statements in the vampire.py program

The order of the elif statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the elif clauses are automatically skipped once a True condition has been found, so if you swap around some of the clauses in

vampire.py, you run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
❶ elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

You can view the execution of this program at <https://aotbor.com/vampire2/>. Say the age variable contains the value 3000 before this code is executed. You might expect the code to print the string 'Unlike you, Alice is not an undead, immortal vampire.'. However, because the age > 100 condition is True (after all, 3,000 *is* greater than 100) ❶, the string 'You are not Alice, grannie.' is printed, and the rest of the elif statements are automatically skipped. Remember that at most only one of the clauses will be executed, and for elif statements, the order matters!

Figure 2-6 shows the flowchart for the previous code. Notice how the diamonds for age > 100 and age > 2000 are swapped.

Optionally, you can have an else statement after the last elif statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every if and elif statement are False, then the else clause is executed. For example, let's re-create the Alice program to use if, elif, and else clauses.

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

You can view the execution of this program at <https://autbor.com/littlekid/>. Figure 2-7 shows the flowchart for this new code, which we'll save as *littleKid.py*.

In plain English, this type of flow control structure would be “If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else.” When you use `if`, `elif`, and `else` statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-6. First, there is always exactly one `if` statement. Any `elif` statements you need should follow the `if` statement. Second, if you want to be sure that at least one clause is executed, close the structure with an `else` statement.

CSE, HIT, Nidasoshi

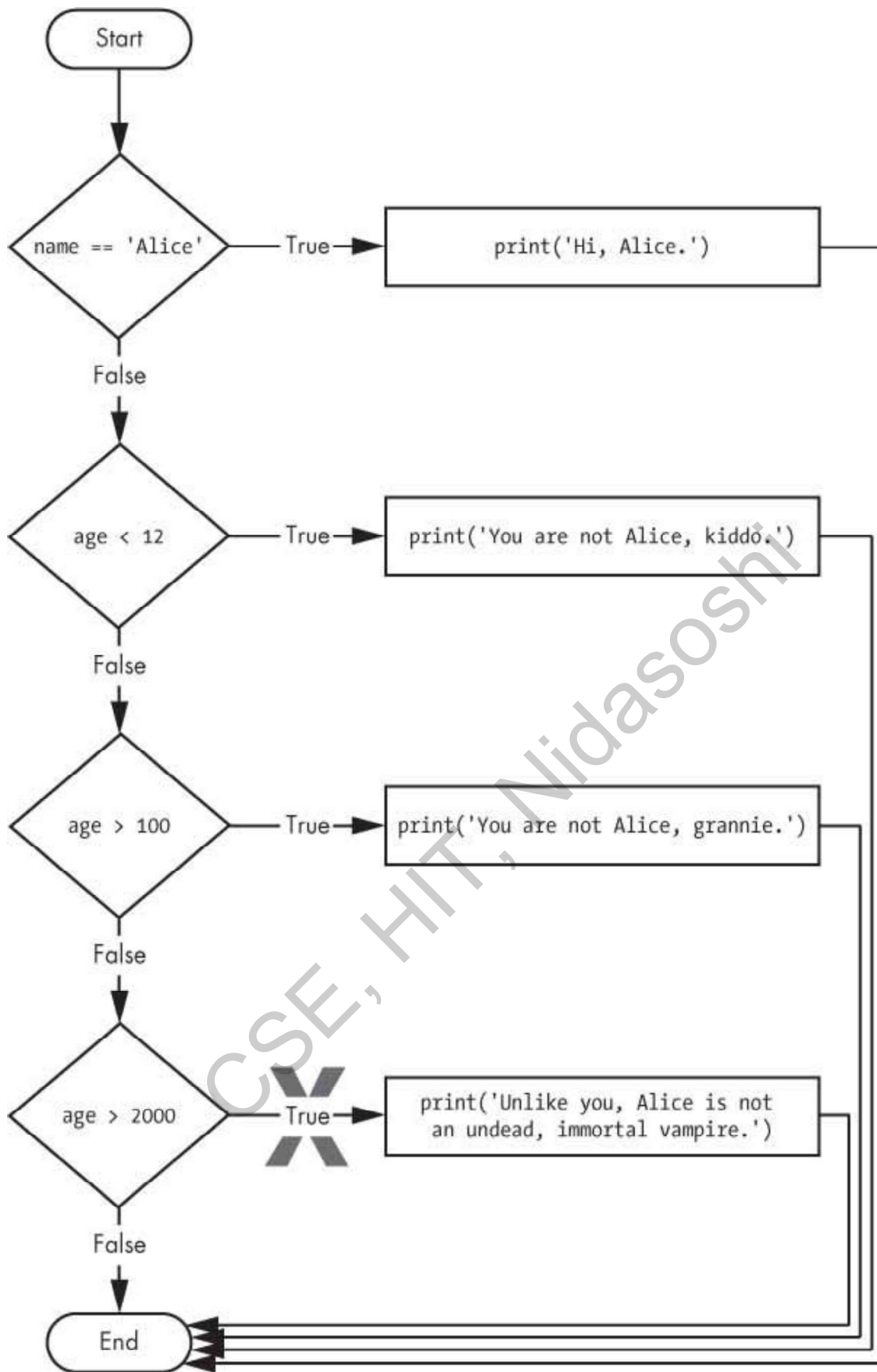


Figure 2-6: The flowchart for the vampire2.py program. The X path will logically never happen, because if age were greater than 2000, it would have already been greater than 100.

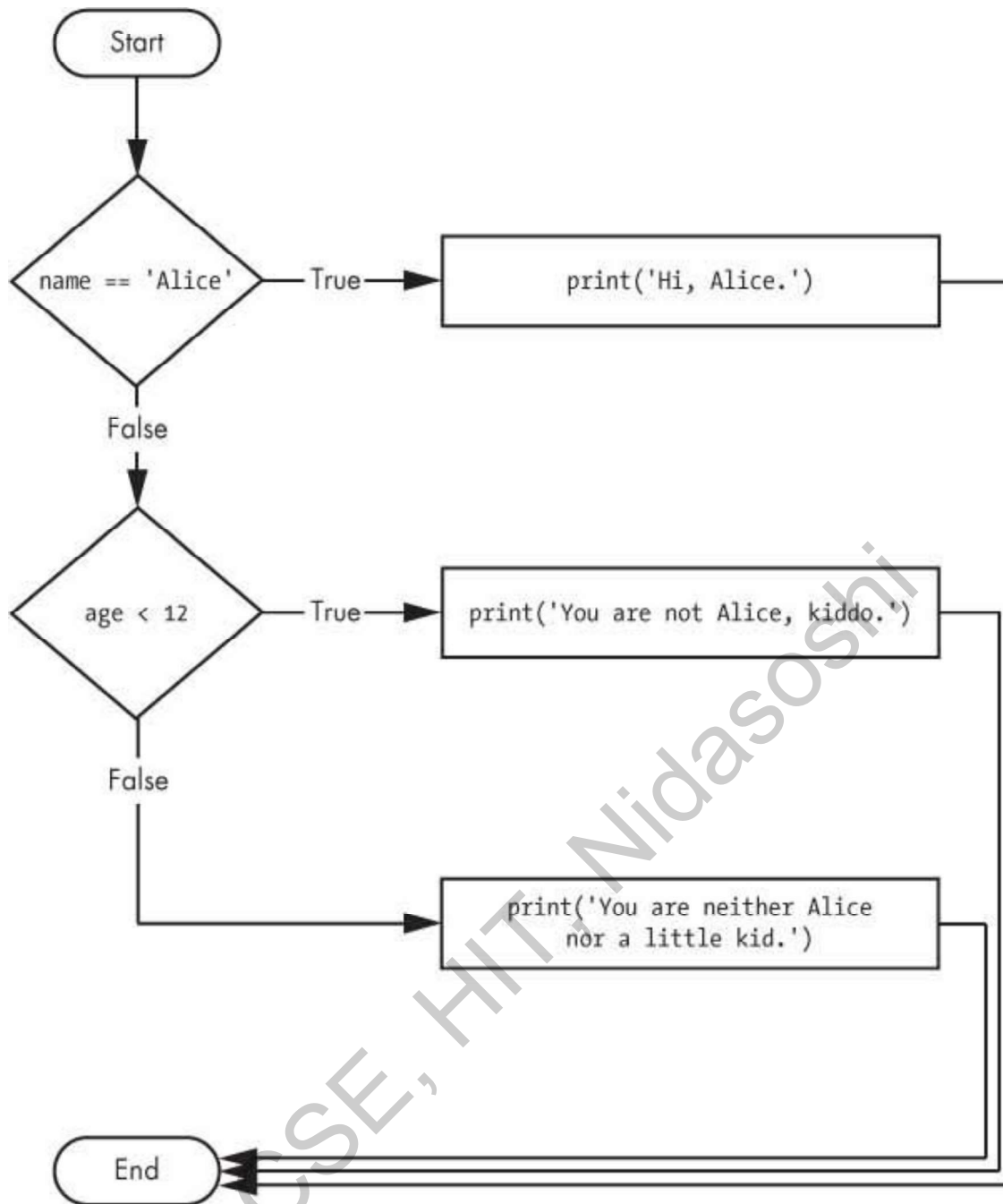


Figure 2-7: Flowchart for the previous littleKid.py program

while Loop Statements

You can make a block of code execute over and over again using a while statement. The code in a while clause will be executed as long as the while statement's condition is True. In code, a while statement always consists of the following:

- The while keyword
- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the while clause)

You can see that a while statement looks similar to an if statement. The difference is in how they behave. At the end of an if clause, the program execution continues after the if statement. But at the end of a while clause, the program execution jumps back to the start of the while statement. The while clause is often called the *while loop* or just the *loop*.

Let's look at an if statement and a while loop that use the same condition and take the same actions based on that condition. Here is the code with an if statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a while statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar—both if and while check the value of spam, and if it's less than 5, they print a message. But when you run these two code snippets, something very different happens for each one. For the if statement, the output is simply "Hello, world.". But for the while statement, it's "Hello, world." repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 2-8 and 2-9, to see why this happens.

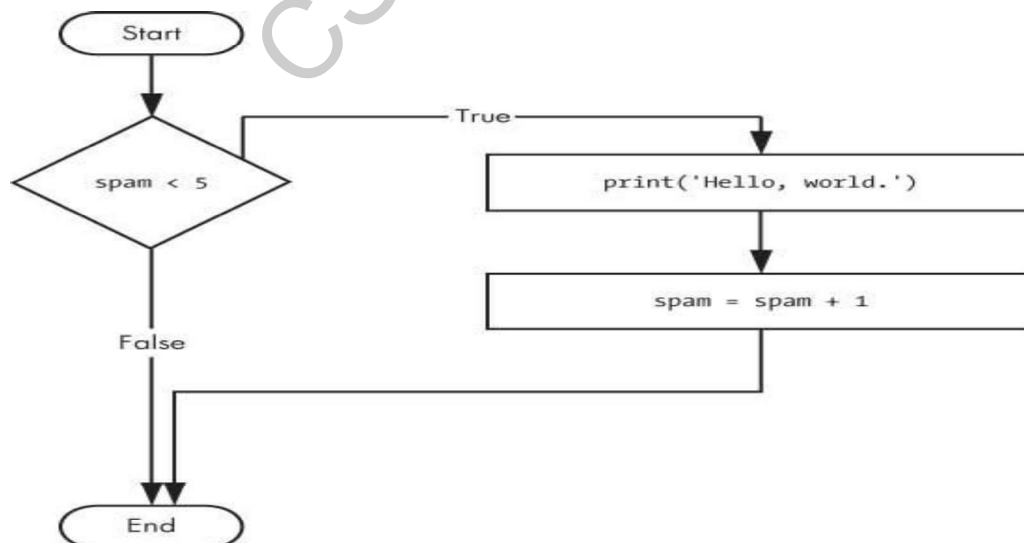


Figure 2-8: The flowchart for the if statement code

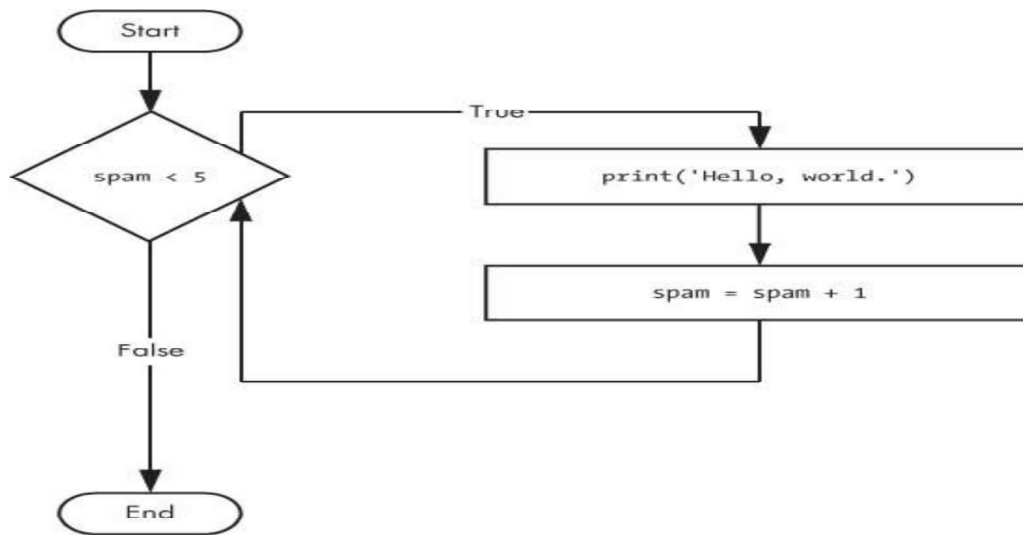


Figure 2-9: The flowchart for the while statement code

The code with the if statement checks the condition, and it prints Hello, world. only once if that condition is true. The code with the while loop, on the other hand, will print it five times. The loop stops after five prints because the integer in spam increases by one at the end of each loop iteration, which means that the loop will execute five times before spam < 5 is False.

In the while loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is True, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be False, the while clause is skipped.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, your name. Select **File** ▶ **New** to open a new file editor window, enter the following code, and save the file as *yourName.py*:

```

❶ name = ""
❷ while name != 'your name':
    print('Please type your name.')
❸ name = input()
❹ print('Thank you!')
```

You can view the execution of this program at <https://autbor.com/yourname/>. First, the program sets the name variable ❶ to an empty string. This is so that the name != 'your name' condition will evaluate to True and the program execution will enter the while loop's clause ❷.

The code inside this clause asks the user to type their name, which is assigned to the name variable ③. Since this is the last line of the block, the execution moves back to the start of the while loop and reevaluates the condition. If the value in name is *not equal* to the string 'your name', then the condition is True, and the execution enters the while clause again.

But once the user types your name, the condition of the while loop will be 'your name' != 'your name', which evaluates to False. The condition is now False, and instead of the program execution reentering the while loop's clause, Python skips past it and continues running the rest of the program ④. Figure 2-10 shows a flowchart for the *yourName.py* program.

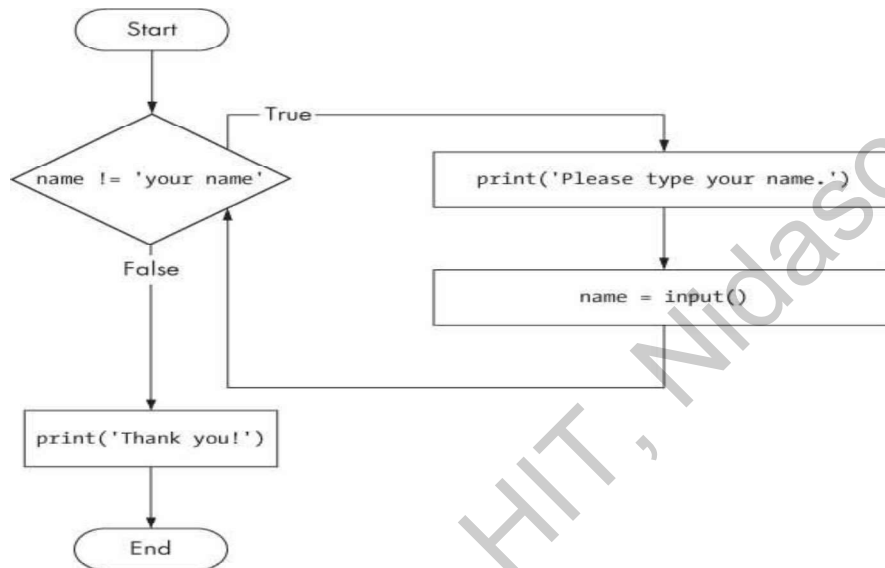


Figure 2-10: A flowchart of the *yourName.py* program

Now, let's see *yourName.py* in action. Press **F5** to run it, and enter something other than your name a few times before you give the program what it wants.

Please type your name.

Al

Please type your name.

Albert

Please type your name.

%#@#%*(^&!!!

Please type your name.

your name

Thank you!

If you never enter your name, then the while loop's condition will never be False, and the program will just keep asking forever. Here, the `input()` call lets the user enter the right string to make the program move on. In other programs, the condition might never actually change, and that can be a problem. Let's look at how you can break out of a while loop.

break Statements

There is a shortcut to getting the program execution to break out of a while loop's clause early. If the execution reaches a break statement, it immediately exits the while loop's clause. In code, a break statement simply contains the break keyword.

Pretty simple, right? Here's a program that does the same thing as the previous program, but it uses a break statement to escape the loop. Enter the following code, and save the file as `yourName2.py`:

```
❶ while True:
    print('Please type your name.')
    ❷ name = input()
    ❸ if name == 'your name':
        ❹ break
    ❺ print('Thank you!')
```

You can view the execution of this program at <https://autbor.com/yourname2/>. The first line ❶ creates an *infinite loop*; it is a while loop whose condition is always True. (The expression True, after all, always evaluates down to the value True.) After the program execution enters this loop, it will exit the loop only when a break statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to enter your name ❷. Now, however, while the execution is still inside the while loop, an if statement checks ❸ whether name is equal to 'your name'. If this condition is True, the break statement is run ❹, and the execution moves out of the loop to print("Thank you!") ❺. Otherwise, the if statement's clause that contains the break statement is skipped, which puts the execution at the end of the while loop. At this point, the program execution jumps back to the start of the while statement ❶ to recheck the condition. Since this condition is merely the True Boolean value, the execution enters the loop to ask the user to type your name again. See Figure 2-11 for this program's flowchart.

Run *yourName2.py*, and enter the same text you entered for *yourName.py*. The rewritten program should respond in the same way as the original.

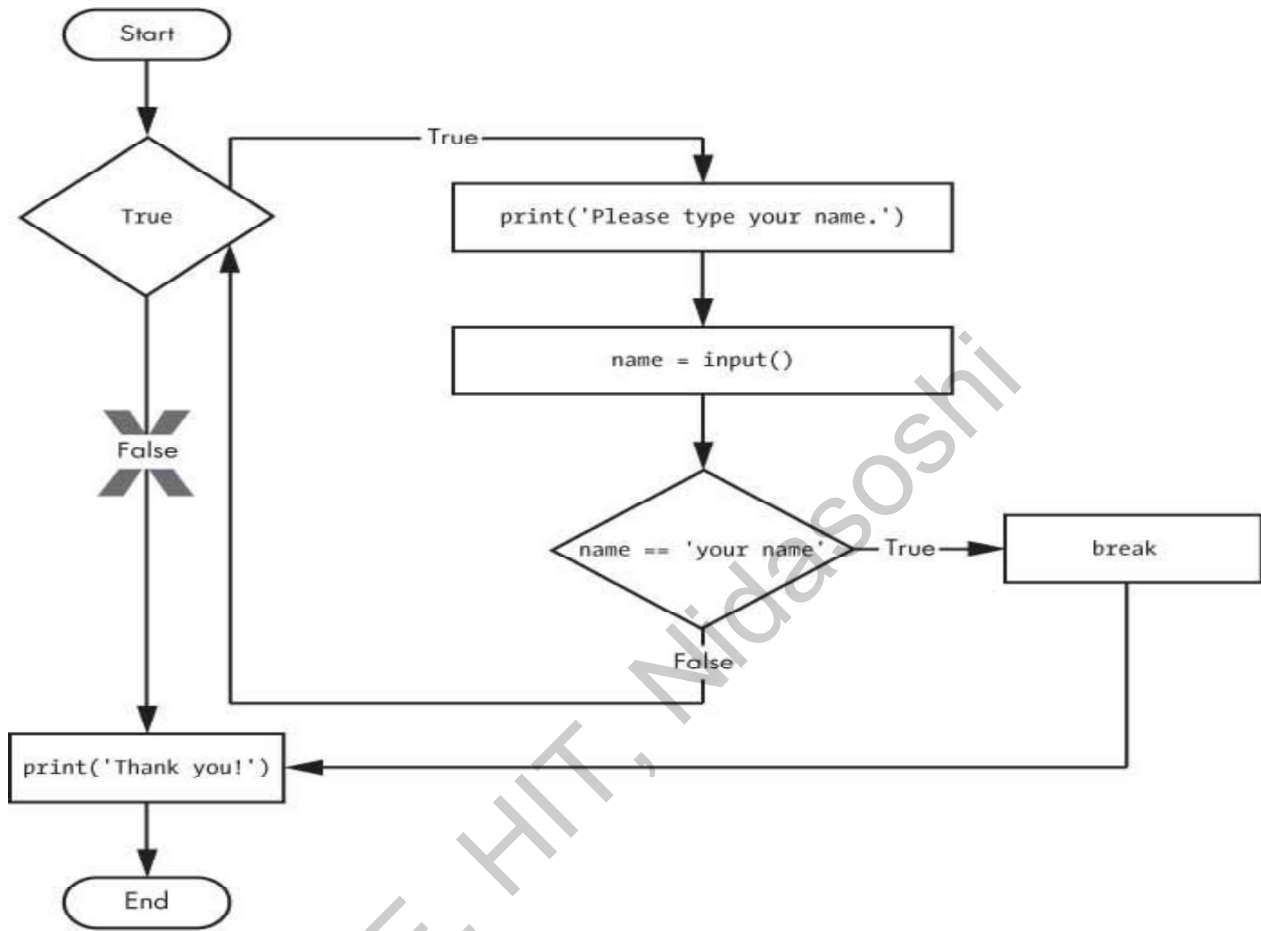


Figure 2-11: The flowchart for the *yourName2.py* program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always True.

***continue* Statements**

Like `break` statements, `continue` statements are used inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

Let's use `continue` to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as *swordfish.py*.

TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C or select **Shell ▶ Restart Shell** from IDLE's menu. This will send a KeyboardInterrupt error to your

program and cause it to stop immediately. Try stopping a program by creating a simple infinite loop in the file editor, and save the program as *infiniteLoop.py*.

```
while True:  
    print('Hello, world!')
```

When you run this program, it will print Hello, world! to the screen forever because the while statement's condition is always True. CTRL-C is also handy if you want to simply terminate your program immediately, even if it's not stuck in an infinite loop.

```
while True:  
    print('Who are you?')  
    name = input()  
    ❶ if name != 'Joe':  
        ❷ continue  
    print('Hello, Joe. What is the password? (It is a fish.)')  
    ❸ password = input()  
    if password == 'swordfish':  
        ❹ break  
    ❺ print('Access granted.')
```

If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop. When the program reevaluates the condition, the execution will always enter the loop, since the condition is simply the value True.

Once the user makes it past that if statement, they are asked for a password ③. If the password entered is swordfish, then the break statement ④ is run, and the execution jumps out of the while loop to print Access granted ⑤. Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop. See Figure 2-12 for this program's flowchart.

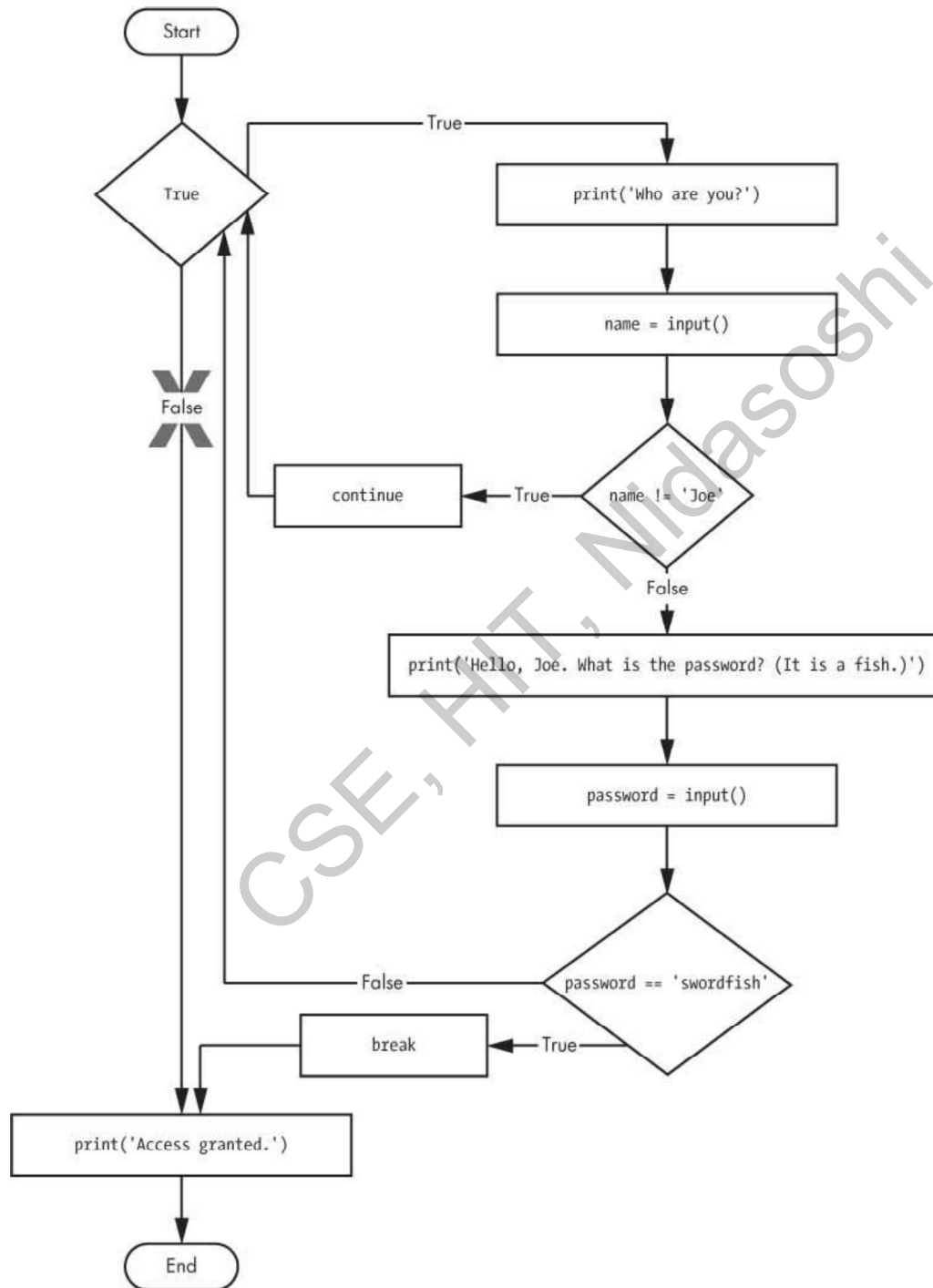


Figure 2-12: A flowchart for swordfish.py. The X path will logically never happen, because the loop condition is always True.

“TRUTHY” AND “FALSEY” VALUES

Conditions will consider some values in other data types equivalent to True and False. When used in conditions, 0, 0.0, and "" (the empty string) are considered False, while all other values are considered True. For example, look at the following program:

```
name = ""  
❶ while not name:  
    print('Enter your name:')  
    name = input()  
print('How many guests will you have?')  
-----  
numOfGuests = int(input())  
❷ if numOfGuests:  
    ❸ print('Be sure to have enough room for all your guests.')  
print('Done')
```

You can view the execution of this program at <https://autbor.com/howmanyguests/>. If the user enters a blank string for name, then the while statement's condition will be True ❶, and the program continues to ask for a name. If the value for numOfGuests is not 0 ❷, then the condition is considered to be True, and the program will print a reminder for the user ❸.

You could have entered `not name != ""` instead of `not name`, and `numOfGuests != 0` instead of `numOfGuests`, but using the truthy and falsey values can make your code easier to read.

Run this program and give it some input. Until you claim to be Joe, the program shouldn't ask for a password, and once you enter the correct password, it should exit.

Who are you?

I'm fine, thanks. Who are you?

Who are you?

Joe

Hello, Joe. What is the password? (It is a fish.)

Mary

Who are you?

Joe

Hello, Joe. What is the password? (It is a fish.)

swordfish

Access granted.

You can view the execution of this program at <https://autbor.com/hellojoe/>.

for Loops and the range() Function

The while loop keeps looping while its condition is True (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a for loop statement and the range() function.

In code, a for statement looks something like `for i in range(5):` and includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause)

Let's create a new program called *fiveTimes.py* to help you see a for loop in action.

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

You can view the execution of this program at <https://autbor.com/fivetimesfor/>. The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to 0. The `print()` call in the clause will print Jimmy Five Times (0). After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one.

This is why `range(5)` results in five iterations through the clause, with `i` being set to 0, then 1, then 2, then 3, and then 4. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 2-13 shows a flowchart for the `fiveTimes.py` program.

When you run this program, it should print Jimmy Five Times followed by the value of `i` five times before leaving the for loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

NOTE

You can use `break` and `continue` statements inside for loops as well. The `continue` statement will continue to the next value of the for loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use `continue` and `break` statements only inside `while` and `for` loops. If you try to use these statements elsewhere, Python will give you an error.

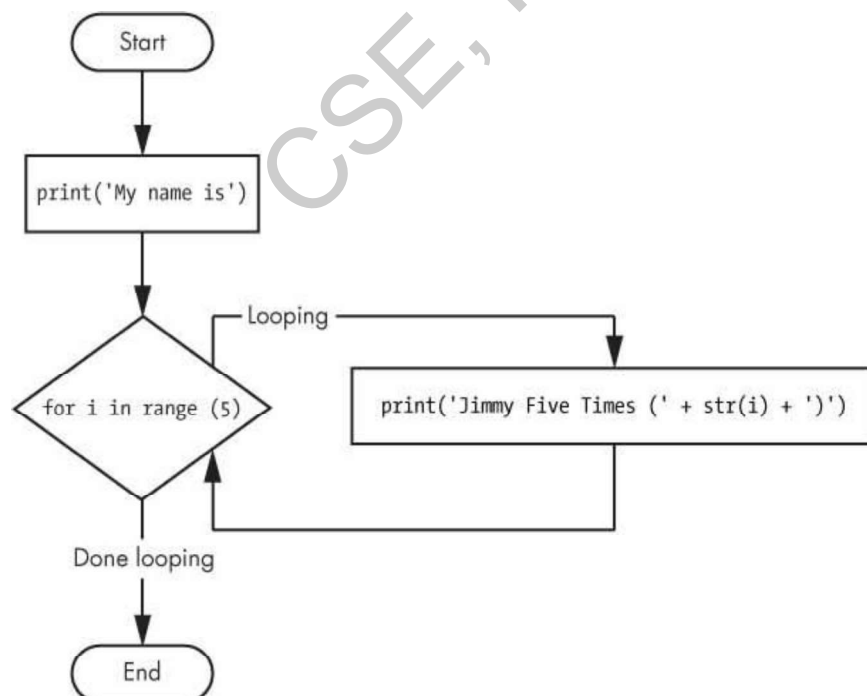


Figure 2-13: The flowchart for `fiveTimes.py`

As another for loop example, consider this story about the mathematician Carl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a for loop to do this calculation for you.

```
❶ total = 0
❷ for num in range(101):
    ❸ total = total + num
❹ print(total)
```

The result should be 5,050. When the program first starts, the total variable is set to 0. The for loop then executes `total = total + num` 100 times. By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to total. At this point, total is printed to the screen. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out a way to solve the problem in seconds. There are 50 pairs of numbers that add up to 101: 1 + 100, 2 + 99, 3 + 98, and so on, until 50 + 51. Since 50×101 is 5,050, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

An Equivalent while Loop

You can actually use a while loop to do the same thing as a for loop; for loops are just more concise. Let's rewrite *fiveTimes.py* to use a while loop equivalent of a for loop.

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

You can view the execution of this program at <https://autbor.com/fivetimeswhile/>. If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a for loop.

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence

of integers, including starting at a number other than zero.

```
for i in range(12, 16):  
    print(i)
```

The first argument will be where the for loop's variable starts, and the second argument will be up to, but not including, the number to stop at.

```
12  
13  
14  
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):  
    print(i)
```

So calling `range(0, 10, 2)` will count from zero to eight by intervals of two.

```
0  
2  
4  
6  
8
```

The `range()` function is flexible in the sequence of numbers it produces for for loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the for loop count down instead of up.

```
for i in range(5, -1, -1):  
    print(i)
```

This for loop would have the following output:

4
3
2
1
0

Running a for loop to print `i` with `range(5, -1, -1)` should print from five down to zero.

IMPORTING MODULES

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number-related functions, and so on.

Before you can use the functions in a module, you must import the module with an import statement. In code, an import statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

Enter this code into the file editor, and save it as *printRandom.py*:

DON'T OVERWRITE MODULE NAMES

When you save your Python scripts, take care not to give them a name that is used by one of Python's modules, such as *random.py*, *sys.py*, *os.py*, or *math.py*. If you accidentally name one of your programs, say, *random.py*, and use an `import random` statement in another program, your program would import your *random.py* file instead of Python's `random` module. This can lead to errors such as `AttributeError: module 'random' has no attribute 'randint'`, since your *random.py* doesn't have the functions that the real `random` module has. Don't use the names of any built-in Python functions either, such as `print()` or `input()`.

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

You can view the execution of this program at <https://autbor.com/printrandom/>. The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Since `randint()` is in the `random` module, you must first type **random.** in front of the function name to tell Python to look for this function inside the `random` module.

Here's an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. We'll learn more about them later in the book.

from import Statements

An alternative form of the import statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star; for example, `from random import *`.

With this form of import statement, calls to functions in `random` will not need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the `import random` form of the statement.

ENDING A PROGRAM EARLY WITH THE `SYS.EXIT()` FUNCTION

The last flow control concept to cover is how to terminate the program. Programs always terminate if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or exit, before the last instruction by calling the `sys.exit()` function. Since this function is in the `sys` module, you have to import `sys` before your program can use it.

Open a file editor window and enter the following code, saving it as *exitExample.py*:

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in IDLE. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the execution reaches the `sys.exit()` call. When `response` is equal to `exit`, the line containing the `sys.exit()` call is executed. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

A SHORT PROGRAM: GUESS THE NUMBER

The examples I've shown you so far are useful for introducing basic concepts, but now let's see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple "guess the number" game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

Enter the following source code into the file editor, and save the file as *guessTheNumber.py*:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + '
guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

You can view the execution of this program at <https://autbor.com/guessthenumber/>.
Let's look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the `random` module so that it can use the `random.randint()` function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable `secretNumber`.

```
print('I am thinking of a number between 1 and 20.')
```

```
# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
```

```
print('Take a guess.')
guess = int(input())
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code that lets the player enter a guess and checks that guess is in a for loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Since `input()` returns a string, its return value is passed straight into `int()`, which translates the string into an integer value. This gets stored in a variable named `guess`.

```
if guess < secretNumber:
    print('Your guess is too low.')
elif guess > secretNumber:
    print('Your guess is too high.')
```

These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
else:
    break # This condition is the correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number—in which case, you want the program execution to break out of the for loop.

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

After the for loop, the previous `if...else` statement checks whether the player has correctly guessed the number and then prints an appropriate message to the screen. In both cases, the program displays a variable that contains an integer value (`guessesTaken` and `secretNumber`). Since it must concatenate these integer values to strings, it passes these variables to the `str()` function, which returns the string value form of these integers. Now these strings can be concatenated with the `+` operators before finally being passed to the `print()` function call.

A SHORT PROGRAM: ROCK, PAPER, SCISSORS

Let's use the programming concepts we've learned so far to create a simple rock, paper, scissors game. The output will look like this:

```
ROCK, PAPER, SCISSORS
0 Wins, 0 Losses, 0 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
p
PAPER versus...
PAPER
It is a tie!
0 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
s
SCISSORS versus...
PAPER
You win!
1 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
q
```

Type the following source code into the file editor, and save the file as *rpsGame.py*:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0

while True: # The main game loop.
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True: # The player input loop.
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        playerMove = input()
        if playerMove == 'q':
            sys.exit() # Quit the program.
```

```

if playerMove == 'r' or playerMove == 'p' or playerMove == 's':
    break # Break out of the player input loop.
print('Type one of r, p, s, or q.')

# Display what the player chose:
if playerMove == 'r':
    print('ROCK versus...')
elif playerMove == 'p':
    print('PAPER versus...')
elif playerMove == 's':
    print('SCISSORS versus...')

# Display what the computer chose:
randomNumber = random.randint(1, 3)
if randomNumber == 1:
    computerMove = 'r'
    print('ROCK')
elif randomNumber == 2:
    computerMove = 'p'
    print('PAPER')
elif randomNumber == 3:
    computerMove = 's'
    print('SCISSORS')

# Display and record the win/loss/tie:
if playerMove == computerMove:
    print('It is a tie!')
    ties = ties + 1
elif playerMove == 'r' and computerMove == 's':
    print('You win!')
    wins = wins + 1

elif playerMove == 'p' and computerMove == 'r':
    print('You win!')
    wins = wins + 1
elif playerMove == 's' and computerMove == 'p':

```

```
print('You win!')
wins = wins + 1
elif playerMove == 'r' and computerMove == 'p':
    print('You lose!')
    losses = losses + 1
elif playerMove == 'p' and computerMove == 's':
    print('You lose!')
    losses = losses + 1
elif playerMove == 's' and computerMove == 'r':
    print('You lose!')
    losses = losses + 1
```

Let's look at this code line by line, starting at the top.

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0
```

First, we import the `random` and `sys` module so that our program can call the `random.randint()` and `sys.exit()` functions. We also set up three variables to keep track of how many wins, losses, and ties the player has had.

```
while True: # The main game loop.
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True: # The player input loop.
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        playerMove = input()
        if playerMove == 'q':
            sys.exit() # Quit the program.
        if playerMove == 'r' or playerMove == 'p' or playerMove == 's':
            break # Break out of the player input loop.
        print('Type one of r, p, s, or q.')
```

This program uses a while loop inside of another while loop. The first loop is the main game loop, and a single game of rock, paper, scissors is played on each iteration through this loop. The second loop asks for input from the player, and keeps looping until the

player has entered an r, p, s, or q for their move. The r, p, and s correspond to rock, paper, and scissors, respectively, while the q means the player intends to quit. In that case, `sys.exit()` is called and the program exits. If the player has entered r, p, or s, the execution breaks out of the loop. Otherwise, the program reminds the player to enter r, p, s, or q and goes back to the start of the loop.

```
# Display what the player chose:
if playerMove == 'r':
    print('ROCK versus...')
elif playerMove == 'p':
    print('PAPER versus...')
elif playerMove == 's':
    print('SCISSORS versus...')
```

The player's move is displayed on the screen.

```
# Display what the computer chose:
randomNumber = random.randint(1, 3)
if randomNumber == 1:
    computerMove = 'r'
    print('ROCK')
elif randomNumber == 2:
    computerMove = 'p'
    print('PAPER')
elif randomNumber == 3:
    computerMove = 's'
    print('SCISSORS')
```

Next, the computer's move is randomly selected. Since `random.randint()` can only return a random number, the 1, 2, or 3 integer value it returns is stored in a variable named `randomNumber`. The program stores a 'r', 'p', or 's' string in `computerMove` based on the integer in `randomNumber`, as well as displays the computer's move.

```
# Display and record the win/loss/tie:
if playerMove == computerMove:
    print('It is a tie!')
    ties = ties + 1
elif playerMove == 'r' and computerMove == 's':
    print('You win!')
```

```
wins = wins + 1
elif playerMove == 'p' and computerMove == 'r':
    print('You win!')
    wins = wins + 1
elif playerMove == 's' and computerMove == 'p':
    print('You win!')
    wins = wins + 1
elif playerMove == 'r' and computerMove == 'p':
    print('You lose!')
    losses = losses + 1
elif playerMove == 'p' and computerMove == 's':
    print('You lose!')
    losses = losses + 1
elif playerMove == 's' and computerMove == 'r':
    print('You lose!')
    losses = losses + 1
```

Finally, the program compares the strings in `playerMove` and `computerMove`, and displays the results on the screen. It also increments the `wins`, `losses`, or `ties` variable appropriately. Once the execution reaches the end, it jumps back to the start of the main program loop to begin another game.

SUMMARY

By using expressions that evaluate to `True` or `False` (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to `True`. The `break` and `continue` statements are useful if you need to exit a loop or jump back to the loop's start.

These flow control statements will let you write more intelligent programs. You can also use another type of flow control by writing your own functions, which is the topic of the next chapter.

PRACTICE QUESTIONS

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
4. What do the following expressions evaluate to?

`(5 > 4) and (3 == 5)`

`not (5 > 4)`

`(5 > 4) or (3 == 5)`

`not ((5 > 4) or (3 == 5))`

`(True and True) and (True == False)`

`(not False) or (not True)`

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
    print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
    print('spam')
print('spam')
```

Write code that prints Hello if 1 is stored in spam, prints Howdy if 2 is stored in spam, and prints Greetings! if anything else is stored in spam.

9. What keys can you press if your program is stuck in an infinite loop?
10. What is the difference between `break` and `continue`?
11. What is the difference between `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` in a `for` loop?
12. Write a short program that prints the numbers 1 to 10 using a `for` loop. Then write an equivalent program that prints the numbers 1 to 10 using a `while` loop.
13. If you had a function named `bacon()` inside a module named `spam`, how would you call it after importing `spam`?

FUNCTIONS

You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a miniprogram within a program.

To better understand how functions work, let's create one. Enter this program into the file editor and save it as *helloFunc.py*:

```
❶ def hello():
    ❷ print('Howdy!')
    print('Howdy!!!')

    print('Hello there.')
```



```
❸ hello()
hello()
hello()
```

You can view the execution of this program at <https://autbor.com/hellofunc/>. The first line is a `def` statement ❶, which defines a function named `hello()`. The code in the block that follows the `def` statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `hello()` lines after the function ❸ are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
```

```
print('Howdy!')
print('Howdy!!!')
print('Hellothere.')
print('Howdy!')
print('Howdy!!!')
print('Hellothere.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

In general, you always want to avoid duplicating code because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

As you get more programming experience, you'll often find yourself *deduplicating* code, which means getting rid of duplicated or copy-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

DEF STATEMENTS WITH PARAMETERS

When you call the `print()` or `len()` function, you pass them values, called *arguments*, by typing them between the parentheses. You can also define your own functions that accept arguments. Type this example into the file editor and save it as *helloFunc2.py*:

```
❶ def hello(name):
    ❷ print('Hello, ' + name)

❸ hello('Alice')
hello('Bob')
```

When you run this program, the output looks like this:

```
Hello, Alice
Hello, Bob
```

You can view the execution of this program at <https://autbor.com/hellofunc2/>. The definition of the `hello()` function in this program has a parameter called `name` ❶. *Parameters* are variables that contain arguments. When a function is called with arguments, the arguments are stored in the parameters. The first time the `hello()` function is called, it is passed the argument 'Alice' ❸. The program execution enters the function,

and the parameter name is automatically set to 'Alice', which is what gets printed by the `print()` statement ❷.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`. This variable is destroyed after the function call `hello('Bob')` returns, so `print(name)` would refer to a `name` variable that does not exist.

This is similar to how a program's variables are forgotten when the program terminates. I'll talk more about why that happens later in the chapter, when I discuss what a function's local scope is.

Define, Call, Pass, Argument, Parameter

The terms *define*, *call*, *pass*, *argument*, and *parameter* can be confusing. Let's look at a code example to review these terms:

```
❶ def sayHello(name):  
    print('Hello, ' + name)  
❷ sayHello('Al')
```

To *define* a function is to create it, just like an assignment statement like `spam = 42` creates the `spam` variable. The `def` statement defines the `sayHello()` function ❶. The `sayHello('Al')` line ❷ *calls* the now-created function, sending the execution to the top of the function's code. This function call is also known as *passing* the string value 'Al' to the function. A value being passed to a function in a function call is an *argument*. The argument 'Al' is assigned to a local variable named `name`. Variables that have arguments assigned to them are *parameters*.

It's easy to mix up these terms, but keeping them straight will ensure that you know precisely what the text in this chapter means.

RETURN VALUES AND RETURN STATEMENTS

When you call the `len()` function and pass it an argument such as 'Hello', the function call evaluates to the integer value 5, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the *return value* of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The `return` keyword
- The value or expression that the function should return

When an expression is used with a return statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what number it is passed as an argument. Enter this code into the file editor and save it as *magic8Ball.py*:

```
❶ import random

❷ def getAnswer(answerNumber):
    ❸ if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

You can view the execution of this program at <https://autbor.com/magic8ball/>. When this program starts, Python first imports the random module ❶. Then the getAnswer() function is defined ❷. Because the function is being defined (and not called), the execution skips over the code in it. Next, the random.randint() function is called with two arguments: 1 and 9 ❹. It evaluates to a random integer between 1 and 9 (including 1 and 9 themselves), and this value is stored in a variable named r.

The getAnswer() function is called with r as the argument ❺. The program execution moves to the top of the getAnswer() function ❸, and the value r is stored in a parameter named answerNumber. Then, depending on the value in answerNumber, the function returns

one of many possible string values. The program execution returns to the line at the bottom of the program that originally called `getAnswer()` ⑤. The returned string is assigned to a variable named `fortune`, which then gets passed to a `print()` call ⑥ and is printed to the screen.

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

to this single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

Remember, expressions are composed of values and operators. A function call can be used in an expression because the call evaluates to its return value.

THE NONE VALUE

In Python, there is a value called `None`, which represents the absence of a value. The `None` value is the only value of the `NoneType` data type. (Other programming languages might call this value `null`, `nil`, or `undefined`.) Just like the Boolean `True` and `False` values, `None` must be typed with a capital *N*.

This value-without-a-value can be helpful when you need to store something that won't be confused for a real value in a variable. One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, but it doesn't need to return anything in the same way `len()` or `input()` does. But since all function calls need to evaluate to a return value, `print()` returns `None`. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds `return None` to the end of any function definition with no `return` statement. This is similar to how a `while` or `for` loop implicitly ends with a `continue` statement. Also, if you use a `return` statement without a value (that is, just the `return` keyword by itself), then `None` is returned.

KEYWORD ARGUMENTS AND THE PRINT() FUNCTION

Most arguments are identified by their position in the function call. For example, `random.randint(1, 10)` is different from `random.randint(10, 1)`. The function call `random.randint(1, 10)` will return a random integer between 1 and 10 because the first argument is the low end of the range and the second argument is the high end (while `random.randint(10, 1)` causes an error).

However, rather than through their position, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for *optional parameters*. For example, the `print()` function has the optional parameters `end` and `sep` to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

If you ran a program with the following code:

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two outputted strings appear on separate lines because the `print()` function automatically adds a newline character to the end of the string it is passed. However, you can set the `end` keyword argument to change the newline character to a different string. For example, if the code were this:

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output is printed on a single line because there is no longer a newline printed after 'Hello'. Instead, the blank string is printed. This is useful if you need to disable the newline that gets added to the end of every `print()` function call.

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
```

```
cats dogs mice
```

But you could replace the default separating string by passing the `sep` keyword argument a different string. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
```

```
cats,dogs,mice
```

You can add keyword arguments to the functions you write as well, but first you'll have to learn about the list and dictionary data types in the next two chapters. For now, just know that some functions have optional keyword arguments that can be specified when the function is called.

THE CALL STACK

Imagine that you have a meandering conversation with someone. You talk about your friend Alice, which then reminds you of a story about your coworker Bob, but first you have to explain something about your cousin Carol. You finish your story about Carol and go back to talking about Bob, and when you finish your story about Bob, you go back to talking about Alice. But then you are reminded about your brother David, so you tell a story about him, and then get back to finishing your original story about Alice. Your conversation followed a *stack*-like structure, like in Figure 3-1. The conversation is stack-like because the current topic is always at the top of the stack.

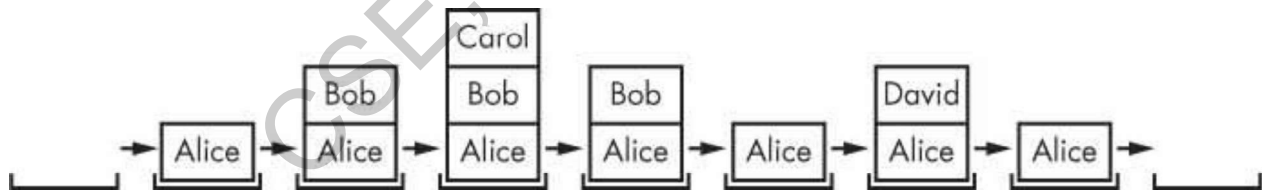


Figure 3-1: Your meandering conversation stack

Similar to our meandering conversation, calling a function doesn't send the execution on a one-way trip to the top of a function. Python will remember which line of code called the function so that the execution can return there when it encounters a `return` statement. If that original function called other functions, the execution would return to *those* function calls first, before returning from the original function call.

Open a file editor window and enter the following code, saving it as *abcdCallStack.py*:

```
def a():  
    print('a() starts')
```


❶ b()

❷ d()

print('a() returns')

def b():

print('b() starts')

❸ c()

print('b() returns')

def c():

❹ print('c() starts')

print('c() returns')

def d():

print('d() starts')

print('d() returns')

❺ a()

If you run this program, the output will look like this:

a() starts

b() starts

c() starts

c() returns

b() returns

d() starts

d() returns

a() returns

You can view the execution of this program at <https://autbor.com/abcdcallstack/>. When a() is called ❺, it calls b() ❶, which in turn calls c() ❸. The c() function doesn't call anything; it just displays c() starts ❹ and c() returns before returning to the line in b() that called it ❸. Once execution returns to the code in b() that called c(), it returns to the line in a() that called b() ❶. The execution continues to the next line in the b() function ❷, which is a call to d(). Like the c() function, the d() function also doesn't call anything. It just displays d() starts and d() returns before returning to the line in b() that called it. Since b() contains no other code, the execution returns to the line in a() that called b() ❷. The last line in a() displays a() returns before returning to the original a() call at the end of the program ❺.

The *call stack* is how Python remembers where to return the execution after each function call. The call stack isn't stored in a variable in your program; rather, Python handles it behind the scenes. When your program calls a function, Python creates a *frame object* on the top of the call stack. Frame objects store the line number of the original function call so that Python can remember where to return. If another function call is made, Python puts another frame object on the call stack above the other one.

When a function call returns, Python removes a frame object from the top of the stack and moves the execution to the line number stored in it. Note that frame objects are always added and removed from the top of the stack and not from any other place. Figure 3-2 illustrates the state of the call stack in *abcdCallStack.py* as each function is called and returns.

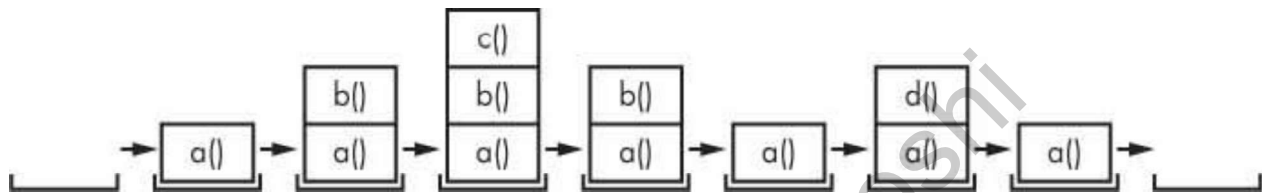


Figure 3-2: The frame objects of the call stack as *abcdCallStack.py* calls and returns from functions

The top of the call stack is which function the execution is currently in. When the call stack is empty, the execution is on a line outside of all functions.

The call stack is a technical detail that you don't strictly need to know about to write programs. It's enough to understand that function calls return to the line number they were called from. However, understanding call stacks makes it easier to understand local and global scopes, described in the next section.

LOCAL AND GLOBAL SCOPE

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call the function,

the local variables will not remember the values stored in them from the last time the function was called. Local variables are also stored in frame objects on the call stack.

Scopes matter for several reasons:

- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the number of lines of code that may be causing a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program, and your program could be hundreds or thousands of lines long! But if the bug is caused by a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():  
    ❶ eggs = 31337  
spam()  
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):  
  File "C:/test1.py", line 4, in <module>  
    print(eggs)  
NameError: name 'eggs' is not defined
```

The error happens because the `eggs` variable exists only in the local scope created when `spam()` is called ❶. Once the program execution returns from `spam`, that local scope is destroyed, and there is no longer a variable named `eggs`. So when your program tries to run `print(eggs)`, Python gives you an error saying that `eggs` is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
def spam():
    ❶ eggs = 99
    ❷ bacon()
    ❸ print(eggs)

def bacon():
    ham = 101
    ❹ eggs = 0

❺ spam()
```

You can view the execution of this program at <https://autbor.com/otherlocalscopes/>. When the program starts, the `spam()` function is called ❺, and a local scope is created. The local variable `eggs` ❶ is set to 99. Then the `bacon()` function is called ❷, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable `ham` is set to 101, and a local variable `eggs`—which is different from the one in `spam()`'s local scope—is also created ❹ and set to 0.

When `bacon()` returns, the local scope for that call is destroyed, including its `eggs` variable. The program execution continues in the `spam()` function to print the value of `eggs` ❸. Since the local scope for the call to `spam()` still exists, the only `eggs` variable is the `spam()` function's `eggs` variable, which was set to 99. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

You can view the execution of this program at <https://autbor.com/readglobal/>. Since there is no parameter named `eggs` or any code that assigns `eggs` a value in the `spam()` function, when `eggs` is used in `spam()`, Python considers it a reference to the global variable `eggs`. This is why 42 is printed when the previous program is run.

Local and Global Variables with the Same Name

Technically, it's perfectly acceptable to use the same variable name for a global variable and local variables in different scopes in Python. But, to simplify your life, avoid doing this. To see what happens, enter the following code into the file editor and save it as *localGlobalSameName.py*:

```
def spam():
    ❶ eggs = 'spam local'
    print(eggs) # prints 'spam local'

def bacon():
    ❷ eggs = 'bacon local'
    print(eggs) # prints 'bacon local'
    spam()
    print(eggs) # prints 'bacon local'

❸ eggs = 'global'
bacon()
print(eggs) # prints 'global'
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

You can view the execution of this program at <https://autbor.com/localglobalsamename/>. There are actually three different variables in

this program, but confusingly they are all named `eggs`. The variables are as follows:

- ❶ A variable named `eggs` that exists in a local scope when `spam()` is called.
- ❷ A variable named `eggs` that exists in a local scope when `bacon()` is called.
- ❸ A variable named `eggs` that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

THE GLOBAL STATEMENT

If you need to modify a global variable from within a function, use the `global` statement. If you have a line such as `global eggs` at the top of a function, it tells Python, “In this function, `eggs` refers to the global variable, so don’t create a local variable with this name.” For example, enter the following code into the file editor and save it as *globalStatement.py*:

```
def spam():  
    ❶ global eggs  
    ❷ eggs = 'spam'  
  
eggs = 'global'  
spam()  
print(eggs)
```

When you run this program, the final `print()` call will output this:

```
spam
```

You can view the execution of this program at <https://autbor.com/globalstatement/>. Because `eggs` is declared `global` at the top of `spam()` ❶, when `eggs` is set to `'spam'` ❷, this assignment is done to the globally scoped `eggs`. No local `eggs` variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a `global` statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

To get a better feel for these rules, here's an example program. Enter the following code into the file editor and save it as *sameNameLocalGlobal.py*:

```
def spam():
    ❶ global eggs
    eggs = 'spam' # this is the global

def bacon():
    ❷ eggs = 'bacon' # this is a local

def ham():
    ❸ print(eggs) # this is the global

eggs = 42 # this is the global
spam()
print(eggs)
```

In the `spam()` function, `eggs` is the global `eggs` variable because there's a global statement for `eggs` at the beginning of the function ❶. In `bacon()`, `eggs` is a local variable because there's an assignment statement for it in that function ❷. In `ham()` ❸, `eggs` is the global variable because there is no assignment statement or global statement for it in that function. If you run *sameNameLocalGlobal.py*, the output will look like this:

```
spam
```

You can view the execution of this program at <https://autbor.com/sameNameLocalGlobal/>. In a function, a variable will either always be global or always be local. The code in a function can't use a local variable named `eggs` and then use the global `eggs` variable later in that same function.

NOTE

If you ever want to modify the value stored in a global variable from in a function, you must use a global statement on that variable.

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, enter the following into the file editor and save it as *sameNameError.py*:

```
def spam():
    print(eggs) # ERROR!
    ❶ eggs = 'spam local'
```

```
    ❷ eggs = 'global'
    spam()
```

If you run the previous program, it produces an error message.

Traceback (most recent call last):

```
File "C:/sameNameError.py", line 6, in <module>
    spam()
File "C:/sameNameError.py", line 2, in spam
    print(eggs) # ERROR!
```

UnboundLocalError: local variable 'eggs' referenced before assignment

You can view the execution of this program at <https://autbor.com/sameNameError/>. This error happens because Python sees that there is an assignment statement for `eggs` in the `spam()` function ❶ and, therefore, considers `eggs` to be local. But because `print(eggs)` is executed before `eggs` is assigned anything, the local variable `eggs` doesn't exist. Python will *not* fall back to using the global `eggs` variable ❷.

FUNCTIONS AS “BLACK BOXES”

Often, all you need to know about a function are its inputs (the parameters) and output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating a function as a “black box.”

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

EXCEPTION HANDLING

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a divide-by-zero error. Open a file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called `spam`, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

You can view the execution of this program at <https://autbor.com/zerodivide/>. A `ZeroDivisionError` happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the `return` statement in `spam()` is causing an error.

Errors can be handled with `try` and `except` statements. The code that could potentially have an error is put in a `try` clause. The program execution moves to the start of a following `except` clause if an error happens.

You can put the previous divide-by-zero code in a `try` clause and have an `except` clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')
```

```
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

You can view the execution of this program at <https://autbor.com/tryexceptzerodivide/>. Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

You can view the execution of this program at <https://autbor.com/spamintry/>. The reason print(spam(1)) is never executed is because once the execution jumps to the code in the except clause, it does not return to the try clause. Instead, it just continues moving down the program as normal.

A SHORT PROGRAM: ZIGZAG

Let's use the programming concepts you've learned so far to create a small animation program. This program will create a back-and-forth, zigzag pattern until the user stops it by pressing the Mu editor's Stop button or by pressing CTRL-C. When you run this program, the output will look something like this:

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

Type the following source code into the file editor, and save the file as *zigzag.py*:

```
import time, sys
indent = 0 # How many spaces to indent.
indentIncreasing = True # Whether the indentation is increasing or not.

try:
    while True: # The main program loop.
        print(' ' * indent, end=" ")
        print('*****')
        time.sleep(0.1) # Pause for 1/10 of a second.

        if indentIncreasing:
            # Increase the number of spaces:
            indent = indent + 1
            if indent == 20:
                # Change direction:
                indentIncreasing = False

        else:
            # Decrease the number of spaces:
            indent = indent - 1
            if indent == 0:
                # Change direction:
```

```
    indentIncreasing = True
except KeyboardInterrupt:
    sys.exit()
```

Let's look at this code line by line, starting at the top.

```
import time, sys
indent = 0 # How many spaces to indent.
indentIncreasing = True # Whether the indentation is increasing or not.
```

First, we'll import the `time` and `sys` modules. Our program uses two variables: the `indent` variable keeps track of how many spaces of indentation are before the band of eight asterisks and `indentIncreasing` contains a Boolean value to determine if the amount of indentation is increasing or decreasing.

```
try:
    while True: # The main program loop.
        print(' ' * indent, end=" ")
        print('*****')
        time.sleep(0.1) # Pause for 1/10 of a second.
```

Next, we place the rest of the program inside a try statement. When the user presses CTRL-C while a Python program is running, Python raises the `KeyboardInterrupt` exception. If there is no try-except statement to catch this exception, the program crashes with an ugly error message. However, for our program, we want it to cleanly handle the `KeyboardInterrupt` exception by calling `sys.exit()`. (The code for this is in the `except` statement at the end of the program.)

The `while True:` infinite loop will repeat the instructions in our program forever. This involves using `' ' * indent` to print the correct amount of spaces of indentation. We don't want to automatically print a newline after these spaces, so we also pass `end=" "` to the first `print()` call. A second `print()` call prints the band of asterisks. The `time.sleep()` function hasn't been covered yet, but suffice it to say that it introduces a one-tenth-second pause in our program at this point.

```
    if indentIncreasing:
        # Increase the number of spaces:
        indent = indent + 1
    if indent == 20:
        indentIncreasing = False # Change direction.
```

Next, we want to adjust the amount of indentation for the next time we print asterisks. If `indentIncreasing` is `True`, then we want to add one to `indent`. But once `indent` reaches 20, we want the indentation to decrease.

```
else:
    # Decrease the number of spaces:
    indent = indent - 1
    if indent == 0:
        indentIncreasing = True # Change direction.
```

Meanwhile, if `indentIncreasing` was `False`, we want to subtract one from `indent`. Once `indent` reaches 0, we want the indentation to increase once again. Either way, the program execution will jump back to the start of the main program loop to print the asterisks again.

```
except KeyboardInterrupt:
    sys.exit()
```

If the user presses `CTRL-C` at any point that the program execution is in the `try` block, the `KeyboardInterrupt` exception is raised and handled by this `except` statement. The program execution moves inside the `except` block, which runs `sys.exit()` and quits the program. This way, even though the main program loop is an infinite loop, the user has a way to shut down the program.

SUMMARY

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions. This limits what code could be changing the values of your variables, which can be helpful when it comes to debugging your code.

Functions are a great tool to help you organize your code. You can think of them as black boxes: they have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

PRACTICE QUESTIONS

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?
5. How many global scopes are there in a Python program? How many local scopes?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a return statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?
10. What is the data type of None?
11. What does the `import realllyourpetsnamederic` statement do?
12. If you had a function named `bacon()` in a module named `spam`, how would you call it after importing `spam`?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the `try` clause? What goes in the `except` clause?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then write a program that lets the user type in an integer and that keeps calling `collatz()` on that number until the function returns the value 1. (Amazingly enough, this sequence actually works for any integer—sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called “the simplest impossible math problem.”)

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value.

Hint: An integer number is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

The output of this program could look something like this:

Enter number: /

3

10

5

16

8

4

2

1

CSE, HIT, Nidasoshi

LISTS

One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes writing programs that handle large amounts of data easier. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then I'll briefly cover the sequence data types (lists, tuples, and strings) and show how they compare with each other. In the next chapter, I'll introduce you to the dictionary data type.

THE LIST DATA TYPE

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`. Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable ❶ is still assigned only one value: the list value. But the list value itself contains other values. The value `[]` is an empty list that contains no values, similar to `"`, the empty string.

Getting Individual Values in a List with Indexes

Say you have the list ['cat', 'bat', 'rat', 'elephant'] stored in a variable named spam. The Python code spam[0] would evaluate to 'cat', and spam[1] would evaluate to 'bat', and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index 0, the second value is at index 1, the third value is at index 2, and so on. Figure 4-1 shows a list value assigned to spam, along with what the index expressions would evaluate to. Note that because the first index is 0, the last index is one less than the size of the list; a list of four items has 3 as its last index.

Figure 4-1: A list value stored in the variable spam, showing which value each index refers to

For example, enter the following expressions into the interactive shell. Start by assigning a list to the variable spam.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello, ' + spam[0]
❷ 'Hello, cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '!'
'The bat ate the cat.'
```

Notice that the expression 'Hello, ' + spam[0] ❶ evaluates to 'Hello, ' + 'cat' because spam[0] evaluates to the string 'cat'. This expression in turn evaluates to the string value 'Hello, cat' ❷.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a

TypeError error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[1]
```

```
'bat'
```

```
>>> spam[1.0]
```

```
Traceback (most recent call last):
```

```
File "<pyshell#13>", line 1, in <module>
```

```
    spam[1.0]
```

```
TypeError: list indices must be integers or slices, not float
```

```
>>> spam[int(1.0)]
```

```
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
```

```
>>> spam[0]
```

```
['cat', 'bat']
```

```
>>> spam[0][1]
```

```
'bat'
```

```
>>> spam[1][4]
```

```
50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints 'bat', the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at 0 and go up, you can also use negative integers for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[-1]
```

```
'elephant'
```

```
>>> spam[-3]
```

```
'bat'
```

```
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
```

```
'The elephant is afraid of the bat.'
```

Getting a List from Another List with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

`spam[2]` is a list with an index (one integer).

`spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[0:4]
```

```
['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[1:3]
```

```
['bat', 'rat']
```

```
>>> spam[0:-1]
```

```
['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[:2]
```

```
['cat', 'bat']
```

```
>>> spam[1:]
```

```
['bat', 'rat', 'elephant']
```

```
>>> spam[:]
```

```
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with the len() Function

The `len()` function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
```

```
>>> len(spam)
```

```
3
```

Changing Values in a List with Indexes

Normally, a variable name goes on the left side of an assignment statement, like `spam = 42`. However, you can also use an index of a list to change the value at that index. For example, `spam[1] = 'aardvark'` means “Assign the value at index 1 in the list `spam` to the string 'aardvark'.” Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[1] = 'aardvark'
```

```
>>> spam
```

```
['cat', 'aardvark', 'rat', 'elephant']
```

```
>>> spam[2] = spam[1]
```

```
>>> spam
```

```
['cat', 'aardvark', 'aardvark', 'elephant']
```

```
>>> spam[-1] = 12345
```

```
>>> spam
```

```
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

Lists can be concatenated and replicated just like strings. The `+` operator combines two lists to create a new list value and the `*` operator can be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
```

```
[1, 2, 3, 'A', 'B', 'C']
```

```
>>> ['X', 'Y', 'Z'] * 3
```

```
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
```

```
>>> spam = [1, 2, 3]
```

```
>>> spam = spam + ['A', 'B', 'C']
```

```
>>> spam
```

```
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The `del` statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the / interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

The `del` statement can also be used on a simple variable to delete it, as if it were an “unassignment” statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists. In practice, you almost never need to delete simple variables. The `del` statement is mostly used to delete values from lists.

WORKING WITH LISTS

When you first begin writing programs, it’s tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

It turns out that this is a bad way to write code. (Also, I don’t actually own this many cats, I swear.) For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
```

```

catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)

```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, enter the following source code and save it as *allMyCats2.py*:

```

catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
        ' (Or enter nothing to stop.):')
    name = input()
    if name == "":
        break
    catNames = catNames + [name] # list concatenation
print('The cat names are:')
for name in catNames:
    print(' ' + name)

```

When you run this program, the output will look something like this:

Enter the name of cat 1 (Or enter nothing to stop.):

Zophie

Enter the name of cat 2 (Or enter nothing to stop.):

Pooka

Enter the name of cat 3 (Or enter nothing to stop.):

Simon

Enter the name of cat 4 (Or enter nothing to stop.):

Lady Macbeth

Enter the name of cat 5 (Or enter nothing to stop.):

Fat-tail

Enter the name of cat 6 (Or enter nothing to stop.):

Miss Cleo

Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:

Zophie

Pooka

Simon

Lady Macbeth

Fat-tail

Miss Cleo

You can view the execution of these programs at <https://autbor.com/allmycats1/> and <https://autbor.com/allmycats2/>. The benefit of using a list is that your data is now in a structure, so your program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

In Chapter 2, you learned about using for loops to execute a block of code a certain number of times. Technically, a for loop repeats the code block once for each item in a list value. For example, if you ran this code:

```
for i in range(4):  
    print(i)
```

the output of this program would be as follows:

```
0  
1  
2  
3
```

This is because the return value from `range(4)` is a sequence value that Python considers similar to `[0, 1, 2, 3]`. (Sequences are described in “Sequence Data Types” on page 93.) The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:  
    print(i)
```

The previous for loop actually loops through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

A common Python technique is to use `range(len(someList))` with a for loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for i in range(len(supplies)):
...     print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

Index 0 in supplies is: pens

Index 1 in supplies is: staplers

Index 2 in supplies is: flamethrowers

Index 3 in supplies is: binders

Using `range(len(supplies))` in the previously shown for loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`). Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
```

True

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
```

```
>>> 'cat' in spam
```

False

```
>>> 'howdy' not in spam
```

False

```
>>> 'cat' not in spam
```

True

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as `myPets.py`:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
```

```
print('Enter a pet name:')
```

```
name = input()
```

```
if name not in myPets:
```

```
    print('I do not have a pet named ' + name)
```

```
else:
```



```
print(name + ' is my pet.')
```

The output may look something like this:

Enter a pet name:

Footfoot

I do not have a pet named Footfoot

You can view the execution of this program at <https://autbor.com/mypets/>.

The Multiple Assignment Trick

The *multiple assignment trick* (technically called *tuple unpacking*) is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a `ValueError`:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition, name = cat
```

Traceback (most recent call last):

```
File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
```

`ValueError: not enough values to unpack (expected 4, got 3)`

Using the enumerate() Function with Lists

Instead of using the `range(len(someList))` technique with a for loop to obtain the integer index of the items in the list, you can call the `enumerate()` function instead. On each iteration of the loop, `enumerate()` will return two values: the index of the item in the list, and the item in the list itself. For example, this code is equivalent to the code in the “Using for Loops with Lists” on page 84:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)
```

Index 0 in supplies is: pens

Index 1 in supplies is: staplers

Index 2 in supplies is: flamethrowers

Index 3 in supplies is: binders

The `enumerate()` function is useful if you need both the item and the item's index in the loop's block.

Using the `random.choice()` and `random.shuffle()` Functions with Lists

The `random` module has a couple functions that accept lists for arguments. The `random.choice()` function will return a randomly selected item from the list. Enter the following into the interactive shell:

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Dog'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
```

You can consider `random.choice(someList)` to be a shorter form of `someList[random.randint(0, len(someList) - 1)]`.

The `random.shuffle()` function will reorder the items in a list. This function modifies the list in place, rather than returning a new list. Enter the following into the interactive shell:

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)
>>> people
```

AUGMENTED ASSIGNMENT OPERATORS

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable `spam`, you would increase the value in `spam` by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator `+=` to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the `+`, `-`, `*`, `/`, and `%` operators, described in Table 4-1.

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement Equivalent assignment statement

<code>spam += 1</code>	<code>spam = spam + 1</code>
<code>spam -= 1</code>	<code>spam = spam - 1</code>
<code>spam *= 1</code>	<code>spam = spam * 1</code>
<code>spam /= 1</code>	<code>spam = spam / 1</code>
<code>spam %= 1</code>	<code>spam = spam % 1</code>

The `+=` operator can also do string and list concatenation, and the `*=` operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello,'
>>> spam += ' world!'
>>> spam
```

```
'Hello world!'
```

```
>>> bacon = ['Zophie']
```

```
>>> bacon *= 3
```

```
>>> bacon
```

```
['Zophie', 'Zophie', 'Zophie']
```

METHODS

A *method* is the same thing as a function, except it is “called on” a value. For example, if a list value were stored in `spam`, you would call the `index()` list method (which I’ll explain shortly) on that list like so: `spam.index('hello')`. The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index() Method

List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’t in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
```

```
>>> spam.index('hello')
```

```
0
```

```
>>> spam.index('heyas')
```

```
3
```

```
>>> spam.index('howdy howdy howdy')
```

```
Traceback (most recent call last):
```

```
File "<pyshell#31>", line 1, in <module>
```

```
    spam.index('howdy howdy howdy')
```

```
ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that `index()` returns 1,

not 3:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
```

```
>>> spam.index('Pooka')
```

```
1
```

Adding Values to Lists with the append() and insert() Methods

To add new values to a list, use the `append()` and `insert()` methods. Enter the following into

the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
```

```
['cat', 'dog', 'bat', 'moose']
```

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted.

Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
```

```
['cat', 'chicken', 'dog', 'bat']
```

Notice that the code is `spam.append('moose')` and `spam.insert(1, 'chicken')`, not `spam = spam.append('moose')` and `spam = spam.insert(1, 'chicken')`. Neither `append()` nor `insert()` gives the new value of `spam` as its return value. (In fact, the return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place*. Modifying a list in place is covered in more detail later in “Mutable and Immutable Data Types” on page 94.

Methods belong to a single data type. The `append()` and `insert()` methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the `AttributeError` error messages that show up:

```
>>> eggs = 'hello'
>>> eggs.append('world')
```

```
Traceback (most recent call last):
```

```
File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
```

```
AttributeError: 'str' object has no attribute 'append'
```

```
>>> bacon = 42
>>> bacon.insert(1, 'world')
```

```
Traceback (most recent call last):
```

```
File "<pyshell#22>", line 1, in <module>
```

```
    bacon.insert(1, 'world')
```

Removing Values from Lists with the remove() Method

The `remove()` method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error that is displayed:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The `del` statement is good to use when you know the index of the value you want to remove from the list. The `remove()` method is useful when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the `sort()` method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
```

```
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order. Enter the following into the interactive shell:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

There are three things you should note about the `sort()` method. First, the `sort()` method sorts the list in place; don't try to capture the return value by writing code like `spam = spam.sort()`.

Second, you cannot sort lists that have both number values *and* string values in them, since Python doesn't know how to compare these values. Enter the following into the interactive shell and notice the `TypeError` error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Third, `sort()` uses “ASCIIbetical order” rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the key keyword argument in the `sort()` method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Reversing the Values in a List with the `reverse()` Method

If you need to quickly reverse the order of the items in a list, you can call the `reverse()` list method. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()
>>> spam
['moose', 'dog', 'cat']
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines does not matter; Python knows that the list is not finished until it sees the ending square bracket. For example, you can have code that looks like this:

```
spam = ['apples',
        'oranges',
        'bananas',
        'cats']
print(spam)
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in the Magic 8 Ball program.

You can also split up a single instruction across multiple lines using the *line continuation character* at the end. Think of `\` as saying, "This instruction continues on the next line." The indentation on the line after a `\` line continuation is not significant. For example, the following is valid Python code:

```
print('Four score and seven ' + \
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

Like the `sort()` list method, `reverse()` doesn't return a list. This is why you write `spam.reverse()`, instead of `spam = spam.reverse()`.

EXAMPLE PROGRAM: MAGIC 8 BALL WITH A LIST

Using lists, you can write a much more elegant version of the previous chapter's Magic 8 Ball program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*.

```
import random

messages = ['It is certain',
'It is decidedly so', 'Yes
definitely',
'Reply hazy try again',
'Ask again later',
'Concentrate and ask again',
'My reply is no',
'Outlook not so good',
'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

You can view the execution of this program at <https://autbor.com/magic8ball2/>.

When you run this program, you'll see that it works the same as the previous *magic8Ball.py* program.

Notice the expression you use as the index for messages: `random.randint(0, len(messages) - 1)`. This produces a random number to use for the index, regardless of the size of messages. That is, you'll get a random number between 0 and the value of `len(messages) - 1`. The benefit of this approach is that you can easily add and remove strings to the messages list without changing other lines of code. If you later update your code, there will be fewer lines you have to change and fewer chances for you to introduce bugs.

SEQUENCE DATA TYPES

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar if you consider a string to be a "list" of single text characters. The Python sequence data types include lists, strings, range objects returned by `range()`, and tuples (explained in the "The Tuple Data Type" on page 96). Many of the things you can do with lists can also be done with strings and other values of sequence types: indexing; slicing; and using them with `for` loops, with `len()`, and with the `in` and `not in` operators. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
```

```

>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
...     print('* * * ' + i + ' * * *')

* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *

```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a *mutable* data type: it can have values added, removed, or changed. However, a string is *immutable*: it cannot be changed. Trying to reassign a single character in a string results in a `TypeError` error, as you can see by entering the following into the interactive shell:

```

>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment

```

The proper way to “mutate” a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string. Enter the following into the interactive shell:

```

>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name

```

```
'Zophie a cat'  
>>> newName  
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified, because strings are immutable.

Although a list value *is* mutable, the second line in the following code does not modify the list eggs:

```
>>> eggs = [1, 2, 3]  
>>> eggs = [4, 5, 6]  
>>> eggs  
[4, 5, 6]
```

The list value in `eggs` isn't being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]). This is depicted in Figure 4-2.

If you wanted to actually modify the original list in `eggs` to contain [4, 5, 6], you would have to do something like this:

CSE, HIT, Nidasochi

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

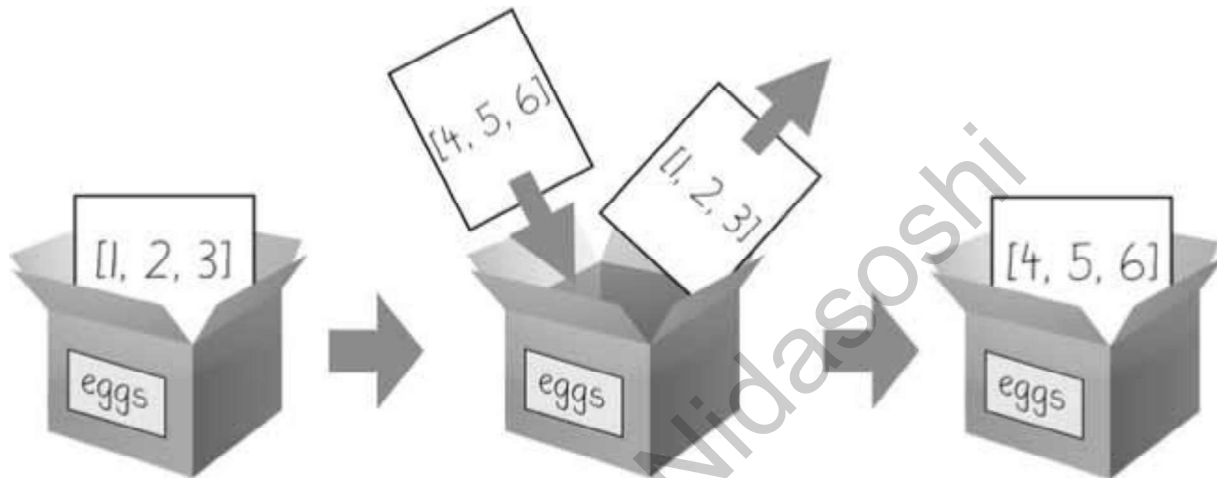


Figure 4-2: When `eggs = [4, 5, 6]` is executed, the contents of `eggs` are replaced with a new list value.

In the first example, the list value that `eggs` ends up with is the same list value it started with. It's just that this list has been changed, rather than overwritten. Figure 4-3 depicts the seven changes made by the first seven lines in the previous interactive shell example.

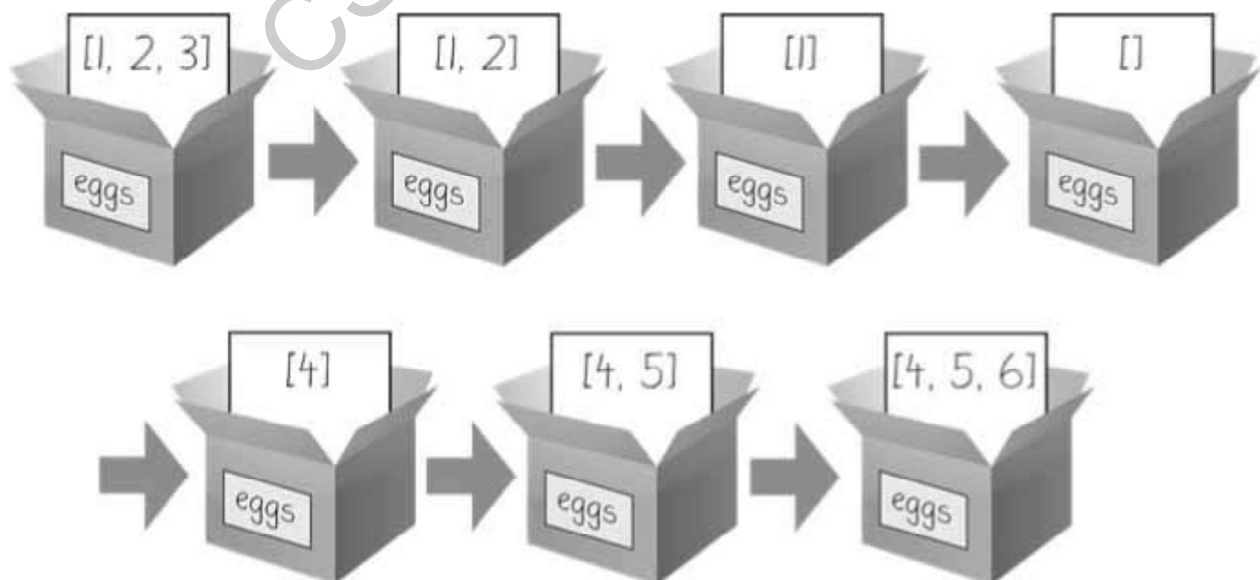


Figure 4-3: The `del` statement and the `append()` method modify the same list value in place.

Changing a value of a mutable data type (like what the `del` statement and `append()` method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.

Mutable versus immutable types may seem like a meaningless distinction, but "Passing References" on page 100 will explain the different behavior when calling functions with mutable arguments versus immutable arguments. But first, let's find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

The *tuple* data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and]. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed. Enter the following into the interactive shell, and look at the `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, it's fine to have a trailing comma after the last item in a list or tuple in Python.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))
<class 'tuple'>
>>> type('hello')
<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Converting Types with the list() and tuple() Functions

Just like how `str(42)` will return `'42'`, the string representation of the integer 42, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a tuple to a list is handy if you need a mutable version of a tuple value.

REFERENCES

As you've seen, variables "store" strings and integer values. However, this explanation is a simplification of what Python is actually doing. Technically, variables are storing references to the computer memory locations where the values are stored. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

When you assign 42 to the `spam` variable, you are actually creating the 42 value in the computer's memory and storing a *reference* to it in the `spam` variable. When you copy the value in `spam` and assign it to the variable `cheese`, you are actually copying the reference. Both the `spam` and `cheese` variables refer to the 42 value in the computer's memory. When you later change the value in `spam` to 100, you're creating a new 100 value and storing a reference to it in `spam`. This doesn't affect the value in `cheese`. Integers are *immutable* values that don't change; changing the `spam` variable is actually making it refer to a completely different value in memory.

But lists don't work this way, because list values can change; that is, lists are *mutable*. Here is some code that will make this distinction easier to understand. Enter this into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam # The reference is being copied, not the list.
❸ >>> cheese[1] = 'Hello!' # This changes the list value.
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese # The cheese variable refers to the same list.
[0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code touched only the `cheese` list, but it seems that both the `cheese` and `spam` lists have changed.

When you create the list ❶, you assign a reference to it in the `spam` variable. But the next line ❷ copies only the list reference in `spam` to `cheese`, not the list value itself. This means the values stored in `spam` and `cheese` now both refer to the same list. There is only one underlying list because the list itself was never actually copied. So when you modify the first element of `cheese` ❸, you are modifying the same list that `spam` refers to.

Remember that variables are like boxes that contain values. The previous figures in this chapter show that lists in boxes aren't exactly accurate, because list variables don't actually contain lists—they contain *references* to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, Figure 4-4 shows what happens when a list is assigned to the `spam` variable.

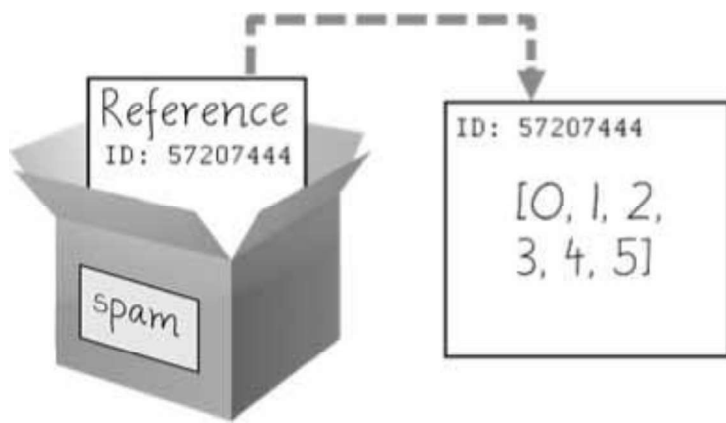


Figure 4-4: `spam = [0, 1, 2, 3, 4, 5]` stores a reference to a list, not the actual list.

Then, in Figure 4-5, the reference in `spam` is copied to `cheese`. Only a new reference was created and stored in `cheese`, not a new list. Note how both references refer to the same list.

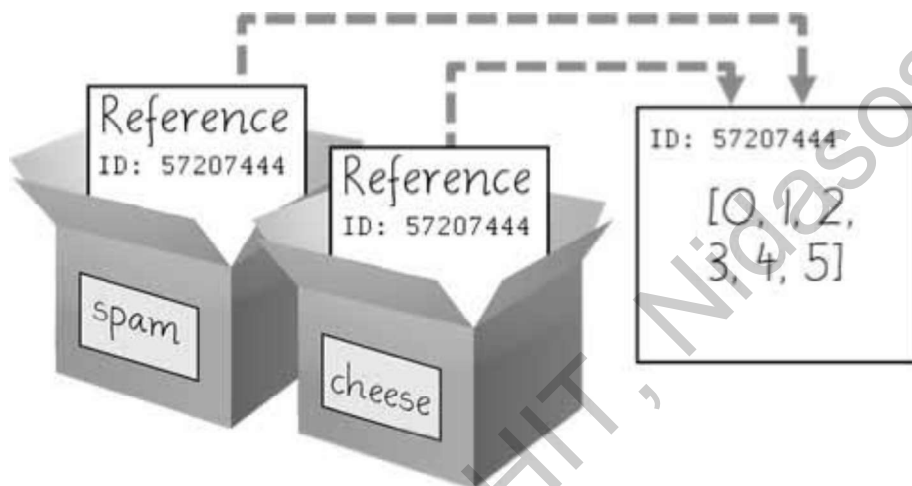


Figure 4-5: `spam = cheese` copies the reference, not the list.

When you alter the list that `cheese` refers to, the list that `spam` refers to is also changed, because both `cheese` and `spam` refer to the same list. You can see this in Figure 4-6.

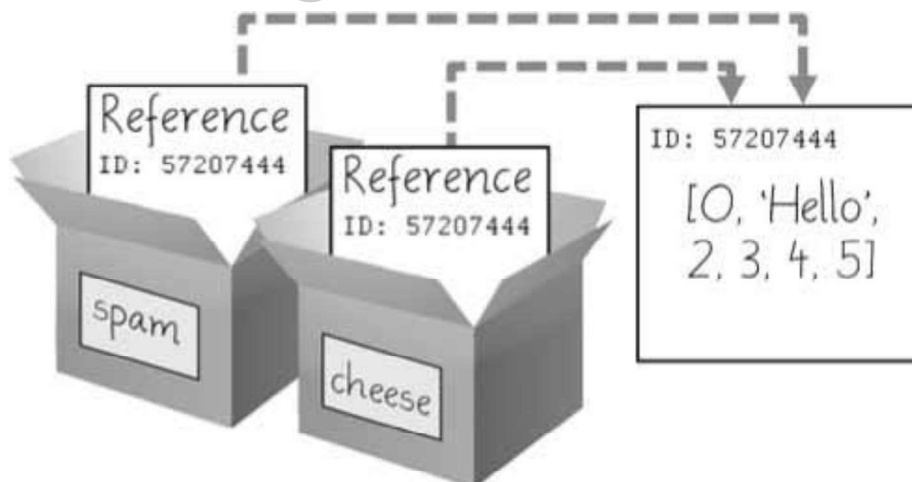


Figure 4-6: `cheese[1] = 'Hello!'` modifies the list that both variables refer to.

Although Python variables technically contain references to values, people often casually say that the variable contains the value.

Identity and the id() Function

You may be wondering why the weird behavior with mutable lists in the previous section doesn't happen with immutable values like integers or strings. We can use Python's `id()` function to understand this. All values in Python have a unique identity that can be obtained with the `id()` function. Enter the following into the interactive shell:

```
>>> id('Howdy') # The returned number will be different on your machine.
44491136
```

When Python runs `id('Howdy')`, it creates the 'Howdy' string in the computer's memory. The numeric memory address where the string is stored is returned by the `id()` function. Python picks this address based on which memory bytes happen to be free on your computer at the time, so it'll be different each time you run this code.

Like all strings, 'Howdy' is immutable and cannot be changed. If you “change” the string in a variable, a new string object is being made at a different place in memory, and the variable refers to this new string. For example, enter the following into the interactive shell and see how the identity of the string referred to by `bacon` changes:

```
>>> bacon = 'Hello'
>>> id(bacon)
44491136
>>> bacon += ' world!' # A new string is made from 'Hello' and ' world!'.
>>> id(bacon) # bacon now refers to a completely different string.
44609712
```

However, lists can be modified because they are mutable objects. The `append()` method doesn't create a new list object; it changes the existing list object. We call this “modifying the object *in-place*.”

```
>>> eggs = ['cat', 'dog'] # This creates a new list.
>>> id(eggs)
35152584
>>> eggs.append('moose') # append() modifies the list "in place".
>>> id(eggs) # eggs still refers to the same list as before.
35152584
>>> eggs = ['bat', 'rat', 'cow'] # This creates a new list, which has a new
identity.
```

```
>>> id(eggs) # eggs now refers to a completely different list.
```

```
44409800
```

If two variables refer to the same list (like `spam` and `cheese` in the previous section) and the list value itself changes, both variables are affected because they both refer to the same list. The `append()`, `extend()`, `remove()`, `sort()`, `reverse()`, and other list methods modify their lists in place.

Python's *automatic garbage collector* deletes any values not being referred to by any variables to free up memory. You don't need to worry about how the garbage collector works, which is a good thing: manual memory management in other programming languages is a common source of bugs.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries, which I'll describe in the next chapter), this means a copy of the reference is used for the parameter. To see the consequences of this, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when `eggs()` is called, a return value is not used to assign a new value to `spam`. Instead, it modifies the list in place, directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

The copy Module's copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value. For this, Python provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. The first of these, `copy.copy()`, can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> id(spam)
44684232
>>> cheese = copy.copy(spam)
>>> id(cheese) # cheese is a different list with different identity.
44685832
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

Now the `spam` and `cheese` variables refer to separate lists, which is why only the list in `cheese` is modified when you assign 42 at index 1. As you can see in Figure 4-7, the reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

Figure 4-7: `cheese = copy.copy(spam)` creates a second list that can be modified independently of the first.

If the list you need to copy contains lists, then use the `copy.deepcopy()` function instead of `copy.copy()`. The `deepcopy()` function will copy these inner lists as well.

A SHORT PROGRAM: CONWAY'S GAME OF LIFE

Conway's Game of Life is an example of *cellular automata*: a set of rules governing the behavior of a field made up of discrete cells. In practice, it creates a pretty animation to look at. You can draw out each step on graph paper, using the squares as cells. A filled-in square will be "alive" and an empty square will be "dead." If a living square has two or three living neighbors, it continues to live on the next step. If a dead square has exactly three living neighbors, it comes alive on the next step. Every other square dies or remains dead on the next step. You can see an example of the progression of steps in Figure 4-8.

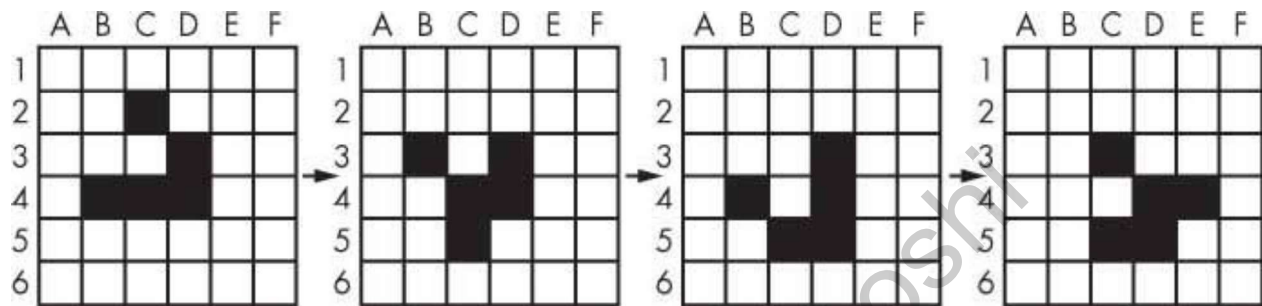


Figure 4-8: Four steps in a Conway's Game of Life simulation

Even though the rules are simple, there are many surprising behaviors that emerge. Patterns in Conway's Game of Life can move, self-replicate, or even mimic CPUs. But at the foundation of all of this complex, advanced behavior is a rather simple program.

We can use a list of lists to represent the two-dimensional field. The inner list represents each column of squares and stores a '#' hash string for living squares and a ' ' space string for dead squares. Type the following source code into the file editor, and save the file as *conway.py*. It's fine if you don't quite understand how all of the code works; just enter it and follow along with comments and explanations provided here as close as you can:

```
# Conway's Game of Life
import random, time, copy

WIDTH = 60
HEIGHT = 20

# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
```

```

else:
    column.append(' ') # Add a dead cell.
nextCells.append(column) # nextCells is a list of column lists.

while True: # Main program loop.
    print('\n\n\n\n\n') # Separate each step with newlines.
    currentCells = copy.deepcopy(nextCells)

    # Print currentCells on the screen:
    for y in range(HEIGHT):
        for x in range(WIDTH):
            print(currentCells[x][y], end=" ") # Print the # or space.
            print() # Print a newline at the end of the row.

    # Calculate the next step's cells based on current step's cells:
    for x in range(WIDTH):
        for y in range(HEIGHT):
            # Get neighboring coordinates:
            # `x - 1` ensures leftCoord is always between 0 and WIDTH - 1
            leftCoord = (x - 1) % WIDTH
            rightCoord = (x + 1) % WIDTH
            aboveCoord = (y - 1) % HEIGHT
            belowCoord = (y + 1) % HEIGHT

            # Count number of living neighbors:
            numNeighbors = 0
            if currentCells[leftCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-left neighbor is alive.
            if currentCells[x][aboveCoord] == '#':
                numNeighbors += 1 # Top neighbor is alive.
            if currentCells[rightCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-right neighbor is alive.
            if currentCells[leftCoord][y] == '#':
                numNeighbors += 1 # Left neighbor is alive.
            if currentCells[rightCoord][y] == '#':
                numNeighbors += 1 # Right neighbor is alive.
            if currentCells[leftCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-left neighbor is alive.
            if currentCells[x][belowCoord] == '#':
                numNeighbors += 1 # Bottom neighbor is alive.

```

```

if currentCells[rightCoord][belowCoord] == '#':
    numNeighbors += 1 # Bottom-right neighbor is alive.

# Set cell based on Conway's Game of Life rules:
if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
    # Living cells with 2 or 3 neighbors stay alive:
    nextCells[x][y] = '#'
elif currentCells[x][y] == '.' and numNeighbors == 3:
    # Dead cells with 3 neighbors become alive:
    nextCells[x][y] = '#'
else:
    # Everything else dies or stays dead:
    nextCells[x][y] = '.'
time.sleep(1) # Add a 1-second pause to reduce flickering.

```

Let's look at this code line by line, starting at the top.

```

# Conway's Game of Life
import random, time, copy
WIDTH = 60
HEIGHT = 20

```

First we import modules that contain functions we'll need, namely the `random.randint()`, `time.sleep()`, and `copy.deepcopy()` functions.

```

# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
        else:
            column.append('.') # Add a dead cell.
    nextCells.append(column) # nextCells is a list of column lists.

```

The very first step of our cellular automata will be completely random. We need to create a list of lists data structure to store the '#' and '.' strings that represent a living or dead cell, and their place in the list of lists reflects their position on the screen. The inner

lists each represent a column of cells. The `random.randint(0, 1)` call gives an even 50/50 chance between the cell starting off alive or dead.

We put this list of lists in a variable called `nextCells`, because the first step in our main program loop will be to copy `nextCells` into `currentCells`. For our list of lists data structure, the x-coordinates start at 0 on the left and increase going right, while the y-coordinates start at 0 at the top and increase going down. So `nextCells[0][0]` will represent the cell at the top left of the screen, while `nextCells[1][0]` represents the cell to the right of that cell and `nextCells[0][1]` represents the cell beneath it.

```
while True: # Main program loop.  
    print('\n\n\n\n\n\n') # Separate each step with newlines.  
    currentCells = copy.deepcopy(nextCells)
```

Each iteration of our main program loop will be a single step of our cellular automata. On each step, we'll copy `nextCells` to `currentCells`, print `currentCells` on the screen, and then use the cells in `currentCells` to calculate the cells in `nextCells`.

```
# Print currentCells on the screen:  
for y in range(HEIGHT):  
    for x in range(WIDTH):  
        print(currentCells[x][y], end=" ") # Print the # or space.  
    print() # Print a newline at the end of the row.
```

These nested for loops ensure that we print a full row of cells to the screen, followed by a newline character at the end of the row. We repeat this for each row in `nextCells`.

```
# Calculate the next step's cells based on current step's cells:  
for x in range(WIDTH):  
    for y in range(HEIGHT):  
        # Get neighboring coordinates:  
        # '% WIDTH' ensures leftCoord is always between 0 and WIDTH - 1  
        leftCoord = (x - 1) % WIDTH  
        rightCoord = (x + 1) % WIDTH  
        aboveCoord = (y - 1) % HEIGHT  
        belowCoord = (y + 1) % HEIGHT
```

Next, we need to use two nested for loops to calculate each cell for the next step. The living or dead state of the cell depends on the neighbors, so let's first calculate the index of the cells to the left, right, above, and below the current x- and y-coordinates.

The % mod operator performs a “wraparound.” The left neighbor of a cell in the leftmost column 0 would be $0 - 1$ or -1 . To wrap this around to the rightmost column’s index, 59, we calculate $(0 - 1) \% \text{WIDTH}$. Since WIDTH is 60, this expression evaluates to 59. This mod-wraparound technique works for the right, above, and below neighbors as well.

```
# Count number of living neighbors:
numNeighbors = 0
if currentCells[leftCoord][aboveCoord] == '#':
    numNeighbors += 1 # Top-left neighbor is alive.
if currentCells[x][aboveCoord] == '#':
    numNeighbors += 1 # Top neighbor is alive.
if currentCells[rightCoord][aboveCoord] == '#':
    numNeighbors += 1 # Top-right neighbor is alive.
if currentCells[leftCoord][y] == '#':
    numNeighbors += 1 # Left neighbor is alive.
if currentCells[rightCoord][y] == '#':
    numNeighbors += 1 # Right neighbor is alive.
if currentCells[leftCoord][belowCoord] == '#':
    numNeighbors += 1 # Bottom-left neighbor is alive.
if currentCells[x][belowCoord] == '#':
    numNeighbors += 1 # Bottom neighbor is alive.
if currentCells[rightCoord][belowCoord] == '#':
    numNeighbors += 1 # Bottom-right neighbor is alive.
```

To decide if the cell at `nextCells[x][y]` should be living or dead, we need to count the number of living neighbors `currentCells[x][y]` has. This series of if statements checks each of the eight neighbors of this cell, and adds 1 to `numNeighbors` for each living one.

```
# Set cell based on Conway's Game of Life rules:
if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
    # Living cells with 2 or 3 neighbors stay alive:
    nextCells[x][y] = '#'
elif currentCells[x][y] == '.' and numNeighbors == 3:
    # Dead cells with 3 neighbors become alive:
    nextCells[x][y] = '#'
else:
    # Everything else dies or stays dead:
```



```
nextCells[x][y] = ''
time.sleep(1) # Add a 1-second pause to reduce flickering.
```

Now that we know the number of living neighbors for the cell at `currentCells[x][y]`, we can set `nextCells[x][y]` to either '#' or '.'. After we loop over every possible x- and y-coordinate, the program takes a 1-second pause by calling `time.sleep(1)`. Then the program execution goes back to the start of the main program loop to continue with the next step.

Several patterns have been discovered with names such as “glider,” “propeller,” or “heavyweight spaceship.” The glider pattern, pictured in Figure 4-8, results in a pattern that “moves” diagonally every four steps. You can create a single glider by replacing this line in our `conway.py` program:

```
if random.randint(0, 1) == 0:
```

with this line:

```
if (x, y) in ((1, 0), (2, 1), (0, 2), (1, 2), (2, 2)):
```

You can find out more about the intriguing devices made using Conway’s Game of Life by searching the web. And you can find other short, text-based Python programs like this one at <https://github.com/asweigart/pythonstdiagames>.

SUMMARY

Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you will see programs using lists to do things that would be difficult or impossible to do without them.

Lists are a sequence data type that is mutable, meaning that their contents can change. Tuples and strings, though also sequence data types, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the `append()` or `remove()` methods do on lists.

Variables do not store list values directly; they store *references* to lists. This is an important distinction when you are copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

PRACTICE QUESTIONS

1. What is []?
2. How would you assign the value 'hello' as the third value in a list stored in a variable named spam? (Assume spam contains [2, 4, 6, 8, 10].)

For the following three questions, let's say spam contains the list ['a', 'b', 'c', 'd'].

3. What does spam[int(int('3' * 2) // 11)] evaluate to?
4. What does spam[-1] evaluate to?
5. What does spam[:2] evaluate to?

For the following three questions, let's say bacon contains the list [3.14, 'cat', 11, 'cat', True].

6. What does bacon.index('cat') evaluate to?
7. What does bacon.append(99) make the list value in bacon look like?
8. What does bacon.remove('cat') make the list value in bacon look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the append() and insert() list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you type the tuple value that has just the integer value 42 in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that “contain” list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between copy.copy() and copy.deepcopy()?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous `spam` list to the function would return 'apples, bananas, tofu, and cats'. But your function should be able to work with any list value passed to it. Be sure to test the case where an empty list `[]` is passed to your function.

Coin Flip Streaks

For this exercise, we'll try doing an experiment. If you flip a coin 100 times and write down an "H" for each heads and "T" for each tails, you'll create a list that looks like "T T T H H H H T T." If you ask a human to make up 100 random coin flips, you'll probably end up with alternating head-tail results like "H T H T H H T H T T," which looks random (to humans), but isn't mathematically random. A human will almost never write down a streak of six heads or six tails in a row, even though it is highly likely to happen in truly random coin flips. Humans are predictably bad at being random.

Write a program to find out how often a streak of six heads or a streak of six tails comes up in a randomly generated list of heads and tails. Your program breaks up the experiment into two parts: the first part generates a list of randomly selected 'heads' and 'tails' values, and the second part checks if there is a streak in it. Put all of this code in a loop that repeats the experiment 10,000 times so we can find out what percentage of the coin flips contains a streak of six heads or tails in a row. As a hint, the function call `random.randint(0, 1)` will return a 0 value 50% of the time and a 1 value the other 50% of the time.

You can start with the following template:

```
import random
numberOfStreaks = 0
for experimentNumber in range(10000):
    # Code that creates a list of 100 'heads' or 'tails' values.

    # Code that checks if there is a streak of 6 heads or tails in a row.
print('Chance of streak: %s%%' % (numberOfStreaks / 100))
```

Of course, this is only an estimate, but 10,000 is a decent sample size. Some knowledge of mathematics could give you the exact answer and save you the trouble of writing a program, but programmers are notoriously bad at math.

Character Picture Grid

Say you have a list of lists where each value in the inner lists is a one-character string, like this:

```

grid = [['.', '.', '.', '.', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['.', 'O', 'O', 'O', 'O', 'O'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]

```

Think of `grid[x][y]` as being the character at the `x`- and `y`-coordinates of a “picture” drawn with text characters. The (0, 0) origin is in the upper-left corner, the `x`-coordinates increase going right, and the `y`-coordinates increase going down.

Copy the previous grid value, and write code that uses it to print the image.

```

..OO.OO..
.OOOOOOO.
.OOOOOOO.
..OOOOO..
...OOO...
....O....

```

Hint: You will need to use a loop in a loop in order to print `grid[0][0]`, then `grid[1][0]`, then `grid[2][0]`, and so on, up to `grid[8][0]`. This will finish the first row, so then print a newline. Then your program should print `grid[0][1]`, then `grid[1][1]`, then `grid[2][1]`, and so on. The last thing your program will print is `grid[8][5]`.

Also, remember to pass the `end` keyword argument to `print()` if you don't want a newline printed automatically after each `print()` call.

/

/

DICTIONARIES AND STRUCTURING DATA

In this chapter, I will cover the dictionary data type, which provides a flexible way to access and organize data. Then, combining dictionaries with your knowledge of lists from the previous chapter, you'll learn how to create a data structure to model a tic-tac-toe board.

THE DICTIONARY DATA TYPE

Like a list, a *dictionary* is a mutable collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is typed with braces, `{}`. Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively. You can access these values through their keys:

```
>>> myCat['size']
```

```
'fat'
```

```
>>> 'My cat has ' + myCat['color'] + ' fur.'
```

```
'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at 0 and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named `spam` would be `spam[0]`. But there is no “first” item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary. Enter the following into the interactive shell:

```

>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True

```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no 'color' key:

```

>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'

```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code. Save it as *birthdays.py*.

```

❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

```

```

while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == "":
        break

```

```

❷ if name in birthdays:

```

```
③ print(birthdays[name] + ' is the birthday of ' + name)
```

else:

```
print('I do not have birthday information for ' + name)
```

```
print('What is their birthday?')
```

```
bday = input()
```

```
④ birthdays[name] = bday
```

```
print('Birthday database updated.')
```

You can view the execution of this program at <https://autbor.com/bdaydb>. You create an initial dictionary and store it in `birthdays` ①. You can see if the entered name exists as a key in the dictionary with the `in` keyword ②, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ③; if not, you can add it using the same square bracket syntax combined with the assignment operator ④.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
```

```
Alice
```

```
Apr 1 is the birthday of Alice
```

```
Enter a name: (blank to quit)
```

```
Eve
```

```
I do not have birthday information for Eve
```

```
What is their birthday?
```

```
Dec 5
```

```
Birthday database updated.
```

```
Enter a name: (blank to quit)
```

```
Eve
```

```
Dec 5 is the birthday of Eve
```

```
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in Chapter 9.

ORDERED DICTIONARIES IN PYTHON 3.7

While they're still not ordered and have no "first" key-value pair, dictionaries in Python 3.7 and later will remember the insertion order of their key-value pairs if you create a sequence value from

them. For example, notice the order of items in the lists made from the eggs and ham dictionaries matches the order in which they were entered:

```
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> list(eggs)
['name', 'species', 'age']
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> list(ham)
['species', 'age', 'name']
```

The dictionaries are still unordered, as you can't access items in them using integer indexes like `eggs[0]` or `ham[2]`. You shouldn't rely on this behavior, as dictionaries in older versions of Python don't remember the insertion order of key-value pairs. For example, notice how the list doesn't match the insertion order of the dictionary's key-value pairs when I run this code in Python 3.5:

```
>>> spam = {}
>>> spam['first key'] = 'value'
>>> spam['second key'] = 'value'
>>> spam['third key'] = 'value'
>>> list(spam)
['first key', 'third key', 'second key']
```

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods are not true lists: they cannot be modified and do not have an `append()` method. But these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
...     print(v)
red
42
```


Here, a for loop iterates over each of the values in the spam dictionary. A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():  
...     print(k)
```

```
color
```

```
age
```

```
>>> for i in spam.items():  
...     print(i)
```

```
('color', 'red')
```

```
('age', 42)
```

When you use the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the `list()` function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
```

```
>>> spam.keys()
```

```
dict_keys(['color', 'age'])
```

```
>>> list(spam.keys())
```

```
['color', 'age']
```

The `list(spam.keys())` line takes the `dict_keys` value returned from `keys()` and passes it to `list()`, which then returns a list value of `['color', 'age']`.

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
```

```
>>> for k, v in spam.items():
```

```
...     print('Key: ' + k + ' Value: ' + str(v))
```

```
Key: age Value: 42
```

```
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

In the previous example, notice that `'color' in spam` is essentially a shorter version of writing `'color' in spam.keys()`. This is always the case: if you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the `in` (or `not in`) keyword with the dictionary value itself.

The get() Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no `'eggs'` key in the `picnicItems` dictionary, the default value `0` is returned by the `get()` method. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

The.setdefault() Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'`, because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as `characterCount.py`:

```

message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    ❶ count.setdefault(character, 0)
    ❷ count[character] = count[character] + 1

print(count)

```

You can view the execution of this program at <https://autbor.com/setdefault>. The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ❶ ensures that the key is in the `count` dictionary (with a default value of 0) so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed ❷. When you run this program, the output will look like this:

```

{' ': 13, ',': 1, '!': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2,
'i': 6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}

```

From the output, you can see that the lowercase letter `c` appears 3 times, the space character appears 13 times, and the uppercase letter `A` appears 1 time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

PRETTY PRINTING

If you import the `pprint` module into your programs, you'll have access to the `pprint()` and `pformat()` functions that will “pretty print” a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what `print()` provides. Modify the previous `characterCount.py` program and save it as `prettyCharacterCount.py`.

```

import pprint

```

```

message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}
for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

```

`pprint.pprint(count)`

You can view the execution of this program at <https://autbor.com/pprint/>. This time, when the program is run, the output looks much cleaner, with the keys sorted.

```
{' ': 13,  
'!': 1,  
'!': 1,  
'A': 1,  
'T': 1,  
--snip--  
't': 6,  
'w': 2,  
'y': 1}
```

The `pprint.pprint()` function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call `pprint.pformat()` instead. These two lines are equivalent to each other:

CSE, HIT, Nidasoshi

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

USING DATA STRUCTURES TO MODEL REAL-WORLD THINGS

Even before the internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in Figure 5-1.

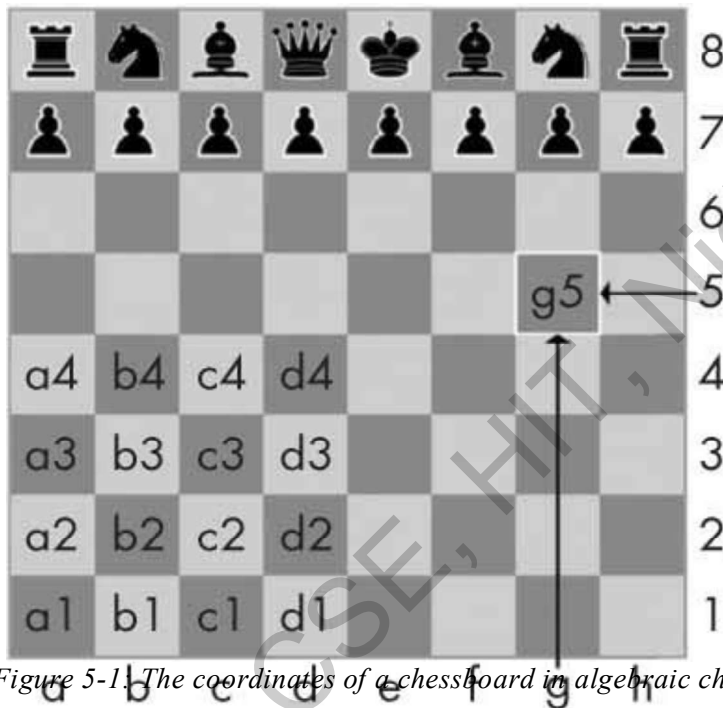


Figure 5-1. The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation *2. Nf3 Nc6* indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: you can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings like '2. Nf3 Nc6'. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model.

This is where lists and dictionaries can come in. For example, the dictionary {'1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'} could represent the chess board in Figure 5-2.



Figure 5-2: A chess board modeled by the dictionary {'1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'}

But for another example, you'll use a game that's a little simpler than chess: tic-tac-toe.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an *X*, an *O*, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure 5-3.

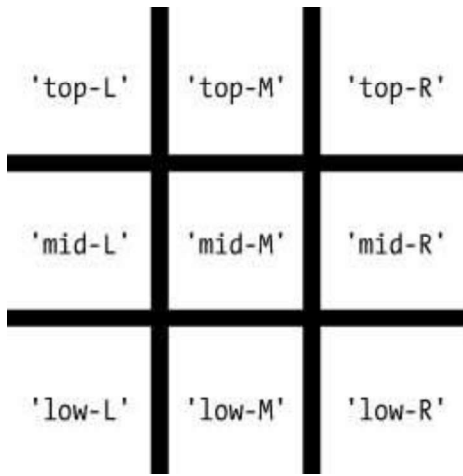


Figure 5-3: The slots of a tic-tac-toe board with their corresponding keys

You can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key 'top-R' can represent the top-right corner, the string value with the key 'low-L' can represent the bottom-left corner, the string value with the key 'mid-M' can represent the middle, and so on.

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named `theBoard`. Open a new file editor window, and enter the following source code, saving it as *ticTacToe.py*:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure stored in the `theBoard` variable represents the tic-tac-toe board in Figure 5-4.

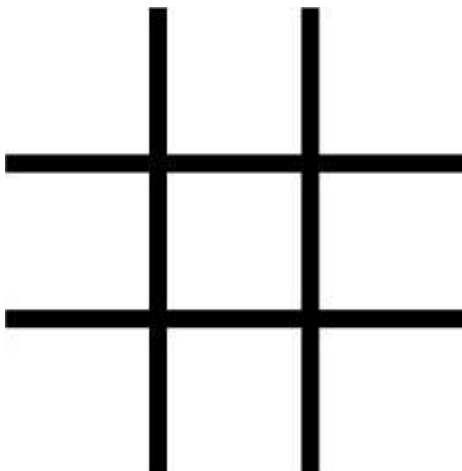


Figure 5-4: An empty tic-tac-toe board

you could represent that board with this dictionary:

```
theBoard = {'top-L': '', 'top-M': '', 'top-R': '',  
            'mid-L': '', 'mid-M': 'X', 'mid-R': '',  
            'low-L': '', 'low-M': '', 'low-R': ''}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-5.

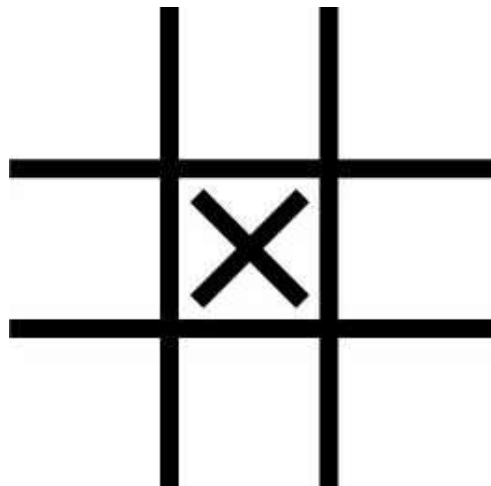


Figure 5-5: The first move

A board where player O has won by placing Os across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',  
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': '',  
            'low-L': '', 'low-M': '', 'low-R': 'X'}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-6.

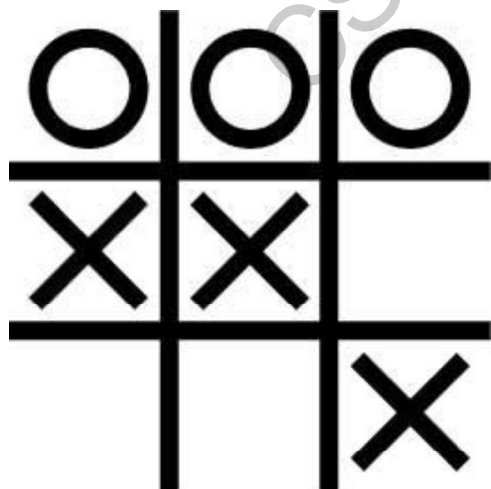


Figure 5-6: Player O wins.

Of course, the player sees only what is printed to the screen, not the contents of variables. Let's create a function to print the board dictionary onto the screen. Make the following addition to `ticTacToe.py` (new code is in bold):

```
theBoard = {'top-L': '', 'top-M': '', 'top-R': '',
            'mid-L': '', 'mid-M': '', 'mid-R': '',
            'low-L': '', 'low-M': '', 'low-R': ''}
```

```
def printBoard(board):
```

```
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
```

```
printBoard(theBoard)
```

You can view the execution of this program at <https://autbor.com/tictactoe1/>. When you run this program, printBoard() will print out a blank tic-tac-toe board.

```
||
-+-+-
||
-+-+-
||
```

The printBoard() function can handle any tic-tac-toe data structure you pass it. Try changing the code to the following:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
            'X', 'mid-R': '', 'low-L': '', 'low-M': '', 'low-R': 'X'}
```

```
def printBoard(board):
```

```
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
```

```
printBoard(theBoard)
```

You can view the execution of this program at <https://autbor.com/tictactoe2/>. Now when you run this program, the new board will be printed to the screen.

```
O|O|O
-+-+-
X|X|
```

-+-+-

|X

Because you created a data structure to represent a tic-tac-toe board and wrote code in `printBoard()` to interpret that data structure, you now have a program that “models” the tic-tac-toe board. You could have organized your data structure differently (for example, using keys like 'TOP-LEFT' instead of 'top-L'), but as long as the code works with your data structures, you will have a correctly working program.

For example, the `printBoard()` function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots. If the dictionary you passed was missing, say, the 'mid-L' key, your program would no longer work.

O|O|O

-+-+-

Traceback (most recent call last):

File "ticTacToe.py", line 10, in <module>

printBoard(theBoard)

File "ticTacToe.py", line 6, in printBoard

print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])

KeyError: 'mid-L'

Now let's add code that allows the players to enter their moves. Modify the *ticTacToe.py* program to look like this:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M':  
' ', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

```
def printBoard(board):
```

```
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
```

```
    print('-+-+')
```

```
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
```

```
    print('-+-+')
```

```
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
```

```
turn = 'X'
```

```
for i in range(9):
```

```
    ❶ printBoard(theBoard)
```

```
        print('Turn for ' + turn + '. Move on which space?')
```

```
    ❷ move = input()
```

```
    ❸ theBoard[move] = turn
```

```
    ❹ if turn == 'X':
```

```
    turn = 'O'
else:
    turn = 'X'
printBoard(theBoard)
```

You can view the execution of this program at <https://autbor.com/tictactoe3/>. The new code prints out the board at the start of each new turn ❶, gets the active player's move ❷, updates the game board accordingly ❸, and then swaps the active player ❹ before moving on to the next turn.

When you run this program, it will look something like this:

```
||
-+-+
||
-+-+
||
Turn for X. Move on which space?
```

mid-M

```
||
-+-+
|X|
-+-+
||
```

--snip--

```
O|O|X
-+-+
X|X|O
-+-+
O| |X
```

Turn for X. Move on which space?

low-M

```
O|O|X
-+-+
X|X|O
-+-+
O|X|X
```

This isn't a complete tic-tac-toe game—for instance, it doesn't ever check whether a player has won—but it's enough to see how data structures can be used in programs.

NOTE

If you are curious, the source code for a complete tic-tac-toe program is described in the resources available from <https://nostarch.com/automatestuff2/>.

Nested Dictionaries and Lists

Modeling a tic-tac-toe board was fairly simple: the board needed only a single dictionary value with nine key-value pairs. As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary that contains other dictionaries of what items guests are bringing to a picnic. The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
            'Bob': {'ham sandwiches': 3, 'apples': 2},
            'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
    ❶ for k, v in guests.items():
        ❷ numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples      ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups        ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes        ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies   ' + str(totalBrought(allGuests, 'apple pies')))
```

You can view the execution of this program at <https://autbor.com/guestpicnic/>. Inside the `totalBrought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the `item` parameter exists as a key in this dictionary,

its value (the quantity) is added to `numBrought` ❷. If it does not exist as a key, the `get()` method returns 0 to be added to `numBrought`.

The output of this program looks like this:

Number of things being brought:

- Apples 7
 - Cups 3
 - Cakes 0
 - Ham Sandwiches 3
 - Apple Pies 1
-

This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `totalBrought()` function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items. Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about the "right" way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program's needs.

SUMMARY

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another (the value), as opposed to lists, which simply contain a series of values in order. Values inside a dictionary are accessed using square brackets just as with lists. Instead of an integer index, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program's values into data structures, you can create representations of real-world objects. You saw an example of this with a tic-tac-toe board.

PRACTICE QUESTIONS

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key 'foo' and a value 42 look like?
3. What is the main difference between a dictionary and a list?
4. What happens if you try to access `spam['foo']` if `spam` is `{'bar': 100}`?

5. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.keys()`?
6. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.values()`?
7. What is a shortcut for the following code?

```
if 'color' not in spam:  
    spam['color'] = 'black'
```

8. What module and function can be used to “pretty print” dictionary values?

CSE, HIT, Nidasoshi

MANIPULATING STRINGS

Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values together with the + operator, but you can do much more than that. You can extract partial strings from string values, add or remove spacing, convert letters to lowercase or uppercase, and check that strings are formatted correctly. You can even write Python code to access the clipboard for copying and pasting text.

In this chapter, you'll learn all this and more. Then you'll work through two different programming projects: a simple clipboard that stores multiple strings of text and a program to automate the boring chore of formatting pieces of text.

WORKING WITH STRINGS

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

Typing string values in Python code is fairly straightforward: they begin and end with a single quote. But then how can you use a quote inside a string? Typing 'That is Alice's cat.' won't work, because Python thinks the string ends after Alice, and the rest (s cat.) is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character you want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) For example, the escape character for a single quote is \'. You

quotes and double quotes inside your strings, respectively.

Table 6-1 lists the escape characters you can use.

Table 6-1: Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

Enter the following into the interactive shell:

```
>>> print("Hello there!\nHow are you?\nI'm doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Raw Strings

You can place an `r` before the beginning quotation mark of a string to make it a raw string. A *raw string* completely ignores all escape characters and prints any backslash that appears in the string. For example, enter the following into the interactive shell:

```
>>> print(r'That is Carol's cat.')
That is Carol's cat.
```

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character. Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for Windows file paths like `r'C:\Users\A\Desktop'` or regular expressions described in the next chapter.

Multiline Strings with Triple Quotes

While you can use the `\n` escape character to put a newline into a string, it is often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the “triple quotes” are considered part of the string. Python’s indentation rules for blocks do not apply to lines inside a multiline string.

Open the file editor and write the following:

```
print("Dear Alice,
```

```
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
```

```
Sincerely,
```

```
Bob")
```

Save this program as *catnapping.py* and run it. The output will look like this:

```
Dear Alice,
```

```
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
```

```
Sincerely,
```

```
Bob
```

Notice that the single quote character in Eve's does not need to be escaped. Escaping single and double quotes is optional in multiline strings. The following `print()` call would print identical text but doesn't use a multiline string:

```
print('Dear Alice,\n\nEve's cat has been arrested for catnapping, cat\n\nburglary, and extortion.\n\nSincerely,\nBob')
```

Multiline Comments

While the hash character (`#`) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines. The following is perfectly valid Python code:

```
"""This is a test Python program.
```

```
Written by Al Sweigart al@inventwithpython.com
```

```
This program was designed for Python 3, not Python 2.
```

```
"""
```

```
def spam():
```

```
    """This is a multiline comment to help  
    explain what the spam() function does."""
```

```
    print('Hello!')
```

Indexing and Slicing Strings

Strings use indexes and slices the same way lists do. You can think of the string 'Hello, world!' as a list and each character in the string as an item with a corresponding index.

```
' H e l l o ,   w o r l d ! '  
 0 1 2 3 4 5 6 7 8 9 10 11 12
```

The space and exclamation point are included in the character count, so 'Hello, world!' is 13 characters long, from H at index 0 to ! at index 12.

Enter the following into the interactive shell:

```
>>> spam = 'Hello, world!'  
>>> spam[0]  
'H'  
>>> spam[4]  
'o'  
>>> spam[-1]  
'!'  
>>> spam[0:5]  
'Hello'  
>>> spam[:5]  
'Hello'  
>>> spam[7:]  
'world!'
```

If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not. That's why, if spam is 'Hello, world!', spam[0:5] is 'Hello'. The substring you get from spam[0:5] will include everything from spam[0] to spam[4], leaving out the comma at index 5 and the space at index 6. This is similar to how range(5) will cause a for loop to iterate up to, but not including, 5.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try entering the following into the interactive shell:

```
>>> spam = 'Hello, world!'  
>>> fizz = spam[0:5]  
>>> fizz  
'Hello'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

The in and not in Operators with Strings

The `in` and `not in` operators can be used with strings just like with list values. An expression with two strings joined using `in` or `not in` will evaluate to a Boolean `True` or `False`. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello, World'
```

```
True
```

```
>>> 'Hello' in 'Hello'
```

```
True
```

```
>>> 'HELLO' in 'Hello, World'
```

```
False
```

```
>>> '' in 'spam'
```

```
True
```

```
>>> 'cats' not in 'cats and dogs'
```

```
False
```

These expressions test whether the first string (the exact string, case-sensitive) can be found within the second string.

PUTTING STRINGS INSIDE OTHER STRINGS

Putting strings inside other strings is a common operation in programming. So far, we've been using the `+` operator and string concatenation to do this:

```
>>> name = 'Al'
```

```
>>> age = 4000
```

```
>>> 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
```

```
'Hello, my name is Al. I am 4000 years old.'
```

However, this requires a lot of tedious typing. A simpler approach is to use *string interpolation*, in which the `%s` operator inside the string acts as a marker to be replaced by values following the string. One benefit of string interpolation is that `str()` doesn't have to be called to convert values to strings. Enter the following into the interactive shell:

```
>>> name = 'Al'
```

```
>>> age = 4000
```

```
>>> 'My name is %s. I am %s years old.' % (name, age)
```

```
'My name is Al. I am 4000 years old.'
```

Python 3.6 introduced *f-strings*, which is similar to string interpolation except that braces are used instead of %s, with the expressions placed directly inside the braces. Like raw strings, f-strings have an f prefix before the starting quotation mark. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> f'My name is {name}. Next year I will be {age + 1}.'
'My name is Al. Next year I will be 4001.'
```

Remember to include the f prefix; otherwise, the braces and their contents will be a part of the string value:

```
>>> 'My name is {name}. Next year I will be {age + 1}.'
'My name is {name}. Next year I will be {age + 1}.'
```

USEFUL STRING METHODS

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

The upper(), lower(), isupper(), and islower() Methods

The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively. Nonletter characters in the string remain unchanged. Enter the following into the interactive shell:

```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam
'HELLO, WORLD!'
>>> spam = spam.lower()
>>> spam
'hello, world!'
```

Note that these methods do not change the string itself but return new string values. If you want to change the original string, you have to call upper() or lower() on the string and then assign the new string to the variable where the original was stored. This is why you must use spam = spam.upper() to change the string in spam instead of simply spam.upper().

(This is just like if a variable `eggs` contains the value 10. Writing `eggs + 3` does not change the value of `eggs`, but `eggs = eggs + 3` does.)

The `upper()` and `lower()` methods are helpful if you need to make a case-insensitive comparison. For example, the strings `'great'` and `'GREat'` are not equal to each other. But in the following small program, it does not matter whether the user types `Great`, `GREAT`, or `grEAT`, because the string is first converted to lowercase.

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

When you run this program, the question is displayed, and entering a variation on `great`, such as `GREat`, will still give the output `I feel great too`. Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail.

```
How are you?
GREat
I feel great too.
```

You can view the execution of this program at <https://autbor.com/convertlowercase/>. The `isupper()` and `islower()` methods will return a Boolean `True` value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns `False`. Enter the following into the interactive shell, and notice what each method call returns:

```
>>> spam = 'Hello, world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
```

```
>>> '12345'.isupper()
```

```
False
```

Since the `upper()` and `lower()` string methods themselves return strings, you can call string methods on *those* returned string values as well. Expressions that do this will look like a chain of method calls. Enter the following into the interactive shell:

```
>>> 'Hello'.upper()
```

```
'HELLO'
```

```
>>> 'Hello'.upper().lower()
```

```
'hello'
```

```
>>> 'Hello'.upper().lower().upper()
```

```
'HELLO'
```

```
>>> 'HELLO'.lower()
```

```
'hello'
```

```
>>> 'HELLO'.lower().islower()
```

```
True
```

The isX() Methods

Along with `islower()` and `isupper()`, there are several other string methods that have names beginning with the word *is*. These methods return a Boolean value that describes the nature of the string. Here are some common *isX* string methods:

isalpha() Returns `True` if the string consists only of letters and isn't blank

isalnum() Returns `True` if the string consists only of letters and numbers and is not blank

isdecimal() Returns `True` if the string consists only of numeric characters and is not blank

isspace() Returns `True` if the string consists only of spaces, tabs, and newlines and is not blank

istitle() Returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()
```

```
True
```

```
>>> 'hello123'.isalpha()
```

```
False
```

```
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> ' '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

The `isX()` string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input. Open a new file editor window and enter this program, saving it as `validateInput.py`:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

In the first while loop, we ask the user for their age and store their input in `age`. If `age` is a valid (decimal) value, we break out of this first while loop and move on to the second, which asks for a password. Otherwise, we inform the user that they need to enter a number and again ask them to enter their age. In the second while loop, we ask for a

password, store the user's input in password, and break out of the loop if the input was alphanumeric. If it wasn't, we're not satisfied, so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

When run, the program's output looks like this:

Enter your age:

forty two

Please enter a number for your age.

Enter your age:

42

Select a new password (letters and numbers only):

secr3t!

Passwords can only have letters and numbers.

Select a new password (letters and numbers only):

secr3t

You can view the execution of this program at <https://autbor.com/validateinput/>. Calling `isdecimal()` and `isalnum()` on variables, we're able to test whether the values stored in those variables are decimal or not, alphanumeric or not. Here, these tests help us reject the input `forty two` but accept `42`, and reject `secr3t!` but accept `secr3t`.

The startswith() and endswith() Methods

The `startswith()` and `endswith()` methods return `True` if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return `False`. Enter the following into the interactive shell:

```
>>> 'Hello, world!'.startswith('Hello')
True
>>> 'Hello, world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello, world!'.startswith('Hello, world!')
True
>>> 'Hello, world!'.endswith('Hello, world!')
True
```

These methods are useful alternatives to the `==` equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The `join()` and `split()` Methods

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. The `join()` method is called on a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string `join()` calls on is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is `'cats, rats, bats'`.

Remember that `join()` is called on a string value and is passed a list value. (It's easy to accidentally call it the other way around.) The `split()` method does the opposite: It's called on a string value and returns a list of strings. Enter the following into the interactive shell:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the string `'My name is Simon'` is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the `split()` method to specify a different string to split upon. For example, enter the following into the interactive shell:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters.

Enter the following into the interactive shell:

```
>>> spam = '''Dear Alice,  
How have you been? I am fine.  
There is a container in the fridge  
that is labeled "Milk Experiment."'''
```

```
'''Please do not drink it.
```

```
Sincerely,
```

```
Bob'''
```

```
>>> spam.split('\n')
```

```
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the  
fridge', 'that is labeled "Milk Experiment."', ' ', 'Please do not drink it.',  
'Sincerely,', 'Bob']
```

Passing `split()` the argument `\n` lets us split the multiline string stored in `spam` along the newlines and return a list in which each item corresponds to one line of the string.

Splitting Strings with the `partition()` Method

The `partition()` string method can split a string into the text before and after a separator string. This method searches the string it is called on for the separator string it is passed, and returns a tuple of three substrings for the “before,” “separator,” and “after” substrings. Enter the following into the interactive shell:

```
>>> 'Hello, world!'.partition('w')  
(Hello, ', 'w', 'orld!')  
>>> 'Hello, world!'.partition('world')  
(Hello, ', 'world', '!')
```

If the separator string you pass to `partition()` occurs multiple times in the string that `partition()` calls on, the method splits the string only on the first occurrence:

```
>>> 'Hello, world!'.partition('o')  
(Hell, 'o', ', world!')
```

If the separator string can't be found, the first string returned in the tuple will be the entire string, and the other two strings will be empty:

```
>>> 'Hello, world!'.partition('XYZ')  
(Hello, world!', ', ')
```

You can use the multiple assignment trick to assign the three returned strings to three variables:

```
>>> before, sep, after = 'Hello, world!'.partition(' ')
>>> before
'Hello,'
>>> after
'world!'
```

The `partition()` method is useful for splitting a string whenever you need the parts before, including, and after a particular separator string.

Justifying Text with the `rjust()`, `ljust()`, and `center()` Methods

The `rjust()` and `ljust()` string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
' Hello'
>>> 'Hello'.rjust(20)
'          Hello'
>>> 'Hello, World'.rjust(20)
' Hello, World'
>>> 'Hello'.ljust(10)
'Hello '
```

'Hello'.`rjust(10)` says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'*****Hello'
>>> 'Hello'.ljust(20, '-')
'Hello.....'
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)
'   Hello   '
>>> 'Hello'.center(20,'=')
'=====Hello====='
```

These methods are especially useful when you need to print tabular data that has correct spacing. Open a new file editor window and enter the following code, saving it as *picnicTable.py*:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

You can view the execution of this program at <https://autbor.com/picnictable/>. In this program, we define a `printPicnic()` method that will take in a dictionary of information and use `center()`, `ljust()`, and `rjust()` to display that information in a neatly aligned table-like format.

The dictionary that we'll pass to `printPicnic()` is `picnicItems`. In `picnicItems`, we have 4 sandwiches, 12 apples, 4 cups, and 8,000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

To do this, we decide how wide we want the left and right columns to be. Along with our dictionary, we'll pass these values to `printPicnic()`.

The `printPicnic()` function takes in a dictionary, a `leftWidth` for the left column of a table, and a `rightWidth` for the right column. It prints a title, `PICNIC ITEMS`, centered above the table. Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

After defining `printPicnic()`, we define the dictionary `picnicItems` and call `printPicnic()` twice, passing it different widths for the left and right table columns.

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

---PICNIC ITEMS--

sandwiches.. 4

apples12

cups4

cookies8000

-----PICNIC ITEMS-----

sandwiches..... 4

apples 12

cups 4

cookies8000

Using `rjust()`, `ljust()`, and `center()` lets you ensure that strings are neatly aligned, even if you aren't sure how many characters long your strings are.

Removing Whitespace with the `strip()`, `rstrip()`, and `lstrip()` Methods

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The `strip()` string method will return a new string without any whitespace characters at the beginning or end. The `lstrip()` and `rstrip()` methods will remove whitespace characters from the left and right ends, respectively. Enter the following into the interactive shell:

```
>>> spam = ' Hello, World '
```

```
>>> spam.strip()
```

```
'Hello, World'
```

```
>>> spam.lstrip()
```

```
'Hello, World '
```

```
>>> spam.rstrip()
```

```
' Hello, World'
```

Optionally, a string argument will specify which characters on the ends should be stripped. Enter the following into the interactive shell:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
```

```
>>> spam.strip('ampS')
```

```
'BaconSpamEggs'
```

Passing `strip()` the argument `'ampS'` will tell it to strip occurrences of `a`, `m`, `p`, and capital `S` from the ends of the string stored in `spam`. The order of the characters in the string

passed to `strip()` does not matter: `strip('ampS')` will do the same thing as `strip('mapS')` or `strip('Spam')`.

NUMERIC VALUES OF CHARACTERS WITH THE `ORD()` AND `CHR()` FUNCTIONS

Computers store information as bytes—strings of binary numbers, which means we need to be able to convert text to numbers. Because of this, every text character has a corresponding numeric value called a *Unicode code point*. For example, the numeric code point is 65 for 'A', 52 for '4', and 33 for '!'. You can use the `ord()` function to get the code point of a one-character string, and the `chr()` function to get the one-character string of an integer code point. Enter the following into the interactive shell:

```
>>> ord('A')
65
>>> ord('4')
52
>>> ord('!')
33
>>> chr(65)
'A'
```

These functions are useful when you need to do an ordering or mathematical operation on characters:

```
>>> ord('B')
66
>>> ord('A') < ord('B')
True
>>> chr(ord('A'))
'A'
>>> chr(ord('A') + 1)
'B'
```

There is more to Unicode and code points, but those details are beyond the scope of this book. If you'd like to know more, I recommend watching Ned Batchelder's 2012 PyCon talk, "Pragmatic Unicode, or, How Do I Stop the Pain?" at <https://youtu.be/sgHbC6udIqc>.

COPYING AND PASTING STRINGS WITH THE `PYPERCLIP` MODULE

The `pyperclip` module has `copy()` and `paste()` functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it into an email, word processor, or some other software.

RUNNING PYTHON SCRIPTS OUTSIDE OF MU

So far, you've been running your Python scripts using the interactive shell and file editor in Mu. However, you won't want to go through the inconvenience of opening Mu and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, macOS, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using Mu.)

The `pyperclip` module does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing `pyperclip`, enter the following into the interactive shell:

```
>>> import pyperclip
>>> pyperclip.copy('Hello, world!')
>>> pyperclip.paste()
'Hello, world!'
```

Of course, if something outside of your program changes the clipboard contents, the `paste()` function will return it. For example, if I copied this sentence to the clipboard and then called `paste()`, it would look like this:

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

PROJECT: MULTI-CLIPBOARD AUTOMATIC MESSAGES

If you've responded to a large number of emails with similar phrasing, you've probably had to do a lot of repetitive typing. Maybe you keep a text document with these phrases so you can easily copy and paste them using the clipboard. But your clipboard can only store one message at a time, which isn't very convenient. Let's make this process a bit easier with a program that stores multiple phrases.

Step 1: Program Design and Data Structures

You want to be able to run this program with a command line argument that is a short key phrase—for instance, *agree* or *busy*. The message associated with that key phrase will be copied to the clipboard so that the user can paste it into an email. This way, the user can have long, detailed messages without having to retype them.

THE CHAPTER PROJECTS

This is the first “chapter project” of the book. From here on, each chapter will have projects that demonstrate the concepts covered in the chapter. The projects are written in a style that takes you from a blank file editor window to a full, working program. Just like with the interactive shell examples, don’t only read the project sections—follow along on your computer!

Open a new file editor window and save the program as *mclip.py*. You need to start the program with a *#!* (*shebang*) line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each piece of text with its key phrase, you can store these as strings in a dictionary. The dictionary will be the data structure that organizes your key phrases and text. Make your program look like the following:

```
#!/python3
# mclip.py - A multi-clipboard program.

TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",
        'busy': """Sorry, can we do this later this week or next week?""",
        'upsell': """Would you consider making this a monthly donation?"""}

```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable `sys.argv`. (See Appendix B for more information on how to use command line arguments in your programs.) The first item in the `sys.argv` list should always be a string containing the program’s filename (`'mclip.py'`), and the second item should be the first command line argument. For this program, this argument is the key phrase of the message you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the `sys.argv` list has fewer than two values in it). Make your program look like the following:

```
#!/python3
# mclip.py - A multi-clipboard program.

```

```
TEXT = {'agree': """"Yes, I agree. That sounds fine to me.""",
        'busy': """"Sorry, can we do this later this week or next week?""",
        'upsell': """"Would you consider making this a monthly donation?"""}

import sys
if len(sys.argv) < 2:
    print('Usage: python mclip.py [keyphrase] - copy phrase text')
    sys.exit()

keyphrase = sys.argv[1] # first command line arg is the keyphrase
```

Step 3: Copy the Right Phrase

Now that the key phrase is stored as a string in the variable `keyphrase`, you need to see whether it exists in the `TEXT` dictionary as a key. If so, you want to copy the key's value to the clipboard using `pyperclip.copy()`. (Since you're using the `pyperclip` module, you need to import it.) Note that you don't actually *need* the `keyphrase` variable; you could just use `sys.argv[1]` everywhere `keyphrase` is used in this program. But a variable named `keyphrase` is much more readable than something cryptic like `sys.argv[1]`.

Make your program look like the following:

```
#!/ python3
# mclip.py - A multi-clipboard program.

TEXT = {'agree': """"Yes, I agree. That sounds fine to me.""",
        'busy': """"Sorry, can we do this later this week or next week?""",
        'upsell': """"Would you consider making this a monthly donation?"""}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py mclip.py [keyphrase] - copy phrase text')
    sys.exit()

keyphrase = sys.argv[1] # first command line arg is the keyphrase

if keyphrase in TEXT:
    pyperclip.copy(TEXT[keyphrase])
    print('Text for ' + keyphrase + ' copied to clipboard.')
else:
    print('There is no text for ' + keyphrase)
```

This new code looks in the TEXT dictionary for the key phrase. If the key phrase is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no key phrase with that name.

That's the complete script. Using the instructions in Appendix B for launching command line programs easily, you now have a fast way to copy messages to the clipboard. You will have to modify the TEXT dictionary value in the source whenever you want to update the program with a new message.

On Windows, you can create a batch file to run this program with the WIN-R Run window. (For more about batch files, see Appendix B.) Enter the following into the file editor and save the file as *mclip.bat* in the *C:\Windows* folder:

```
@py.exe C:\path_to_file\mclip.py %*  
@pause
```

With this batch file created, running the multi-clipboard program on Windows is just a matter of pressing WIN-R and typing *mclip key phrase*.

PROJECT: ADDING BULLETS TO WIKI MARKUP

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front. But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, if I copied the following text (for the Wikipedia article “List of Lists of Lists”) to the clipboard:

```
Lists of animals  
Lists of aquarium life  
Lists of biologists by author abbreviation  
Lists of cultivars
```

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

```
* Lists of animals  
* Lists of aquarium life
```

* Lists of biologists by author abbreviation

* Lists of cultivars

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

1. Paste text from the clipboard.
2. Do something to it.
3. Copy the new text to the clipboard.

That second step is a little tricky, but steps 1 and 3 are pretty straightforward: they just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let's just write the part of the program that covers steps 1 and 3. Enter the following, saving the program as *bulletPointAdder.py*:

```
#!/ python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

The TODO comment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text and Add the Star

The call to `pyperclip.paste()` returns all the text on the clipboard as one big string. If we used the “List of Lists of Lists” example, the string stored in `text` would look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

The `\n` newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard. There are many “lines” in this one string

value. You want to add a star to the start of each of these lines.

You could write code that searches for each `\n` newline character in the string and then adds the star just after that. But it would be easier to use the `split()` method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Make your program look like the following:

```
#!/python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)): # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in `lines` and then loop through the items in `lines`. For each line, we add a star and a space to the start of the line. Now each string in `lines` begins with a star.

Step 3: Join the Modified Lines

The `lines` list now contains modified lines that start with stars. But `pyperclip.copy()` is expecting a single string value, however, not a list of string values. To make this single string value, pass `lines` into the `join()` method to get a single string joined from the list's strings. Make your program look like the following:

```
#!/python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
```

```
lines = text.split('\n')
for i in range(len(lines)): # loop through all indexes for "lines" list
    lines[i] = '*' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don't need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.

A SHORT PROGRAM: PIG LATIN

Pig Latin is a silly made-up language that alters English words. If a word begins with a vowel, the word *yay* is added to the end of it. If a word begins with a consonant or consonant cluster (like *ch* or *gr*), that consonant or cluster is moved to the end of the word followed by *ay*.

Let's write a Pig Latin program that will output something like this:

Enter the English message to translate into Pig Latin:

My name is AL SWEIGART and I am 4,000 years old.

Ymay amenay isyay ALYAY EIGARTSWAY andyay Iyay amyay 4,000 yearsyay oldyay.

This program works by altering a string using the methods introduced in this chapter. Type the following source code into the file editor, and save the file as *pigLat.py*:

```
# English to Pig Latin
print('Enter the English message to translate into Pig Latin:')
message = input()

VOWELS = ('a', 'e', 'i', 'o', 'u', 'y')

pigLatin = [] # A list of the words in Pig Latin.
for word in message.split():
    # Separate the non-letters at the start of this word:
    prefixNonLetters = ""
    while len(word) > 0 and not word[0].isalpha():
```

```

    prefixNonLetters += word[0]
    word = word[1:]
if len(word) == 0:
    pigLatin.append(prefixNonLetters)
    continue

# Separate the non-letters at the end of this word:
suffixNonLetters = ""
while not word[-1].isalpha():
    suffixNonLetters += word[-1]
    word = word[:-1]

# Remember if the word was in uppercase or title case.
wasUpper = word.isupper()
wasTitle = word.istitle()

word = word.lower() # Make the word lowercase for translation.

# Separate the consonants at the start of this word:
prefixConsonants = ""
while len(word) > 0 and not word[0] in VOWELS:
    prefixConsonants += word[0]
    word = word[1:]

# Add the Pig Latin ending to the word:
if prefixConsonants != "":
    word += prefixConsonants + 'ay'
else:
    word += 'yay'

# Set the word back to uppercase or title case:
if wasUpper:
    word = word.upper()
if wasTitle:
    word = word.title()

# Add the non-letters back to the start or end of the word.
pigLatin.append(prefixNonLetters + word + suffixNonLetters)

```

```
# Join all the words back together into a single string:
```

```
print(' '.join(pigLatin))
```

Let's look at this code line by line, starting at the top:

```
# English to Pig Latin
```

```
print('Enter the English message to translate into Pig Latin:')
```

```
message = input()
```

```
VOWELS = ('a', 'e', 'i', 'o', 'u', 'y')
```

First, we ask the user to enter the English text to translate into Pig Latin. Also, we create a constant that holds every lowercase vowel letter (and *y*) as a tuple of strings. This will be used later in our program.

Next, we're going to create the `pigLatin` variable to store the words as we translate them into Pig Latin:

```
pigLatin = [] # A list of the words in Pig Latin.
```

```
for word in message.split():
```

```
    # Separate the non-letters at the start of this word:
```

```
    prefixNonLetters = ""
```

```
    while len(word) > 0 and not word[0].isalpha():
```

```
        prefixNonLetters += word[0]
```

```
        word = word[1:]
```

```
    if len(word) == 0:
```

```
        pigLatin.append(prefixNonLetters)
```

```
        continue
```

We need each word to be its own string, so we call `message.split()` to get a list of the words as separate strings. The string 'My name is AL SWEIGART and I am 4,000 years old.' would cause `split()` to return ['My', 'name', 'is', 'AL', 'SWEIGART', 'and', 'I', 'am', '4,000', 'years', 'old.'].

We need to remove any non-letters from the start and end of each word so that strings like 'old.' translate to 'oldyay.' instead of 'old.yay'. We'll save these non-letters to a variable named `prefixNonLetters`.

```
    # Separate the non-letters at the end of this word:
```

```
    suffixNonLetters = ""
```

```
    while not word[-1].isalpha():
```

```
        suffixNonLetters += word[-1]
```

```
        word = word[:-1]
```

A loop that calls `isalpha()` on the first character in the word will determine if we should remove a character from a word and concatenate it to the end of `prefixNonLetters`. If the entire word is made of non-letter characters, like '4,000', we can simply append it to the `pigLatin` list and continue to the next word to translate. We also need to save the non-letters at the end of the word string. This code is similar to the previous loop.

Next, we'll make sure the program remembers if the word was in uppercase or title case so we can restore it after translating the word to Pig Latin:

```
# Remember if the word was in uppercase or title case.
wasUpper = word.isupper()
wasTitle = word.istitle()

word = word.lower() # Make the word lowercase for translation.
```

For the rest of the code in the `for` loop, we'll work on a lowercase version of `word`.

To convert a word like *sweigart* to *eigart-sway*, we need to remove all of the consonants from the beginning of `word`:

```
# Separate the consonants at the start of this word:
prefixConsonants = ""
while len(word) > 0 and not word[0] in VOWELS:
    prefixConsonants += word[0]
    word = word[1:]
```

We use a loop similar to the loop that removed the non-letters from the start of `word`, except now we are pulling off consonants and storing them to a variable named `prefixConsonants`.

If there were any consonants at the start of the word, they are now in `prefixConsonants` and we should concatenate that variable and the string 'ay' to the end of `word`. Otherwise, we can assume `word` begins with a vowel and we only need to concatenate 'yay':

```
# Add the Pig Latin ending to the word:
if prefixConsonants != "":
    word += prefixConsonants + 'ay'
else:
    word += 'yay'
```

Recall that we set `word` to its lowercase version with `word = word.lower()`. If `word` was originally in uppercase or title case, this code will convert `word` back to its original case:

```
# Set the word back to uppercase or title case:
```

```
if wasUpper:
```

```
    word = word.upper()
```

```
if wasTitle:
```

```
    word = word.title()
```

At the end of the for loop, we append the word, along with any non-letter prefix or suffix it originally had, to the pigLatin list:

```
# Add the non-letters back to the start or end of the word.
```

```
pigLatin.append(prefixNonLetters + word + suffixNonLetters)
```

```
# Join all the words back together into a single string:
```

```
print(' '.join(pigLatin))
```

After this loop finishes, we combine the list of strings into a single string by calling the `join()` method. This single string is passed to `print()` to display our Pig Latin on the screen.

You can find other short, text-based Python programs like this one at <https://github.com/asweigart/pythonstdiogames/>.

SUMMARY

Text is a common form of data, and Python comes with many helpful string methods to process the text stored in string values. You will make use of indexing, slicing, and string methods in almost every Python program you write.

The programs you are writing now don't seem too sophisticated—they don't have graphical user interfaces with images and colorful text. So far, you're displaying text with `print()` and letting the user enter text with `input()`. However, the user can quickly enter large amounts of text through the clipboard. This ability provides a useful avenue for writing programs that manipulate massive amounts of text. These text-based programs might not have flashy windows or graphics, but they can get a lot of useful work done quickly.

Another way to manipulate large amounts of text is reading and writing files directly off the hard drive. You'll learn how to do this with Python in Chapter 9.

That just about covers all the basic concepts of Python programming! You'll continue to learn new concepts throughout the rest of this book, but you now know enough to start writing some useful programs that can automate tasks. If you'd like to see a collection of short, simple Python programs built from the basic concepts you've learned so far, check out <https://github.com/asweigart/pythonstdiogames/>. Try copying the source code for

each program by hand, and then make modifications to see how they affect the behavior of the program. Once you have an understanding of how the program works, try re-creating the program yourself from scratch. You don't need to re-create the source code exactly; just focus on what the program does rather than how it does it.

You might not think you have enough Python knowledge to do things such as download web pages, update spreadsheets, or send text messages, but that's where Python modules come in! These modules, written by other programmers, provide functions that make it easy for you to do all these things. So let's learn how to write real programs to do useful automated tasks.

PRACTICE QUESTIONS

1. What are escape characters?
2. What do the `\n` and `\t` escape characters represent?
3. How can you put a `\` backslash character in a string?
4. The string value "Howl's Moving Castle" is a valid string. Why isn't it a problem that the single quote character in the word Howl's isn't escaped?
5. If you don't want to put `\n` in your string, how can you write a string with newlines in it?
6. What do the following expressions evaluate to?
 - `'Hello, world!'[1]`
 - `'Hello, world!'[0:5]`
 - `'Hello, world!':5]`
 - `'Hello, world!':[3:]`
7. What do the following expressions evaluate to?
 - `'Hello'.upper()`
 - `'Hello'.upper().isupper()`
 - `'Hello'.upper().lower()`
8. What do the following expressions evaluate to?
 - `'Remember, remember, the fifth of November.'.split()`
 - `'-'.join('There can be only one.'.split())`
9. What string methods can you use to right-justify, left-justify, and center a string?

10. How can you trim whitespace characters from the beginning or end of a string?

PRACTICE PROJECTS

For practice, write programs that do the following.

Table Printer

Write a function named `printTable()` that takes a list of lists of strings and displays it in a well-organized table with each column right-justified. Assume that all the inner lists will contain the same number of strings. For example, the value could look like this:

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol', 'David'],
             ['dogs', 'cats', 'moose', 'goose']]
```

Your `printTable()` function would print the following:

```
apples Alice dogs
oranges Bob cats
cherries Carol moose
banana David goose
```

Hint: your code will first have to find the longest string in each of the inner lists so that the whole column can be wide enough to fit all the strings. You can store the maximum width of each column as a list of integers. The `printTable()` function can begin with `colWidths = [0] * len(tableData)`, which will create a list containing the same number of 0 values as the number of inner lists in `tableData`. That way, `colWidths[0]` can store the width of the longest string in `tableData[0]`, `colWidths[1]` can store the width of the longest string in `tableData[1]`, and so on. You can then find the largest value in the `colWidths` list to find out what integer width to pass to the `rjust()` string method.

Zombie Dice Bots

Programming games are a game genre where instead of playing a game directly, players write bot programs to play the game autonomously. I've created a *Zombie Dice* simulator, which allows programmers to practice their skills while making game-playing AIs. *Zombie Dice* bots can be simple or incredibly complex, and are great for a class exercise or an individual programming challenge.

Zombie Dice is a quick, fun dice game from Steve Jackson Games. The players are zombies trying to eat as many human brains as possible without getting shot three times. There is a cup of 13 dice with brains, footsteps, and shotgun icons on their faces. The

dice icons are colored, and each color has a different likelihood of each event occurring. Every die has two sides with footsteps, but dice with green icons have more sides with brains, red-icon dice have more shotguns, and yellow-icon dice have an even split of brains and shotguns. Do the following on each player's turn:

1. Place all 13 dice in the cup. The player randomly draws three dice from the cup and then rolls them. Players always roll exactly three dice.
2. They set aside and count up any brains (humans whose brains were eaten) and shotguns (humans who fought back). Accumulating three shotguns automatically ends a player's turn with zero points (regardless of how many brains they had). If they have between zero and two shotguns, they may continue rolling if they want. They may also choose to end their turn and collect one point per brain.
3. If the player decides to keep rolling, they must reroll all dice with footsteps. Remember that the player must always roll three dice; they must draw more dice out of the cup if they have fewer than three footsteps to roll. A player may keep rolling dice until either they get three shotguns—losing everything—or all 13 dice have been rolled. A player may not reroll only one or two dice, and may not stop mid-reroll.
4. When someone reaches 13 brains, the rest of the players finish out the round. The person with the most brains wins. If there's a tie, the tied players play one last tiebreaker round.

Zombie Dice has a push-your-luck game mechanic: the more you reroll the dice, the more brains you can get, but the more likely you'll eventually accrue three shotguns and lose everything. Once a player reaches 13 points, the rest of the players get one more turn (to potentially catch up) and the game ends. The player with the most points wins. You can find the complete rules at <https://github.com/asweigart/zombiedice/>.

Install the `zombiedice` module with `pip` by following the instructions in Appendix A. You can run a demo of the simulator with some pre-made bots by running the following in the interactive shell:

```
>>> importzombiedice
```

```
>>> zombiedice.demo()
```

```
Zombie Dice Visualization is running. Open your browser to http://
```

```
localhost:51810 to view it.
```

```
Press Ctrl-C to quit.
```

The program launches your web browser, which will look like Figure 6-1.



Figure 6-1: The web GUI for the Zombie Dice simulator

You'll create bots by writing a class with a `turn()` method, which is called by the simulator when it's your bot's turn to roll the dice. Classes are beyond the scope of this book, so the class code is already set up for you in the `myzombie.py` program, which is in the downloadable ZIP file for this book at <https://nostarch.com/automatestuff2/>. Writing a method is essentially the same as writing a function, and you can use the `turn()` code in the `myZombie.py` program as a template. Inside this `turn()` method, you'll call the `zombiedice.roll()` function as often as you want your bot to roll the dice.

```
import zombiedice

class MyZombie:
    def __init__(self, name):
        # All zombies must have a name:
        self.name = name

    def turn(self, gameState):
        # gameState is a dict with info about the current state of the game.
        # You can choose to ignore it in your code.

        diceRollResults = zombiedice.roll() # first roll
        # roll() returns a dictionary with keys 'brains', 'shotgun', and
        # 'footsteps' with how many rolls of each type there were.
        # The 'rolls' key is a list of (color, icon) tuples with the
```

```

# exact roll result information.
# Example of a roll() return value:
# {'brains': 1, 'footsteps': 1, 'shotgun': 1,
# 'rolls': [(('yellow', 'brains'), ('red', 'footsteps'),
#           ('green', 'shotgun'))]}

# REPLACE THIS ZOMBIE CODE WITH YOUR OWN:
brains = 0
while diceRollResults is not None:
    brains += diceRollResults['brains']

    if brains < 2:
        diceRollResults = zombiedice.roll() # roll again
    else:
        break

zombies = (
    zombiedice.examples.RandomCoinFlipZombie(name='Random'),
    zombiedice.examples.RollsUntilInTheLeadZombie(name='Until Leading'),
    zombiedice.examples.MinNumShotgunsThenStopsZombie(name='Stop at 2
Shotguns', minShotguns=2),
    zombiedice.examples.MinNumShotgunsThenStopsZombie(name='Stop at 1
Shotgun', minShotguns=1),
    MyZombie(name='My Zombie Bot'),
    # Add any other zombie players here.
)

# Uncomment one of the following lines to run in CLI or Web GUI mode:
#zombiedice.runTournament(zombies=zombies, numGames=1000)
zombiedice.runWebGui(zombies=zombies, numGames=1000)

```

The `turn()` method takes two parameters: `self` and `gameState`. You can ignore these in your first few zombie bots and consult the online documentation for details later if you want to learn more. The `turn()` method should call `zombiedice.roll()` at least once for the initial roll. Then, depending on the strategy the bot uses, it can call `zombiedice.roll()` again as many times as it wants. In `myZombie.py`, the `turn()` method calls `zombiedice.roll()` twice, which means the zombie bot will always roll its dice two times per turn regardless of the results of the roll.

The return value of `zombiedice.roll()` tells your code the results of the dice roll. It is a dictionary with four keys. Three of the keys, 'shotgun', 'brains', and 'footsteps', have integer values of how many dice came up with those icons. The fourth 'rolls' key has a value that is a list of tuples for each die roll. The tuples contain two strings: the color of the die at index 0 and the icon rolled at index 1. Look at the code comments in the `turn()` method's definition for an example. If the bot has already rolled three shotguns, then `zombiedice.roll()` will return `None`.

Try writing some of your own bots to play Zombie Dice and see how they compare against the other bots. Specifically, try to create the following bots:

A bot that, after the first roll, randomly decides if it will continue or stop

A bot that stops rolling after it has rolled two brains

A bot that stops rolling after it has rolled two shotguns

A bot that initially decides it'll roll the dice one to four times, but will stop early if it rolls two shotguns

A bot that stops rolling after it has rolled more shotguns than brains

Run these bots through the simulator and see how they compare to each other. You can also examine the code of some premade bots at <https://github.com/asweigart/zombiedice/>. If you find yourself playing this game in the real world, you'll have the benefit of thousands of simulated games telling you that one of the best strategies is to simply stop once you've rolled two shotguns. But you could always try pressing your luck . . .

***PATTERN MATCHING WITH REGULAR
EXPRESSIONS***

CSE, HIT, Nidasochi

you live in the United States or Canada, you know it will be three digits, followed by a hyphen, and then four more digits (and optionally, a three-digit area code at the start). This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but 4,155,551,234 is not.

We also recognize all sorts of other text patterns every day: email addresses have @ symbols in the middle, US social security numbers have nine digits and two hyphens, website URLs often have periods and forward slashes, news headlines use title case, social media hashtags begin with # and contain no spaces, and more.

Regular expressions are helpful, but few non-programmers know about them even though most modern text editors and word processors, such as Microsoft Word or OpenOffice, have find and find-and-replace features that can search based on regular expressions. Regular expressions are huge time-savers, not just for software users but also for programmers. In fact, tech writer Cory Doctorow argues that we should be teaching regular expressions even before programming:

Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you're a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.¹

In this chapter, you'll start by writing a program to find text patterns *without* using regular expressions and then see how to use regular expressions to make the code much less bloated. I'll show you basic matching with regular expressions and then move on to some more powerful features, such as string substitution and creating your own character classes. Finally, at the end of the chapter, you'll write a program that can automatically extract phone numbers and email addresses from a block of text.

FINDING PATTERNS OF TEXT WITHOUT REGULAR EXPRESSIONS

Say you want to find an American phone number in a string. You know the pattern if you're American: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.

Let's use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either `True` or `False`. Open a new file editor tab and enter the following code; then save the file as `isPhoneNumber.py`:

```
def isPhoneNumber(text):
```

```
    ● if len(text) != 12:
```

```

        return False
    ❸ if text[3] != '-':
        return False
    for i in range(4, 7):
        ❹ if not text[i].isdecimal():
            return False
    ❺ if text[7] != '-':
        return False
    for i in range(8, 12):
        ❻ if not text[i].isdecimal():
            return False
    ❼ return True

```

```

print('Is 415-555-4242 a phone number?')
print(isPhoneNumber('415-555-4242'))
print('Is Moshi moshi a phone number?')
print(isPhoneNumber('Moshi moshi'))

```

When this program is run, the output looks like this:

```

Is 415-555-4242 a phone number?
True
Is Moshi moshi a phone number?
False

```

The `isPhoneNumber()` function has code that does several checks to see whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`. First the code checks that the string is exactly 12 characters ❶. Then it checks that the area code (that is, the first three characters in `text`) consists of only numeric characters ❷. The rest of the function checks that the string follows the pattern of a phone number: the number must have the first hyphen after the area code ❸, three more numeric characters ❹, then another hyphen ❺, and finally four more numbers ❻. If the program execution manages to get past all the checks, it returns `True` ❼.

Calling `isPhoneNumber()` with the argument `'415-555-4242'` will return `True`. Calling `isPhoneNumber()` with `'Moshi moshi'` will return `False`; the first test fails because `'Moshi moshi'` is not 12 characters long.

If you wanted to find a phone number within a larger string, you would have to add even more code to find the phone number pattern. Replace the last four `print()` function calls in `isPhoneNumber.py` with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
```

```
for i in range(len(message)):
```

```
    ❶ chunk = message[i:i+12]
```

```
    ❷ if isPhoneNumber(chunk):
```

```
        print('Phone number found: ' + chunk)
```

```
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
```

```
Phone number found: 415-555-9999
```

```
Done
```

On each iteration of the for loop, a new chunk of 12 characters from message is assigned to the variable chunk ❶. For example, on the first iteration, i is 0, and chunk is assigned message[0:12] (that is, the string 'Call me at 4'). On the next iteration, i is 1, and chunk is assigned message[1:13] (the string 'all me at 41'). In other words, on each iteration of the for loop, chunk takes on the following values:

- 'Call me at 4'
- 'all me at 41'
- 'll me at 415'
- 'l me at 415-'
- . . . and so on.

You pass chunk to isPhoneNumber() to see whether it matches the phone number pattern ❷, and if so, you print the chunk.

Continue to loop through message, and eventually the 12 characters in chunk will be a phone number. The loop goes through the entire string, testing each 12-character piece and printing any chunk it finds that satisfies isPhoneNumber(). Once we're done going through message, we print Done.

While the string in message is short in this example, it could be millions of characters long and the program would still run in less than a second. A similar program that finds phone numbers using regular expressions would also run in less than a second, but regular expressions make it quicker to write these programs.

FINDING PATTERNS OF TEXT WITH REGULAR EXPRESSIONS

The previous phone number–finding program works, but it uses a lot of code to do something limited: the `isPhoneNumber()` function is 17 lines but can find only one pattern of phone numbers. What about a phone number formatted like 415.555.4242 or (415) 555-4242? What if the phone number had an extension, like 415-555-4242 x99? The `isPhoneNumber()` function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.

Regular expressions, called *regexes* for short, are descriptions for a pattern of text. For example, a `\d` in a regex stands for a digit character—that is, any single numeral from 0 to 9. The regex `\d\d\d-\d\d\d-\d\d\d\d` is used by Python to match the same text pattern the previous `isPhoneNumber()` function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `\d\d\d-\d\d\d-\d\d\d\d` regex.

But regular expressions can be much more sophisticated. For example, adding a 3 in braces (`{3}`) after a pattern is like saying, “Match this pattern three times.” So the slightly shorter regex `\d{3}-\d{3}-\d{4}` also matches the correct phone number format.

Creating Regex Objects

All the regex functions in Python are in the `re` module. Enter the following into the interactive shell to import this module:

```
>>> import re
```

NOTE

Most of the examples in this chapter will require the `re` module, so remember to import it at the beginning of any script you write or any time you restart Mu. Otherwise, you'll get a `NameError: name 're' is not defined` error message.

Passing a string value representing your regular expression to `re.compile()` returns a Regex pattern object (or simply, a Regex object).

To create a Regex object that matches the phone number pattern, enter the following into the interactive shell. (Remember that `\d` means “a digit character” and `\d\d\d-\d\d\d-\d\d\d\d` is the regular expression for a phone number pattern.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the `phoneNumRegex` variable contains a Regex object.

Matching Regex Objects

A `Regex` object's `search()` method searches the string it is passed for any matches to the regex. The `search()` method will return `None` if the regex pattern is not found in the string. If the pattern *is* found, the `search()` method returns a `Match` object, which have a `group()` method that will return the actual matched text from the searched string. (I'll explain groups shortly.) For example, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

The `mo` variable name is just a generic name to use for `Match` objects. This example might seem complicated at first, but it is much shorter than the earlier `isPhoneNumber.py` program and does the same thing.

Here, we pass our desired pattern to `re.compile()` and store the resulting `Regex` object in `phoneNumRegex`. Then we call `search()` on `phoneNumRegex` and pass `search()` the string we want to match for during the search. The result of the search gets stored in the variable `mo`. In this example, we know that our pattern will be found in the string, so we know that a `Match` object will be returned. Knowing that `mo` contains a `Match` object and not the null value `None`, we can call `group()` on `mo` to return the match. Writing `mo.group()` inside our `print()` function call displays the whole match, `415-555-4242`.

Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the `regex` module with `import re`.
2. Create a `Regex` object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the `Regex` object's `search()` method. This returns a `Match` object.
4. Call the `Match` object's `group()` method to return a string of the actual matched text.

NOTE

While I encourage you to enter the example code into the interactive shell, you should also make use of web-based regular expression testers, which can show you exactly how a regex matches a piece of text that you enter. I recommend the tester at <https://pythex.org/>.

MORE PATTERN MATCHING WITH REGULAR EXPRESSIONS

Now that you know the basic steps for creating and finding regular expression objects using Python, you're ready to try some of their more powerful pattern-matching capabilities.

Grouping with Parentheses

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create *groups* in the regex: `(\d\d\d)-(\d\d\d-\d\d\d)`. Then you can use the `group()` match object method to grab the matching text from just one group.

The first set of parentheses in a regex string will be group 1. The second set will be group 2. By passing the *integer* 1 or 2 to the `group()` match object method, you can grab different parts of the matched text. Passing 0 or nothing to the `group()` method will return the entire matched text. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `areaCode, mainNumber = mo.groups()` line.

Parentheses have a special meaning in regular expressions, but what do you do if you need to match a parenthesis in your text? For instance, maybe the phone numbers you are trying to match have the area code set in parentheses. In this case, you need to escape the (and) characters with a backslash. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

The \ (and \) escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters. In regular expressions, the following characters have special meanings:

```
. ^ $ * + ? { } [ ] \ | ( )
```

If you want to detect these characters as part of your text pattern, you need to escape them with a backslash:

```
\. \^ \$ \* \+ \? \{ \} \[ \] \| \\\ ( \)
```

Make sure to double-check that you haven't mistaken escaped parentheses \ (and \) for parentheses (and) in a regular expression. If you receive an error message about “missing)” or “unbalanced parenthesis,” you may have forgotten to include the closing unescaped parenthesis for a group, like in this example:

```
>>> re.compile(r'\(Parentheses\))')
Traceback (most recent call last):
  --snip--
re.error: missing ), unterminated subpattern at position 0
```

The error message tells you that there is an opening parenthesis at index 0 of the `r'(Parentheses\))'` string that is missing its corresponding closing parenthesis.

Matching Multiple Groups with the Pipe

The | character is called a *pipe*. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'.

When *both* Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object. Enter the following into the interactive shell:

```
>>> heroRegex = re.compile(r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman')
>>> mo2.group()
'Tina Fey'
```

NOTE

You can find all matching occurrences with the `findall()` method that's discussed in "The `findall()` Method" on page 171.

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once. This can be done with parentheses. Enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

The method call `mo.group()` returns the full matched text 'Batmobile', while `mo.group(1)` returns just the part of the matched text inside the first parentheses group, 'mobile'. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like `\|`.

Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match regardless of whether that bit of text is there. The `?` character flags the group that precedes it as an optional part of the pattern. For example, enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

The `(wo)?` part of the regular expression means that the pattern `wo` is an optional group. The regex will match text that has zero instances or one instance of `wo` in it. This is why the regex matches both 'Batwoman' and 'Batman'.

Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code. Enter the following into the interactive shell:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'

>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

You can think of the `?` as saying, “Match zero or one of the group preceding this question mark.”

If you need to match an actual question mark character, escape it with `\?`.

Matching Zero or More with the Star

The `*` (called the *star* or *asterisk*) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Let’s look at the Batman example again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

For 'Batman', the (wo)* part of the regex matches zero instances of wo in the string; for 'Batwoman', the (wo)* matches one instance of wo; and for 'Batwowowowoman', (wo)* matches four instances of wo.

If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

Matching One or More with the Plus

While * means “match zero or more,” the + (or *plus*) means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear *at least once*. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

The regex `Bat(wo)+man` will not match the string 'The Adventures of Batman', because at least one `wo` is required by the plus sign.

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: `\+`.

Matching Specific Repetitions with Braces

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in braces. For example, the regex `(Ha){3}` will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the `(Ha)` group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the braces. For example, the regex `(Ha){3,5}` will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

You can also leave out the first or second number in the braces to leave the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the `(Ha)` group, while `(Ha){,5}` will match zero to five instances. Braces can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

And these two regular expressions also match identical patterns:

```
(Ha){3,5}
((Ha)(Ha)(Ha))((Ha)(Ha)(Ha)(Ha))((Ha)(Ha)(Ha)(Ha)(Ha))
```

Enter the following into the interactive shell:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Here, `(Ha){3}` matches 'HaHaHa' but not 'Ha'. Since it doesn't match 'Ha', `search()` returns `None`.

GREEDY AND NON-GREEDY MATCHING

Since `(Ha){3,5}` can match three, four, or five instances of `Ha` in the string `'HaHaHaHaHa'`, you may wonder why the `Match` object's call to `group()` in the previous brace example returns `'HaHaHaHaHa'` instead of the shorter possibilities. After all, `'HaHaHa'` and `'HaHaHaHa'` are also valid matches of the regular expression `(Ha){3,5}`.

Python's regular expressions are *greedy* by default, which means that in ambiguous situations they will match the longest string possible. The *non-greedy* (also called *lazy*) version of the braces, which matches the shortest string possible, has the closing brace followed by a question mark.

Enter the following into the interactive shell, and notice the difference between the greedy and non-greedy forms of the braces searching the same string:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a non-greedy match or flagging an optional group. These meanings are entirely unrelated.

THE FINDALL() METHOD

In addition to the `search()` method, `Regex` objects also have a `findall()` method. While `search()` will return a `Match` object of the *first* matched text in the searched string, the `findall()` method will return the strings of *every* match in the searched string. To see how `search()` returns a `Match` object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

On the other hand, `findall()` will not return a `Match` object but a list of strings—as long as there are no groups in the regular expression. Each string in the list is a piece of the searched text that matched the regular expression. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then `findall()` will return a list of tuples. Each tuple represents a found match, and its items are the matched strings for each group in the regex. To see `findall()` in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[( '415', '555', '9999'), ('212', '555', '0000')]
```

To summarize what the `findall()` method returns, remember the following:

- When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d\d`, the method `findall()` returns a list of string matches, such as `['415-555-9999', '212-555-0000']`.
- When called on a regex that has groups, such as `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, the method `findall()` returns a list of tuples of strings (one string for each group), such as `[('415', '555', '9999'), ('212', '555', '0000')]`.

CHARACTER CLASSES

In the earlier phone number regex example, you learned that `\d` could stand for any numeric digit. That is, `\d` is shorthand for the regular expression `(0|1|2|3|4|5|6|7|8|9)`. There are many such *shorthand character classes*, as shown in Table 7-1.

Table 7-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents
<code>\d</code>	Any numeric digit from 0 to 9.
<code>\D</code>	Any character that is <i>not</i> a numeric digit from 0 to 9.

Shorthand character class	Represents
<code>\w</code>	Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)
<code>\W</code>	Any character that is <i>not</i> a letter, numeric digit, or the underscore character.
<code>\s</code>	Any space, tab, or newline character. (Think of this as matching “space” characters.)
<code>\S</code>	Any character that is <i>not</i> a space, tab, or newline.

Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers 0 to 5; this is much shorter than typing `(0|1|2|3|4|5)`. Note that while `\d` matches digits and `\w` matches digits, letters, and the underscore, there is no shorthand character class that matches only letters. (Though you can use the `[a-zA-Z]` character class, as explained next.)

For example, enter the following into the interactive shell:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression `\d+\s\w+` will match text that has one or more numeric digits (`\d+`), followed by a whitespace character (`\s`), followed by one or more letter/digit/underscore characters (`\w+`). The `findall()` method returns all matching strings of the regex pattern in a list.

MAKING YOUR OWN CHARACTER CLASSES

There are times when you want to match a set of characters but the shorthand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. Enter the following into the interactive shell:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the `.`, `*`, `?`, or `()` characters with a preceding backslash. For example, the character class `[0-5.]` will match digits 0 to 5 and a period. You do not need to write it as `[0-5\.]`.

By placing a caret character (`^`) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class. For example, enter the following into the interactive shell:

```
>>> consonantRegex = re.compile(r'^aeiouAEIOU')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'C', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', ' ', ' ',
', 'B', 'B', 'Y', ' ', 'F', 'D', ', '']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

THE CARET AND DOLLAR SIGN CHARACTERS

You can also use the caret symbol (`^`) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text. Likewise, you can put a dollar sign (`$`) at the end of the regex to indicate the string must *end* with this regex pattern. And you can use the `^` and `$` together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example, the `r'^Hello'` regular expression string matches strings that begin with 'Hello'. Enter the following into the interactive shell:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello, world!')
<re.Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9. Enter the following into the interactive shell:


```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<re.Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<re.Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

The last two `search()` calls in the previous interactive shell example demonstrate how the entire string must match the regex if `^` and `$` are used.

I always confuse the meanings of these two symbols, so I use the mnemonic “Carrots cost dollars” to remind myself that the caret comes first and the dollar sign comes last.

THE WILDCARD CHARACTER

The `.` (or *dot*) character in a regular expression is called a *wildcard* and will match any character except for a newline. For example, enter the following into the interactive shell:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Remember that the dot character will match just one character, which is why the match for the text `flat` in the previous example matched only `lat`. To match an actual dot, escape the dot with a backslash: `\.`

Matching Everything with Dot-Star

Sometimes you will want to match everything and anything. For example, say you want to match the string `'First Name:'`, followed by any and all text, followed by `'Last Name:'`, and then followed by anything again. You can use the dot-star (`.*`) to stand in for that

“anything.” Remember that the dot character means “any single character except the newline,” and the star character means “zero or more of the preceding character.”

Enter the following into the interactive shell:

```
>>> nameRegex = re.compile(r'First Name: (.*?) Last Name: (.*?)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

The dot-star uses *greedy* mode: It will always try to match as much text as possible. To match any and all text in a *non-greedy* fashion, use the dot, star, and question mark (`.*?`). Like with braces, the question mark tells Python to match in a non-greedy way.

Enter the following into the interactive shell to see the difference between the greedy and non-greedy versions:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Both regexes roughly translate to “Match an opening angle bracket, followed by anything, followed by a closing angle bracket.” But the string `<To serve man> for dinner.>` has two possible matches for the closing angle bracket. In the non-greedy version of the regex, Python matches the shortest possible string: `<To serve man>`. In the greedy version, Python matches the longest possible string: `<To serve man> for dinner.>`.

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

Enter the following into the interactive shell:

```

>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'

```

The regex `noNewlineRegex`, which did not have `re.DOTALL` passed to the `re.compile()` call that created it, will match everything only up to the first newline character, whereas `newlineRegex`, which *did* have `re.DOTALL` passed to `re.compile()`, matches everything. This is why the `newlineRegex.search()` call matches the full string, including its newline characters.

REVIEW OF REGEX SYMBOLS

This chapter covered a lot of notation, so here's a quick review of what you learned about basic regular expression syntax:

- The `?` matches zero or one of the preceding group.
- The `*` matches zero or more of the preceding group.
- The `+` matches one or more of the preceding group.
- The `{n}` matches exactly *n* of the preceding group.
- The `{n,}` matches *n* or more of the preceding group.
- The `{,m}` matches 0 to *m* of the preceding group.
- The `{n,m}` matches at least *n* and at most *m* of the preceding group.
- `{n,m}?` or `*?` or `+?` performs a non-greedy match of the preceding group.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively.
- `[abc]` matches any character between the brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the brackets.

CASE-INSENSITIVE MATCHING

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`. Enter the following into the interactive shell:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

SUBSTITUTING STRINGS WITH THE SUB() METHOD

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The `sub()` method for `Regex` objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The `sub()` method returns a string with the substitutions applied.

For example, enter the following into the interactive shell:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can type `\1`, `\2`, `\3`, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.”

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1*****'` as the first argument to `sub()`. The `\1` in that string will be replaced by whatever text was matched by group 1—that is, the `(\w)` group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1*****', 'Agent Alice told Agent Carol that Agent
Eve knew Agent Bob was a double agent.')
A***** told C***** that E***** knew B***** was a double agent.'
```

MANAGING COMPLEX REGEXES

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. This “verbose mode” can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3})|(\d{3}\d{3}))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
\s*(ext|x|ext.)\s*\d{2,5}?)')
```

you can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''
(\d{3})|(\d{3}\d{3})?      # area code
(\s|-|\.)?                # separator
\d{3}                      # first 3 digits
(\s|-|\.)                  # separator
\d{4}                      # last 4 digits
\s*(ext|x|ext.)\s*\d{2,5}? # extension
''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (`'''`) to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as regular Python code: the `#` symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part

of the text pattern to be matched. This lets you organize the regular expression so it's easier to read.

COMBINING `re.IGNORECASE`, `re.DOTALL`, AND `re.VERBOSE`

What if you want to use `re.VERBOSE` to write comments in your regular expression but also want to use `re.IGNORECASE` to ignore capitalization? Unfortunately, the `re.compile()` function takes only a single value as its second argument. You can get around this limitation by combining the `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE` variables using the pipe character (`|`), which in this context is known as the *bitwise or* operator.

So if you want a regular expression that's case-insensitive *and* includes newlines to match the dot character, you would form your `re.compile()` call like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

Including all three options in the second argument will look like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at <https://nostarch.com/automatestuff2/> for more information. You can also pass other options for the second argument; they're uncommon, but you can read more about them in the resources, too.

PROJECT: PHONE NUMBER AND EMAIL ADDRESS EXTRACTOR

Say you have the boring task of finding every phone number and email address in a long web page or document. If you manually scroll through the page, you might end up searching for a long time. But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press `CTRL-A` to select all the text, press `CTRL-C` to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

Whenever you're tackling a new project, it can be tempting to dive right into writing code. But more often than not, it's best to take a step back and consider the bigger picture. I recommend first drawing up a high-level plan for what your program needs to do. Don't think about the actual code yet—you can worry about that later. Right now, stick to broad strokes.

For example, your phone and email address extractor will need to do the following:

1. Get the text off the clipboard.
2. Find all phone numbers and email addresses in the text.
3. Paste them onto the clipboard.

Now you can start thinking about how this might work in code. The code will need to do the following:

1. Use the `pyperclip` module to copy and paste strings.
2. Create two regexes, one for matching phone numbers and the other for matching email addresses.
3. Find all matches, not just the first match, of both regexes.
4. Neatly format the matched strings into a single string to paste.
5. Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately. Each step is fairly manageable and expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

First, you have to create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as `phoneAndEmail.py`:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r"(
    (\d{3}|(\d{3}\))?          # area code
    (\s|-|\.)?                # separator
    (\d{3})                    # first 3 digits
    (\s|-|\.)                 # separator
    (\d{4})                    # last 4 digits
    (\s*(ext|x|ext.)\s*(\d{2,5}))? # extension
)", re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.
```

TODO: Copy results to the clipboard.

The TODO comments are just a skeleton for the program. They'll be replaced as you write the actual code.

The phone number begins with an *optional* area code, so the area code group is followed with a question mark. Since the area code can be just three digits (that is, `\d{3}`) or three digits within parentheses (that is, `\(\d{3}\)`), you should have a pipe joining those parts. You can add the regex comment `# Area code` to this part of the multiline string to help you remember what `(\d{3}|\(\d{3}\))?` is supposed to match.

The phone number separator character can be a space (`\s`), hyphen (`-`), or period (`.`), so these parts should also be joined by pipes. The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an optional extension made up of any number of spaces followed by `ext`, `x`, or `ext.`, followed by two to five digits.

NOTE

It's easy to get mixed up with regular expressions that contain groups with parentheses () and escaped parentheses \(\). Remember to double-check that you're using the correct one if you get a "missing), unterminated subpattern" error message.

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''
--snip--

# Create email regex.
emailRegex = re.compile(r'''
    ❶ [a-zA-Z0-9._%+-]+ # username
    ❷ @ # @ symbol
```



```
③ [a-zA-Z0-9.-]+ # domain name
(\\.[a-zA-Z]{2,4}) # dot-something
)''', re.VERBOSE)
```

```
# TODO: Find matches in clipboard text.
```

```
# TODO: Copy results to the clipboard.
```

The username part of the email address ① is one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen. You can put all of these into a character class: `[a-zA-Z0-9._%+@-]`.

The domain and username are separated by an `@` symbol ②. The domain name ③ has a slightly less permissive character class with only letters, numbers, periods, and hyphens: `[a-zA-Z0-9.-]`. And last will be the “dot-com” part (technically known as the *top-level domain*), which can really be dot-anything. This is between two and four characters.

The format for email addresses has a lot of weird rules. This regular expression won't match every possible valid email address, but it'll match almost any typical email address you'll encounter.

Step 3: Find All Matches in the Clipboard Text

Now that you have specified the regular expressions for phone numbers and email addresses, you can let Python's `re` module do the hard work of finding all the matches on the clipboard. The `pyperclip.paste()` function will get a string value of the text on the clipboard, and the `findall()` regex method will return a list of tuples.

Make your program look like the following:

```
#!/python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''
--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())
```

```
❶ matches = []
❷ for groups in phoneRegex.findall(text):
    phoneNum = '-'.join([groups[1], groups[3], groups[5]])
    if groups[8] != '':
        phoneNum += ' x' + groups[8]
    matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
    matches.append(groups[0])
```

```
# TODO: Copy results to the clipboard.
```

There is one tuple for each match, and each tuple contains strings for each group in the regular expression. Remember that group 0 matches the entire regular expression, so the group at index 0 of the tuple is the one you are interested in.

As you can see at ❶, you'll store the matches in a list variable named `matches`. It starts off as an empty list, and a couple for loops. For the email addresses, you append group 0 of each match ❸. For the matched phone numbers, you don't want to just append group 0. While the program *detects* phone numbers in several formats, you want the phone number appended to be in a single, standard format. The `phoneNum` variable contains a string built from groups 1, 3, 5, and 8 of the matched text ❷. (These groups are the area code, first three digits, last four digits, and extension.)

Step 4: Join the Matches into a String for the Clipboard

Now that you have the email addresses and phone numbers as a list of strings in `matches`, you want to put them on the clipboard. The `pyperclip.copy()` function takes only a single string value, not a list of strings, so you call the `join()` method on `matches`.

To make it easier to see that the program is working, let's print any matches you find to the terminal. If no phone numbers or email addresses were found, the program should tell the user this.

Make your program look like the following:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

--snip--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
```

```
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

Running the Program

For an example, open your web browser to the No Starch Press contact page at <https://nostarch.com/contactus/>, press CTRL-A to select all the text on the page, and press CTRL-C to copy it to the clipboard. When you run this program, the output will look something like this:

```
Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
info@nostarch.com
```

Ideas for Similar Programs

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications. For example, you could:

- Find website URLs that begin with `http://` or `https://`.
- Clean up dates in different date formats (such as 3/14/2019, 03-14-2019, and 2015/3/19) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security or credit card numbers.
- Find common typos such as multiple spaces between words, accidentally accidentally repeated words, or multiple exclamation marks at the end of sentences. Those are annoying!!

SUMMARY

While a computer can search for text quickly, it must be told precisely what to look for. Regular expressions allow you to specify the pattern of characters you are looking for, rather than the exact text itself. In fact, some word processing and spreadsheet

applications provide find-and-replace features that allow you to search using regular expressions.

The `re` module that comes with Python lets you compile Regex objects. These objects have several methods: `search()` to find a single match, `findall()` to find all matching instances, and `sub()` to do a find-and-replace substitution of text.

You can find out more in the official Python documentation at <https://docs.python.org/3/library/re.html>. Another useful resource is the tutorial website <https://www.regular-expressions.info/>.

PRACTICE QUESTIONS

1. What is the function that creates Regex objects?
2. Why are raw strings often used when creating Regex objects?
3. What does the `search()` method return?
4. How do you get the actual strings that match the pattern from a Match object?
5. In the regex created from `r'(d\d\d)-(d\d\d\d\d\d\d)`, what does group 0 cover? Group 1? Group 2?
6. Parentheses and periods have specific meanings in regular expression syntax. How would you specify that you want a regex to match actual parentheses and period characters?
7. The `findall()` method returns a list of strings or a list of tuples of strings. What makes it return one or the other?
8. What does the `|` character signify in regular expressions?
9. What two things does the `?` character signify in regular expressions?
10. What is the difference between the `+` and `*` characters in regular expressions?
11. What is the difference between `{3}` and `{3,5}` in regular expressions?
12. What do the `\d`, `\w`, and `\s` shorthand character classes signify in regular expressions?
13. What do the `\D`, `\W`, and `\S` shorthand character classes signify in regular expressions?
14. What is the difference between `.*` and `.*??`
15. What is the character class syntax to match all numbers and lowercase letters?
16. How do you make a regular expression case-insensitive?
17. What does the `.` character normally match? What does it match if `re.DOTALL` is passed as the second argument to `re.compile()`?

18. If `numRegex = re.compile(r'\d+')`, what will `numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')` return?

19. What does passing `re.VERBOSE` as the second argument to `re.compile()` allow you to do?

20. How would you write a regex that matches a number with commas for every three digits? It must match the following:

- '42'
- '1,234'
- '6,368,745'

but not the following:

- '12,34,567' (which has only two digits between the commas)
- '1234' (which lacks commas)

21. How would you write a regex that matches the full name of someone whose last name is Watanabe? You can assume that the first name that comes before it will always be one word that begins with a capital letter. The regex must match the following:

- 'Haruto Watanabe'
- 'Alice Watanabe'
- 'RoboCop Watanabe'

but not the following:

- 'haruto Watanabe' (where the first name is not capitalized)
- 'Mr. Watanabe' (where the preceding word has a nonletter character)
- 'Watanabe' (which has no first name)
- 'Haruto watanabe' (where Watanabe is not capitalized)

22. How would you write a regex that matches a sentence where the first word is either *Alice*, *Bob*, or *Carol*; the second word is either *eats*, *pets*, or *throws*; the third word is *apples*, *cats*, or *baseballs*; and the sentence ends with a period? This regex should be case-insensitive. It must match the following:

- 'Alice eats apples.'
- 'Bob pets cats.'
- 'Carol throws baseballs.'

- 'Alice throws Apples.'
- 'BOB EATS CATS.'

but not the following:

- 'RoboCop eats apples.'
- 'ALICE THROWS FOOTBALLS.'
- 'Carol eats 7 cats.'

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Date Detection

Write a regular expression that can detect dates in the *DD/MM/YYYY* format. Assume that the days range from 01 to 31, the months range from 01 to 12, and the years range from 1000 to 2999. Note that if the day or month is a single digit, it'll have a leading zero.

The regular expression doesn't have to detect correct days for each month or for leap years; it will accept nonexistent dates like 31/02/2020 or 31/04/2021. Then store these strings into variables named *month*, *day*, and *year*, and write additional code that can detect if it is a valid date. April, June, September, and November have 30 days, February has 28 days, and the rest of the months have 31 days. February has 29 days in leap years. Leap years are every year evenly divisible by 4, except for years evenly divisible by 100, unless the year is also evenly divisible by 400. Note how this calculation makes it impossible to make a reasonably sized regular expression that can detect a valid date.

Strong Password Detection

Write a function that uses regular expressions to make sure the password string it is passed is strong. A strong password is defined as one that is at least eight characters long, contains both uppercase and lowercase characters, and has at least one digit. You may need to test the string against multiple regex patterns to validate its strength.

Regex Version of the strip() Method

Write a function that takes a string and does the same thing as the `strip()` string method. If no other arguments are passed other than the string to strip, then whitespace characters will be removed from the beginning and end of the string. Otherwise, the characters specified in the second argument to the function will be removed from the string.

READING AND WRITING FILES

CSE, HIT, Nidasoshi

FILES AND FILE PATHS

A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer. For example, there is a file on my Windows laptop with the filename *project.docx* in the path *C:\Users\Al\Documents*. The part of the filename after the last period is called the file's *extension* and tells you a file's type. The filename *project.docx* is a Word document, and *Users*, *Al*, and *Documents* all refer to *folders* (also called *directories*). Folders can contain files and other folders. For example, *project.docx* is in the *Documents* folder, which is inside the *Al* folder, which is inside the *Users* folder. Figure 9-1 shows this folder organization.

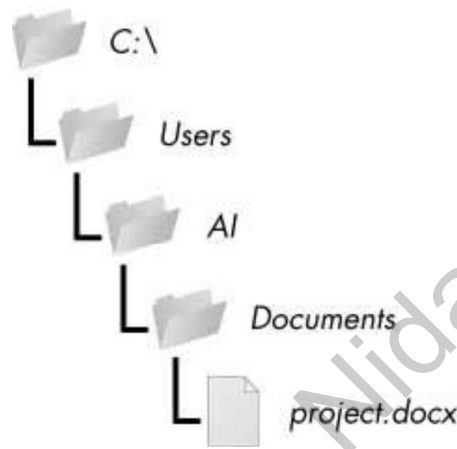


Figure 9-1: A file in a hierarchy of folders

The *C:* part of the path is the *root folder*, which contains all other folders. On Windows, the root folder is named *C:* and is also called the *C: drive*. On macOS and Linux, the root folder is */*. In this book, I'll use the Windows-style root folder, *C:*. If you are entering the interactive shell examples on macOS or Linux, enter */* instead.

Additional *volumes*, such as a DVD drive or USB flash drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as *D:* or *E:*. On macOS, they appear as new folders under the */Volumes* folder. On Linux, they appear as new folders under the */mnt* (“mount”) folder. Also note that while folder names and filenames are not case-sensitive on Windows and macOS, they are case-sensitive on Linux.

NOTE

Since your system probably has different files and folders on it than mine, you won't be able to follow every example in this chapter exactly. Still, try to follow along using folders that exist on your computer.

Backslash on Windows and Forward Slash on macOS and Linux

On Windows, paths are written using backslashes (\) as the separator between folder names. The macOS and Linux operating systems, however, use the forward slash (/) as their path separator. If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.

Fortunately, this is simple to do with the `Path()` function in the `pathlib` module. If you pass it the string values of individual file and folder names in your path, `Path()` will return a string with a file path using the correct path separators. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')

WindowsPath('spam/bacon/eggs')
>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

Note that the convention for importing `pathlib` is to run `from pathlib import Path`, since otherwise we'd have to enter `pathlib.Path` everywhere `Path` shows up in our code. Not only is this extra typing redundant, but it's also redundant.

I'm running this chapter's interactive shell examples on Windows, so `Path('spam', 'bacon', 'eggs')` returned a `WindowsPath` object for the joined path, represented as `WindowsPath('spam/bacon/eggs')`. Even though Windows uses backslashes, the `WindowsPath` representation in the interactive shell displays them using forward slashes, since open source software developers have historically favored the Linux operating system.

If you want to get a simple text string of this path, you can pass it to the `str()` function, which in our example returns `'spam\\bacon\\eggs'`. (Notice that the backslashes are doubled because each backslash needs to be escaped by another backslash character.) If I had called this function on, say, Linux, `Path()` would have returned a `PosixPath` object that, when passed to `str()`, would have returned `'spam/bacon/eggs'`. (*POSIX* is a set of standards for Unix-like operating systems such as Linux.)

These `Path` objects (really, `WindowsPath` or `PosixPath` objects, depending on your operating system) will be passed to several of the file-related functions introduced in this chapter. For example, the following code joins names from a list of filenames to the end of a folder's name:

```
>>> from pathlib import Path
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
```

```
print(Path(r'C:\Users\AI', filename))
```

```
C:\Users\AI\accounts.txt
```

```
C:\Users\AI\details.csv
```

```
C:\Users\AI\invite.docx
```

On Windows, the backslash separates directories, so you can't use it in filenames. However, you can use backslashes in filenames on macOS and Linux. So while `Path(r'spam\eggs')` refers to two separate folders (or a file *eggs* in a folder *spam*) on Windows, the same command would refer to a single folder (or file) named *spam\eggs* on macOS and Linux. For this reason, it's usually a good idea to always use forward slashes in your Python code (and I'll be doing so for the rest of this chapter). The `pathlib` module will ensure that it always works on all operating systems.

Note that `pathlib` was introduced in Python 3.4 to replace older `os.path` functions. The Python Standard Library modules support it as of Python 3.6, but if you are working with legacy Python 2 versions, I recommend using `pathlib2`, which gives you `pathlib`'s features on Python 2.7. Appendix A has instructions for installing `pathlib2` using `pip`. Whenever I've replaced an older `os.path` function with `pathlib`, I've made a short note. You can look up the older functions at <https://docs.python.org/3/library/os.path.html>.

Using the / Operator to Join Paths

We normally use the `+` operator to add two integer or floating-point numbers, such as in the expression `2 + 2`, which evaluates to the integer value 4. But we can also use the `+` operator to concatenate two string values, like the expression `'Hello' + 'World'`, which evaluates to the string value `'HelloWorld'`. Similarly, the `/` operator that we normally use for division can also combine `Path` objects and strings. This is helpful for modifying a `Path` object after you've already created it with the `Path()` function.

For example, enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Using the `/` operator with `Path` objects makes joining paths just as easy as string concatenation. It's also safer than using string concatenation or the `join()` method, like we do in this example:

```
>>> homeFolder = r'C:\Users\AI'
>>> subFolder = 'spam'
>>> homeFolder + '\\' + subFolder
'C:\Users\AI\spam'
>>> '\\'.join([homeFolder, subFolder])
'C:\Users\AI\spam'
```

A script that uses this code isn't safe, because its backslashes would only work on Windows. You could add an if statement that checks `sys.platform` (which contains a string describing the computer's operating system) to decide what kind of slash to use, but applying this custom code everywhere it's needed can be inconsistent and bug-prone.

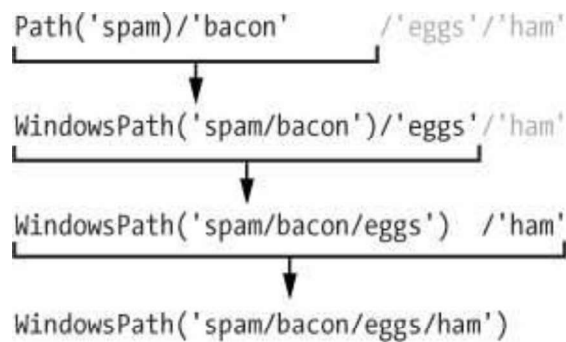
The `pathlib` module solves these problems by reusing the `/` math division operator to join paths correctly, no matter what operating system your code is running on. The following example uses this strategy to join the same paths as in the previous example:

```
>>> homeFolder = Path('C:/Users/AI')
>>> subFolder = Path('spam')
>>> homeFolder / subFolder
WindowsPath('C:/Users/AI/spam')
>>> str(homeFolder / subFolder)
'C:\Users\AI\spam'
```

The only thing you need to keep in mind when using the `/` operator for joining paths is that one of the first two values must be a `Path` object. Python will give you an error if you try entering the following into the interactive shell:

```
>>> 'spam' / 'bacon' / 'eggs'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Python evaluates the `/` operator from left to right and evaluates to a `Path` object, so either the first or second leftmost value must be a `Path` object for the entire expression to evaluate to a `Path` object. Here's how the `/` operator and a `Path` object evaluate to the final `Path` object.



If you see the `TypeError: unsupported operand type(s) for /: 'str' and 'str'` error message shown previously, you need to put a `Path` object on the left side of the expression.

The `/` operator replaces the older `os.path.join()` function, which you can learn more about from <https://docs.python.org/3/library/os.path.html#os.path.join>.

The Current Working Directory

Every program that runs on your computer has a *current working directory*, or *cwd*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.

NOTE

While folder is the more modern name for directory, note that current working directory (or just working directory) is the standard term, not “current working folder.”

You can get the current working directory as a string value with the `Path.cwd()` function and change it using `os.chdir()`. Enter the following into the interactive shell:

```

>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')

```

Here, the current working directory is set to `C:\Users\Al\AppData\Local\Programs\Python\Python37`, so the filename `project.docx` refers to `C:\Users\Al\AppData\Local\Programs\Python\Python37\project.docx`. When we change the current working directory to `C:\Windows\System32`, the filename `project.docx` is interpreted as `C:\Windows\System32\project.docx`.

Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:/ThisFolderDoesNotExist')
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

FileNotFoundError: [WinError 2] The system cannot find the file specified:

'C:/ThisFolderDoesNotExist'

There is no `pathlib` function for changing the working directory, because changing the current working directory while a program is running can often lead to subtle bugs.

The `os.getcwd()` function is the older way of getting the current working directory as a string.

The Home Directory

All users have a folder for their own files on the computer called the *home directory* or *home folder*. You can get a `Path` object of the home folder by calling `Path.home()`:

```
>>> Path.home()
```

```
WindowsPath('C:/Users/Al')
```

The home directories are located in a set place depending on your operating system:

- On Windows, home directories are under `C:\Users`.
- On Mac, home directories are under `/Users`.
- On Linux, home directories are often under `/home`.

Your scripts will almost certainly have permissions to read and write the files under your home directory, so it's an ideal place to put the files that your Python programs will work with.

Absolute vs. Relative Paths

There are two ways to specify a file path:

- An *absolute path*, which always begins with the root folder
- A *relative path*, which is relative to the program's current working directory

There are also the *dot* (`.`) and *dot-dot* (`..`) folders. These are not real folders but special names that can be used in a path. A single period (“dot”) for a folder name is shorthand for “this directory.” Two periods (“dot-dot”) means “the parent folder.”

Figure 9-2 is an example of some folders and files. When the current working directory is set to `C:\bacon`, the relative paths for the other folders and files are set as they are in the figure.

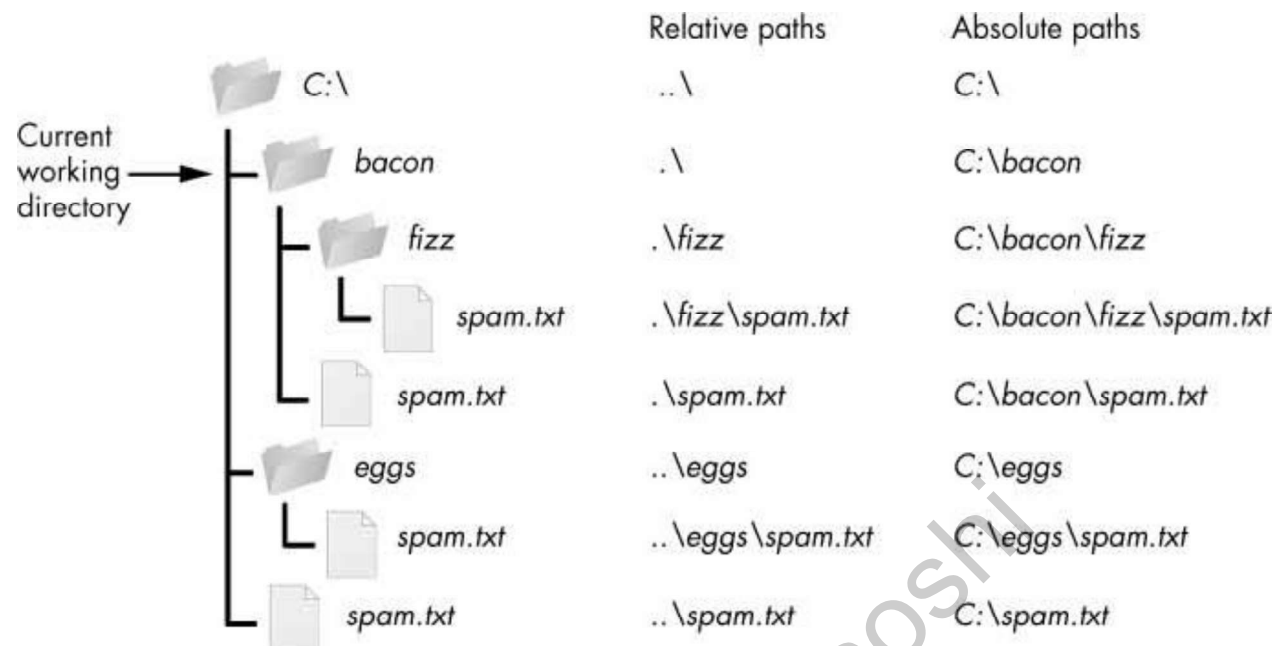


Figure 9-2: The relative paths for folders and files in the working directory `C:\bacon`

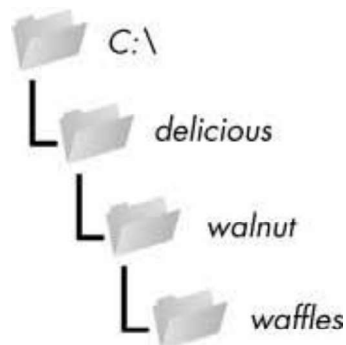
The `.\` at the start of a relative path is optional. For example, `.\spam.txt` and `spam.txt` refer to the same file.

Creating New Folders Using the `os.makedirs()` Function

Your programs can create new folders (directories) with the `os.makedirs()` function. Enter the following into the interactive shell:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

This will create not just the `C:\delicious` folder but also a `walnut` folder inside `C:\delicious` and a `waffles` folder inside `C:\delicious\walnut`. That is, `os.makedirs()` will create any necessary intermediate folders in order to ensure that the full path exists. Figure 9-3 shows this hierarchy of folders.



To make a directory from a Path object, call the `mkdir()` method. For example, this code will create a *spam* folder under the home folder on my computer:

```
>>> from pathlib import Path
>>> Path(r'C:\Users\AI\spam').mkdir()
```

Note that `mkdir()` can only make one directory at a time; it won't make several subdirectories at once like `os.makedirs()`.

Handling Absolute and Relative Paths

The `pathlib` module provides methods for checking whether a given path is an absolute path and returning the absolute path of a relative path.

Calling the `is_absolute()` method on a Path object will return `True` if it represents an absolute path or `False` if it represents a relative path. For example, enter the following into the interactive shell, using your own files and folders instead of the exact ones listed here:

```
>>> Path.cwd()
WindowsPath('C:/Users/AI/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().is_absolute()
True
>>> Path('spam/bacon/eggs').is_absolute()
False
```

To get an absolute path from a relative path, you can put `Path.cwd() /` in front of the relative Path object. After all, when we say “relative path,” we almost always mean a path that is relative to the current working directory. Enter the following into the interactive shell:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.cwd() / Path('my/relative/path')
WindowsPath('C:/Users/AI/AppData/Local/Programs/Python/Python37/my/relative/
path')
```

If your relative path is relative to another path besides the current working directory, just replace `Path.cwd()` with that other path instead. The following example gets an absolute path using the home directory instead of the current working directory:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.home()/ Path('my/relative/path')
WindowsPath('C:/Users/Al/my/relative/path')
```

The `os.path` module also has some useful functions related to absolute and relative paths:

- Calling `os.path.abspath(path)` will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
- Calling `os.path.isabs(path)` will return `True` if the argument is an absolute path and `False` if it is a relative path.
- Calling `os.path.relpath(path, start)` will return a string of a relative path from the `start` path to `path`. If `start` is not provided, the current working directory is used as the start path.

Try these functions in the interactive shell:

```
>>> os.path.abspath('.')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'
>>> os.path.abspath('..\\Scripts')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Since `C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37` was the working directory when `os.path.abspath()` was called, the “single-dot” folder represents the absolute path `'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'`.

Enter the following calls to `os.path.relpath()` into the interactive shell:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
```

When the relative path is within the same parent folder as the path, but is within subfolders of a different path, such as `'C:\\Windows'` and `'C:\\spam\\eggs'`, you can use the

“dot-dot” notation to return to the parent folder.

Getting the Parts of a File Path

Given a Path object, you can extract the file path’s different parts as strings using several Path object attributes. These can be useful for constructing new file paths based on existing ones. The attributes are diagrammed in Figure 9-4.

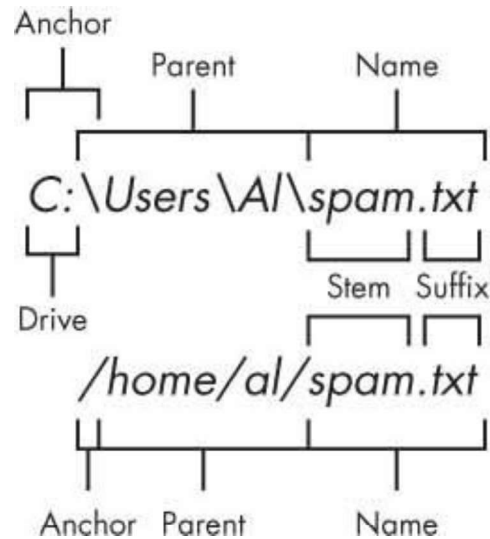


Figure 9-4: The parts of a Windows (top) and macOS/Linux (bottom) file path

The parts of a file path include the following:

- The *anchor*, which is the root folder of the filesystem
- On Windows, the *drive*, which is the single letter that often denotes a physical hard drive or other storage device
- The *parent*, which is the folder that contains the file
- The *name* of the file, made up of the *stem* (or *base name*) and the *suffix* (or *extension*)

Note that Windows Path objects have a drive attribute, but macOS and Linux Path objects don’t. The drive attribute doesn’t include the first backslash.

To extract each attribute from the file path, enter the following into the interactive shell:

```
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.anchor
'C:\'
>>> p.parent # This is a Path object, not a string.
WindowsPath('C:/Users/Al')
>>> p.name
'spam.txt'
```

```
>>> p.stem
'spam'
>>> p.suffix
'.txt'
>>> p.drive
'C:'
```

These attributes evaluate to simple string values, except for `parent`, which evaluates to another `Path` object.

The `parents` attribute (which is different from the `parent` attribute) evaluates to the ancestor folders of a `Path` object with an integer index:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().parents[0]
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python')
>>> Path.cwd().parents[1]
WindowsPath('C:/Users/Al/AppData/Local/Programs')
>>> Path.cwd().parents[2]
WindowsPath('C:/Users/Al/AppData/Local')
>>> Path.cwd().parents[3]
WindowsPath('C:/Users/Al/AppData')
>>> Path.cwd().parents[4]
WindowsPath('C:/Users/Al')
>>> Path.cwd().parents[5]
WindowsPath('C:/Users')
>>> Path.cwd().parents[6]
WindowsPath('C:/')
```

The older `os.path` module also has similar functions for getting the different parts of a path written in a string value. Calling `os.path.dirname(path)` will return a string of everything that comes before the last slash in the path argument. Calling `os.path.basename(path)` will return a string of everything that comes after the last slash in the path argument. The directory (or dir) name and base name of a path are outlined in Figure 9-5.



Figure 9-5: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

For example, enter the following into the interactive shell:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(calcFilePath)
'calc.exe'
>>> os.path.dirname(calcFilePath)
'C:\\Windows\\System32'
```

If you need a path's dir name and base name together, you can just call `os.path.split()` to get a tuple value with these two strings, like so:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Notice that you could create the same tuple by calling `os.path.dirname()` and `os.path.basename()` and placing their return values in a tuple:

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

But `os.path.split()` is a nice shortcut if you need both values.

Also, note that `os.path.split()` does *not* take a file path and return a list of strings of each folder. For that, use the `split()` string method and split on the string in `os.sep`. (Note that `sep` is in `os`, not `os.path`.) The `os.sep` variable is set to the correct folder-separating slash for the computer running the program, `\"` on Windows and `'/'` on macOS and Linux, and splitting on it will return a list of the individual folders.

For example, enter the following into the interactive shell:

```
>>> calcFilePath.split(os.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

This returns all the parts of the path as strings.

On macOS and Linux systems, the returned list of folders will begin with a blank string, like this:

```
>>> '/usr/bin'.split(os.sep)
['', 'usr', 'bin']
```

The `split()` string method will work to return a list of each part of the path.

Finding File Sizes and Folder Contents

Once you have ways of handling file paths, you can then start gathering information about specific files and folders. The `os.path` module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling `os.path.getsize(path)` will return the size in bytes of the file in the `path` argument.
- Calling `os.listdir(path)` will return a list of filename strings for each file in the `path` argument. (Note that this function is in the `os` module, not `os.path`.)

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
27648
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

As you can see, the `calc.exe` program on my computer is 27,648 bytes in size, and I have a lot of files in `C:\\Windows\\system32`. If I want to find the total size of all the files in this directory, I can use `os.path.getsize()` and `os.listdir()` together.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
    totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))
>>> print(totalSize)
2559970473
```

As I loop over each filename in the `C:\\Windows\\System32` folder, the `totalSize` variable is incremented by the size of each file. Notice how when I call `os.path.getsize()`, I use `os.path.join()` to join the folder name with the current filename. The integer that `os.path.getsize()` returns is added to the value of `totalSize`. After looping through all the files, I print `totalSize` to see the total size of the `C:\\Windows\\System32` folder.

Modifying a List of Files Using Glob Patterns

If you want to work on specific files, the `glob()` method is simpler to use than `listdir()`. Path objects have a `glob()` method for listing the contents of a folder according to a *glob*

pattern. Glob patterns are like a simplified form of regular expressions often used in command line commands. The `glob()` method returns a generator object (which are beyond the scope of this book) that you'll need to pass to `list()` to easily view in the interactive shell:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> p.glob('*')
<generator object Path.glob at 0x000002A6E389DED0>
>>> list(p.glob('*')) # Make a list from the generator.
[WindowsPath('C:/Users/Al/Desktop/1.png'), WindowsPath('C:/Users/Al/
Desktop/22-ap.pdf'), WindowsPath('C:/Users/Al/Desktop/cat.jpg'),
--snip--
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

The asterisk (*) stands for “multiple of any characters,” so `p.glob('*')` returns a generator of all files in the path stored in `p`.

Like with regexes, you can create complex expressions:

```
>>> list(p.glob('*.txt')) # Lists all text files.
[WindowsPath('C:/Users/Al/Desktop/foo.txt'),
--snip--
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

The glob pattern `*.txt` will return files that start with any combination of characters as long as it ends with the string `.txt`, which is the text file extension.

In contrast with the asterisk, the question mark (?) stands for any single character:

```
>>> list(p.glob('project?.docx'))
[WindowsPath('C:/Users/Al/Desktop/project1.docx'), WindowsPath('C:/Users/Al/
Desktop/project2.docx'),
--snip--
WindowsPath('C:/Users/Al/Desktop/project9.docx')]
```

The glob expression `'project?.docx'` will return `'project1.docx'` or `'project5.docx'`, but it will not return `'project10.docx'`, because ? only matches to one character—so it will not match to the two-character string `'10'`.

Finally, you can also combine the asterisk and question mark to create even more complex glob expressions, like this:

```
>>> list(p.glob('*.?x?'))
[WindowsPath('C:/Users/Al/Desktop/calc.exe'), WindowsPath('C:/Users/Al/
Desktop/foo.txt'),
--snip--
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

The glob expression '*.?x?' will return files with any name and any three-character extension where the middle character is an 'x'.

By picking out files with specific attributes, the glob() method lets you easily specify the files in a directory you want to perform some operation on. You can use a for loop to iterate over the generator that glob() returns:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> for textFilePathObj in p.glob('*.txt'):
...     print(textFilePathObj) # Prints the Path object as a string.
...     # Do something with the text file.
...
C:\Users\Al\Desktop\foo.txt
C:\Users\Al\Desktop\spam.txt
C:\Users\Al\Desktop\zzz.txt
```

If you want to perform some operation on every file in a directory, you can use either os.listdir(p) or p.glob('*').

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist. Luckily, Path objects have methods to check whether a given path exists and whether it is a file or folder. Assuming that a variable p holds a Path object, you could expect the following:

- Calling p.exists() returns True if the path exists or returns False if it doesn't exist.
- Calling p.is_file() returns True if the path exists and is a file, or returns False otherwise.
- Calling p.is_dir() returns True if the path exists and is a directory, or returns False otherwise.

On my computer, here's what I get when I try these methods in the interactive shell:

```
>>> winDir = Path('C:/Windows')
>>> notExistsDir = Path('C:/This/Folder/Does/Not/Exist')
```

```
>>> calcFile = Path('C:/Windows
/System32/calc.exe')
>>> winDir.exists()
True
>>> winDir.is_dir()
True
>>> notExistsDir.exists()
False
>>> calcFile.is_file()
True
>>> calcFile.is_dir()
False
```

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `exists()` method. For instance, if I wanted to check for a flash drive with the volume named `D:\` on my Windows computer, I could do that with the following:

```
>>> dDrive = Path('D:/')
>>> dDrive.exists()
False
```

Oops! It looks like I forgot to plug in my flash drive.

The older `os.path` module can accomplish the same task with the `os.path.exists(path)`, `os.path.isfile(path)`, and `os.path.isdir(path)` functions, which act just like their `Path` function counterparts. As of Python 3.6, these functions can accept `Path` objects as well as strings of the file paths.

THE FILE READING/WRITING PROCESS

Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write. The functions covered in the next few sections will apply to plaintext files. *Plaintext files* contain only basic text characters and do not include font, size, or color information. Text files with the `.txt` extension or Python script files with the `.py` extension are examples of plaintext files. These can be opened with Windows's Notepad or macOS's TextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like in Figure 9-6.

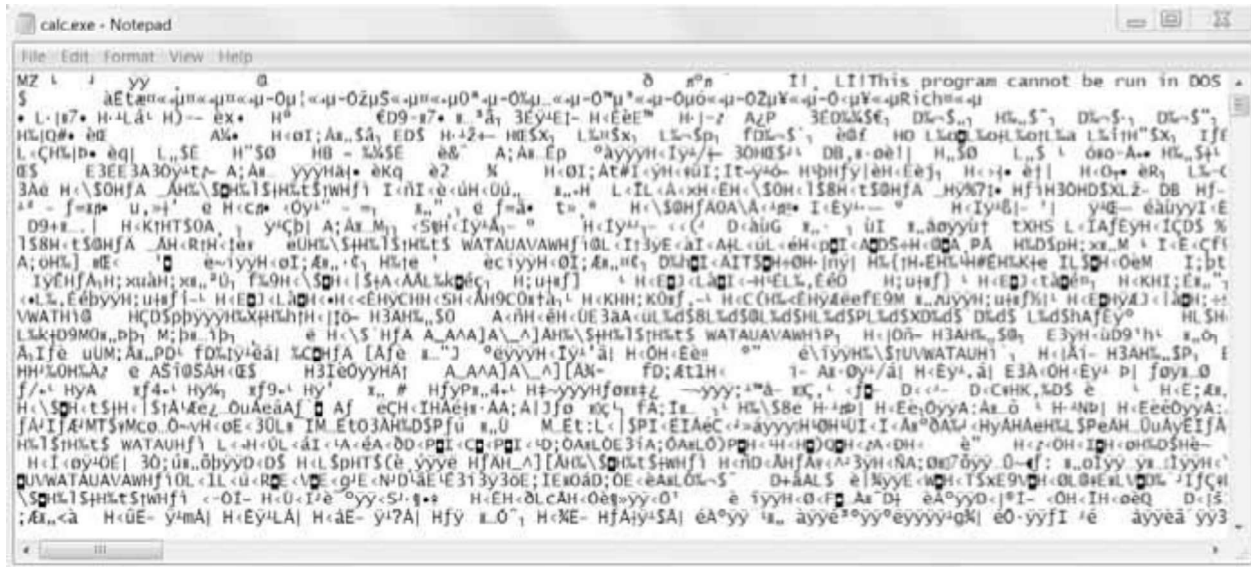


Figure 9-6: The Windows *calc.exe* program opened in Notepad

Since every different type of binary file must be handled in its own way, this book will not go into reading and writing raw binary files directly. Fortunately, many modules make working with binary files easier—you will explore one of them, the *shelve* module, later in this chapter. The *pathlib* module's `read_text()` method returns a string of the full contents of a text file. Its `write_text()` method creates a new text file (or overwrites an existing one) with the string passed to it. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('spam.txt')
>>> p.write_text('Hello, world!')
13
>>> p.read_text()
'Hello, world!'
```

These method calls create a *spam.txt* file with the content 'Hello, world!'. The 13 that `write_text()` returns indicates that 13 characters were written to the file. (You can often disregard this information.) The `read_text()` call reads and returns the contents of our new file as a string: 'Hello, world!'.

Keep in mind that these *Path* object methods only provide basic interactions with files. The more common way of writing to a file involves using the `open()` function and file objects. There are three steps to reading or writing files in Python:

1. Call the `open()` function to return a *File* object.
2. Call the `read()` or `write()` method on the *File* object.
3. Close the file by calling the `close()` method on the *File* object.

We'll go over these steps in the following sections.

Opening Files with the open() Function

To open a file with the `open()` function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path. The `open()` function returns a File object.

Try it by creating a text file named *hello.txt* using Notepad or TextEdit. Type **Hello, world!** as the content of this text file and save it in your user home folder. Then enter the following into the interactive shell:

```
>>> helloFile = open(Path.home() / 'hello.txt')
```

The `open()` function can also accept strings. If you're using Windows, enter the following into the interactive shell:

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

If you're using macOS, enter the following into the interactive shell instead:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

Make sure to replace *your_home_folder* with your computer username. For example, my username is *Al*, so I'd enter `'C:\\Users\\Al\\hello.txt'` on Windows. Note that the `open()` function only accepts Path objects as of Python 3.6. In previous versions, you always need to pass a string to `open()`.

Both these commands will open the file in “reading plaintext” mode, or *read mode* for short. When a file is opened in read mode, Python lets you only read data from the file; you can't write or modify it in any way. Read mode is the default mode for files you open in Python. But if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value `'r'` as a second argument to `open()`. So `open('/Users/Al/hello.txt', 'r')` and `open('/Users/Al/hello.txt')` do the same thing.

The call to `open()` returns a File object. A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with. In the previous example, you stored the File object in the variable `helloFile`. Now, whenever you want to read from or write to the file, you can do so by calling methods on the File object in `helloFile`.

Reading the Contents of Files

Now that you have a File object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the File object's `read()` method. Let's

continue with the *hello.txt* File object you stored in `helloFile`. Enter the following into the interactive shell:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello, world!'
```

If you think of the contents of a file as a single large string value, the `read()` method returns the string that is stored in the file.

Alternatively, you can use the `readlines()` method to get a *list* of string values from the file, one string for each line of text. For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and write the following text in it:

```
When, in disgrace with fortune and men's eyes,
I all alone beweeep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Make sure to separate the four lines with line breaks. Then enter the following into the interactive shell:

```
>>> sonnetFile = open(Path.home() / 'sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweeep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

Note that, except for the last line of the file, each of the string values ends with a newline character `\n`. A list of strings is often easier to work with than a single large string value.

Writing to Files

Python allows you to write content to a file in a way similar to how the `print()` function “writes” strings to the screen. You can’t write to a file you’ve opened in read mode, though. Instead, you need to open it in “write plaintext” mode or “append plaintext” mode, or *write mode* and *append mode* for short.

Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable’s value with a new value. Pass `'w'` as the second argument to `open()` to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this as appending to a list in a variable, rather than

overwriting the variable altogether. Pass 'a' as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

Let's put these concepts together. Enter the following into the interactive shell:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello, world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello, world!
Bacon is not a vegetable.
```

First, we open *bacon.txt* in write mode. Since there isn't a *bacon.txt* yet, Python creates one. Calling `write()` on the opened file and passing `write()` the string argument 'Hello, world! /n' writes the string to the file and returns the number of characters written, including the newline. Then we close the file.

To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write 'Bacon is not a vegetable.' to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call `read()`, store the resulting File object in `content`, close the file, and print `content`.

Note that the `write()` method does not automatically add a newline character to the end of the string like the `print()` function does. You will have to add this character yourself.

As of Python 3.6, you can also pass a Path object to the `open()` function instead of a string for the filename.

SAVING VARIABLES WITH THE SHELVE MODULE

You can save variables in your Python programs to binary shelf files using the `shelve` module. This way, your program can restore data to variables from the hard drive. The

shelve module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Enter the following into the interactive shell:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Call `shelve.open()` and pass it a filename, and then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelfFile`. We create a list `cats` and write `shelfFile['cats'] = cats` to store the list in `shelfFile` as a value associated with the key 'cats' (like in a dictionary). Then we call `close()` on `shelfFile`. Note that as of Python 3.7, you have to pass the `open()` shelf method filenames as strings. You can't pass it Path object.

After running the previous code on Windows, you will see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On macOS, only a single *mydata.db* file will be created.

These binary files contain the data you stored in your shelf. The format of these binary files is not important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode—they can do both once opened. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering `shelfFile['cats']` returns the same list that we stored earlier, so we know that the list is

correctly stored, and we call `close()`.

Just like dictionaries, shelf values have `keys()` and `values()` methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the `list()` function to get them in list form. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the `shelve` module.

SAVING VARIABLES WITH THE `PPRINT.PFORMAT()` FUNCTION

Recall from “Pretty Printing” on page 118 that the `pprint.pprint()` function will “pretty print” the contents of a list or dictionary to the screen, while the `pprint.pformat()` function will return this same text as a string instead of printing it. Not only is this string formatted to be easy to read, but it is also syntactically correct Python code. Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use. Using `pprint.pformat()` will give you a string that you can write to a `.py` file. This file will be your very own module that you can import whenever you want to use the variable stored in it.

For example, enter the following into the interactive shell:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Here, we import `pprint` to let us use `pprint.pformat()`. We have a list of dictionaries, stored in a variable `cats`. To keep the list in `cats` available even after we close the shell, we use

`pprint.pformat()` to return it as a string. Once we have the data in `cats` as a string, it's easy to write the string to a file, which we'll call *myCats.py*.

The modules that an `import` statement imports are themselves just Python scripts. When the string from `pprint.pformat()` is saved to a *.py* file, the file is a module that can be imported just like any other.

And since Python scripts are themselves just text files with the *.py* file extension, your Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a *.py* file (as opposed to saving variables with the `shelve` module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the `shelve` module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.

PROJECT: GENERATING RANDOM QUIZ FILES

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

1. Creates 35 different quizzes
2. Creates 50 multiple-choice questions for each quiz, in random order
3. Provides the correct answer and three random wrong answers for each question, in random order
4. Writes the quizzes to 35 text files

5. Writes the answer keys to 35 text files

This means the code will need to do the following:

1. Store the states and their capitals in a dictionary
2. Call `open()`, `write()`, and `close()` for the quiz and answer key text files
3. Use `random.shuffle()` to randomize the order of the questions and multiple-choice options

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
#!/python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
❶ import random
# The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany',
'North Carolina': 'Raleigh', 'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
❸ for quizNum in range(35):
    # TODO: Create the quiz and answer key files.
```

```
# TODO: Write out the header for the quiz.
```

```
# TODO: Shuffle the order of the states.
```

```
# TODO: Loop through all 50 states, making a question for each.
```

Since this program will be randomly ordering the questions and answers, you'll need to import the `random` module ❶ to make use of its functions. The `capitals` variable ❷ contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with `TODO` comments for now) will go inside a `for` loop that loops 35 times ❸. (This number can be changed to generate any number of quiz files.)

Step 2: Create the Quiz File and Shuffle the Question Order

Now it's time to start filling in those `TODOS`.

The code in the loop will be repeated 35 times—once for each quiz—so you have to worry about only one quiz at a time within the loop. First you'll create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period. Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Add the following lines of code to `randomQuizGenerator.py`:

```
#!/ python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
    ❶ quizFile = open(f'capitalsquiz{quizNum + 1}.txt', 'w')
    ❷ answerKeyFile = open(f'capitalsquiz_answers{quizNum + 1}.txt', 'w')
    # Write out the header for the quiz.
    ❸ quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + f'State Capitals Quiz (Form{quizNum + 1})')
    quizFile.write('\n\n')
```



```
# Shuffle the order of the states.
```

```
states = list(capitals.keys())
```

```
❶ random.shuffle(states)
```

```
# TODO: Loop through all 50 states, making a question for each.
```

The filenames for the quizzes will be *capitalsquiz<N>.txt*, where *<N>* is a unique number for the quiz that comes from *quizNum*, the for loop's counter. The answer key for *capitalsquiz<N>.txt* will be stored in a text file named *capitalsquiz_answers<N>.txt*. Each time through the loop, the *{quizNum + 1}* placeholder in *f'capitalsquiz{quizNum + 1}.txt'* and *f'capitalsquiz_answers{quizNum + 1}.txt'* will be replaced by the unique number, so the first quiz and answer key created will be *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*. These files will be created with calls to the *open()* function at ❶ and ❷, with 'w' as the second argument to open them in write mode.

The *write()* statements at ❸ create a quiz header for the student to fill out. Finally, a randomized list of US states is created with the help of the *random.shuffle()* function ❹, which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate the answer options for each question, which will be multiple choice from A to D. You'll need to create another for loop—this one to generate the content for each of the 50 questions on the quiz. Then there will be a third for loop nested inside to generate the multiple-choice options for each question. Make your code look like the following:

```
#!/ python3
```

```
# randomQuizGenerator.py - Creates quizzes with questions and answers in
```

```
# random order, along with the answer key.
```

```
--snip--
```

```
# Loop through all 50 states, making a question for each.
```

```
for questionNum in range(50):
```

```
    # Get right and wrong answers.
```

```
    ❶ correctAnswer = capitals[states[questionNum]]
```

```
    ❷ wrongAnswers = list(capitals.values())
```

```
    ❸ del wrongAnswers[wrongAnswers.index(correctAnswer)]
```

```
    ❹ wrongAnswers = random.sample(wrongAnswers, 3)
```

⑤ `answerOptions = wrongAnswers + [correctAnswer]`

⑥ `random.shuffle(answerOptions)`

TODO: Write the question and answer options to the quiz file.

TODO: Write the answer key to a file.

The correct answer is easy to get—it's stored as a value in the `capitals` dictionary ①. This loop will loop through the states in the shuffled `states` list, from `states[0]` to `states[49]`, find each state in `capitals`, and store that state's corresponding capital in `correctAnswer`.

The list of possible wrong answers is trickier. You can get it by duplicating *all* the values in the `capitals` dictionary ②, deleting the correct answer ③, and selecting three random values from this list ④. The `random.sample()` function makes it easy to do this selection. Its first argument is the list you want to select from; the second argument is the number of values you want to select. The full list of answer options is the combination of these three wrong answers with the correct answers ⑤. Finally, the answers need to be randomized ⑥ so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
#!/python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Loop through all 50 states, making a question for each.
for questionNum in range(50):
    --snip--

    # Write the question and the answer options to the quiz file.
    quizFile.write(f'{questionNum + 1}. What is the capital of
{states[questionNum]}?\n')
    ① for i in range(4):
        ② quizFile.write(f' {ABCD'[i]}. {answerOptions[i]}\n")
    quizFile.write('\n')
```

Write the answer key to a file.

```
❸ answerKeyFile.write(f"{questionNum + 1}.  
{'ABCD'[answerOptions.index(correctAnswer)]}")  
quizFile.close()  
answerKeyFile.close()
```

A for loop that goes through integers 0 to 3 will write the answer options in the answerOptions list ❶. The expression 'ABCD'[i] at ❷ treats the string 'ABCD' as an array and will evaluate to 'A','B', 'C', and then 'D' on each respective iteration through the loop.

In the final line ❸, the expression answerOptions.index(correctAnswer) will find the integer index of the correct answer in the randomly ordered answer options, and 'ABCD'[answerOptions.index(correctAnswer)] will evaluate to the correct answer's letter to be written to the answer key file.

After you run the program, this is how your *capitalsquiz1.txt* file will look, though of course your questions and answer options may be different from those shown here, depending on the outcome of your random.shuffle() calls:

Name:

Date:

Period:

State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?

- A. Hartford
- B. Santa Fe
- C. Harrisburg
- D. Charleston

2. What is the capital of Colorado?

- A. Raleigh
- B. Harrisburg
- C. Denver
- D. Lincoln

--snip--

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

1. D

2. C

3. A

4. C

--snip--

PROJECT: UPDATABLE MULTI-CLIPBOARD

Let's rewrite the "multi-clipboard" program from Chapter 6 so that it uses the `shelve` module. The user will now be able to save new strings to load to the clipboard without having to modify the source code. We'll name this new program `mcb.pyw` (since "mcb" is shorter to type than "multi-clipboard"). The `.pyw` extension means that Python won't show a Terminal window when it runs this program. (See Appendix B for more details.)

The program will save each piece of clipboard text under a keyword. For example, when you run `py mcb.pyw save spam`, the current contents of the clipboard will be saved with the keyword `spam`. This text can later be loaded to the clipboard again by running `py mcb.pyw spam`. And if the user forgets what keywords they have, they can run `py mcb.pyw list` to copy a list of all keywords to the clipboard.

Here's what the program does:

1. The command line argument for the keyword is checked.
2. If the argument is `save`, then the clipboard contents are saved to the keyword.
3. If the argument is `list`, then all the keywords are copied to the clipboard.
4. Otherwise, the text for the keyword is copied to the clipboard.

This means the code will need to do the following:

1. Read the command line arguments from `sys.argv`.
2. Read and write to the clipboard.
3. Save and load to a shelf file.

If you use Windows, you can easily run this script from the Run... window by creating a batch file named `mcb.bat` with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Step 1: Comments and Shelf Setup

Let's start by making a skeleton script with some comments and basic setup. Make your code look like the following:

```
#!/python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
#     py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
#     py.exe mcb.pyw list - Loads all keywords to clipboard.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

# TODO: Save clipboard content.

# TODO: List keywords and load content.

mcbShelf.close()
```

It's common practice to put general usage information in comments at the top of the file ❶. If you ever forget how to run your script, you can always look at these comments for a reminder. Then you import your modules ❷. Copying and pasting will require the `pyperclip` module, and reading the command line arguments will require the `sys` module. The `shelve` module will also come in handy: Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file. Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into your program. The shelf file will be named with the prefix `mcb` ❸.

Step 2: Save Clipboard Content with a Keyword

The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case. Make your code look like the following:

```
#!/python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    ❷ mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    ❸ # TODO: List keywords and load content.
```

```
mcbShelf.close()
```

If the first command line argument (which will always be at index 1 of the `sys.argv` list) is 'save' ❶, the second command line argument is the keyword for the current content of the clipboard. The keyword will be used as the key for `mcbShelf`, and the value will be the text currently on the clipboard ❷.

If there is only one command line argument, you will assume it is either 'list' or a keyword to load content onto the clipboard. You will implement that code later. For now, just put a TODO comment there ❸.

Step 3: List Keywords and Load a Keyword's Content

Finally, let's implement the two remaining cases: the user wants to load clipboard text in from a keyword, or they want a list of all available keywords. Make your code look like the following:

```
#!/ python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # List keywords and load content.
    ❶ if sys.argv[1].lower() == 'list':
        ❷ pyperclip.copy(str(list(mcbShelf.keys())))
    elif sys.argv[1] in mcbShelf:
        ❸ pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

If there is only one command line argument, first let's check whether it's 'list' ❶. If so, a string representation of the list of shelf keys will be copied to the clipboard ❷. The user can paste this list into an open text editor to read it.

Otherwise, you can assume the command line argument is a keyword. If this keyword exists in the `mcbShelf` shelf as a key, you can load the value onto the clipboard ❸.

And that's it! Launching this program has different steps depending on what operating system your computer uses. See Appendix B for details.

Recall the password locker program you created in Chapter 6 that stored the passwords in a dictionary. Updating the passwords required changing the source code of the program. This isn't ideal, because average users don't feel comfortable changing source code to update their software. Also, every time you modify the source code to a program, you run the risk of accidentally introducing new bugs. By storing the data for a program in a different place than the code, you can make your programs easier for others to use and more resistant to bugs.

SUMMARY

Files are organized into folders (also called directories), and a path describes the location of a file. Every program running on your computer has a current working directory, which allows you to specify file paths relative to the current location instead of always typing the full (or absolute) path. The `pathlib` and `os.path` modules have many functions for manipulating file paths.

Your programs can also directly interact with the contents of text files. The `open()` function can open these files to read in their contents as one large string (with the `read()` method) or as a list of strings (with the `readlines()` method). The `open()` function can open files in write or append mode to create new text files or add to existing text files, respectively.

In previous chapters, you used the clipboard as a way of getting large amounts of text into a program, rather than typing it all in. Now you can have your programs read files directly from the hard drive, which is a big improvement, since files are much less volatile than the clipboard.

In the next chapter, you will learn how to handle the files themselves, by copying them, deleting them, renaming them, moving them, and more.

PRACTICE QUESTIONS

1. What is a relative path relative to?
2. What does an absolute path start with?
3. What does `Path('C:/Users') / 'A'` evaluate to on Windows?
4. What does `'C:/Users' / 'A'` evaluate to on Windows?
5. What do the `os.getcwd()` and `os.chdir()` functions do?
6. What are the `.` and `..` folders?
7. In `C:\bacon\eggs\spam.txt`, which part is the dir name, and which part is the base name?

8. What are the three “mode” arguments that can be passed to the `open()` function?
9. What happens if an existing file is opened in write mode?
10. What is the difference between the `read()` and `readlines()` methods?
11. What data structure does a shelf value resemble?

PRACTICE PROJECTS

For practice, design and write the following programs.

Extending the Multi-Clipboard

Extend the multi-clipboard program in this chapter so that it has a `delete <keyword>` command line argument that will delete a keyword from the shelf. Then add a `delete` command line argument that will delete *all* keywords.

Mad Libs

Create a Mad Libs program that reads in text files and lets the user add their own text anywhere the word *ADJECTIVE*, *NOUN*, *ADVERB*, or *VERB* appears in the text file. For example, a text file may look like this:

The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was unaffected by these events.

The program would find these occurrences and prompt the user to replace them.

Enter an adjective:

silly

Enter a noun:

chandelier

Enter a verb:

screamed

Enter a noun:

pickup truck

The following text file would then be created:

The silly panda walked to the chandelier and then screamed. A nearby pickup truck was unaffected by these events.

ORGANIZING FILES

CSE, HIT, Nidasoshi

Making copies of all PDF files (and *only* the PDF files) in every subfolder of a folder

Removing the leading zeros in the filenames for every file in a folder of hundreds of files named *spam001.txt*, *spam002.txt*, *spam003.txt*, and so on

Compressing the contents of several folders into one ZIP file (which could be a simple backup system)

All this boring stuff is just begging to be automated in Python. By programming your computer to do these tasks, you can transform it into a quick-working file clerk who never makes mistakes.

As you begin working with files, you may find it helpful to be able to quickly see what the extension (*.txt*, *.pdf*, *.jpg*, and so on) of a file is. With macOS and Linux, your file browser most likely shows extensions automatically. With Windows, file extensions may be hidden by default. To show extensions, go to **Start** ▶ **Control Panel** ▶ **Appearance and Personalization** ▶ **Folder Options**. On the View tab, under Advanced Settings, uncheck the **Hide extensions for known file types** checkbox.

THE SHUTIL MODULE

The `shutil` (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the `shutil` functions, you will first need to use `import shutil`.

Copying Files and Folders

The `shutil` module provides functions for copying files, as well as entire folders.

Calling `shutil.copy(source, destination)` will copy the file at the path *source* to the folder at the path *destination*. (Both *source* and *destination* can be strings or `Path` objects.) If *destination* is a filename, it will be used as the new name of the copied file. This function returns a string or `Path` object of the copied file.

Enter the following into the interactive shell to see how `shutil.copy()` works:

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
❶ >>> shutil.copy(p / 'spam.txt', p / 'some_folder')
'C:\\Users\\AI\\some_folder\\spam.txt'
```

```
❷ >>> shutil.copy(p / 'eggs.txt', p / 'some_folder/eggs2.txt')
```

```
WindowsPath('C:/Users/Al/some_folder/eggs2.txt')
```

The first `shutil.copy()` call copies the file at `C:\Users\Al\spam.txt` to the folder `C:\Users\Al\some_folder`. The return value is the path of the newly copied file. Note that since a folder was specified as the destination ❶, the original `spam.txt` filename is used for the new, copied file's filename. The second `shutil.copy()` call ❷ also copies the file at `C:\Users\Al\eggs.txt` to the folder `C:\Users\Al\some_folder` but gives the copied file the name `eggs2.txt`.

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it. Calling `shutil.copytree(source, destination)` will copy the folder at the path `source`, along with all of its files and subfolders, to the folder at the path `destination`. The `source` and `destination` parameters are both strings. The function returns a string of the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
>>> shutil.copytree(p / 'spam', p / 'spam_backup')
WindowsPath('C:/Users/Al/spam_backup')
```

The `shutil.copytree()` call creates a new folder named `spam_backup` with the same content as the original `spam` folder. You have now safely backed up your precious, precious spam.

Moving and Renaming Files and Folders

Calling `shutil.move(source, destination)` will move the file or folder at the path `source` to the path `destination` and will return a string of the absolute path of the new location.

If `destination` points to a folder, the `source` file gets moved into `destination` and keeps its current filename. For example, enter the following into the interactive shell:

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named `eggs` already exists in the `C:\` directory, this `shutil.move()` call says, “Move `C:\bacon.txt` into the folder `C:\eggs`.”

If there had been a *bacon.txt* file already in *C:\eggs*, it would have been overwritten. Since it's easy to accidentally overwrite files in this way, you should take some care when using `move()`.

The *destination* path can also specify a filename. In the following example, the *source* file is moved *and* renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This line says, “Move *C:\bacon.txt* into the folder *C:\eggs*, and while you're at it, rename that *bacon.txt* file to *new_bacon.txt*.”

Both of the previous examples worked under the assumption that there was a folder *eggs* in the *C:* directory. But if there is no *eggs* folder, then `move()` will rename *bacon.txt* to a file named *eggs*.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Here, `move()` can't find a folder named *eggs* in the *C:* directory and so assumes that *destination* must be specifying a filename, not a folder. So the *bacon.txt* text file is renamed to *eggs* (a text file without the *.txt* file extension)—probably not what you wanted! This can be a tough-to-spot bug in your programs since the `move()` call can happily do something that might be quite different from what you were expecting. This is yet another reason to be careful when using `move()`.

Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  --snip--
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\eggs\\ham'
```

Python looks for *eggs* and *ham* inside the directory *does_not_exist*. It doesn't find the nonexistent directory, so it can't move *spam.txt* to the path you specified.

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the `os` module, whereas to delete a folder and all of its contents, you use the `shutil` module.

- Calling `os.unlink(path)` will delete the file at `path`.
- Calling `os.rmdir(path)` will delete the folder at `path`. This folder must be empty of any files or folders.
- Calling `shutil.rmtree(path)` will remove the folder at `path`, and all files and folders it contains will also be deleted.

Be careful when using these functions in your programs! It's often a good idea to first run your program with these calls commented out and with `print()` calls added to show the files that would be deleted. Here is a Python program that was intended to delete files that have the `.txt` file extension but has a typo (highlighted in bold) that causes it to delete `.rxt` files instead:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt'):
    os.unlink(filename)
```

If you had any important files ending with `.rxt`, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt'):
    #os.unlink(filename)
    print(filename)
```

Now the `os.unlink()` call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted. Running this version of the program first will show you that you've accidentally told the program to delete `.rxt` files instead of `.txt` files.

Once you are certain the program works as intended, delete the `print(filename)` line and uncomment the `os.unlink(filename)` line. Then run the program again to actually delete the files.

Safe Deletes with the `send2trash` Module

Since Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the third-party `send2trash` module. You can install this module by running `pip install --user send2trash` from a Terminal window. (See Appendix A for a more in-depth explanation of how to install third-party modules.)

Using `send2trash` is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with `send2trash` you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed `send2trash`, enter the following into the interactive shell:

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

In general, you should always use the `send2trash.send2trash()` function to delete files and folders. But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does. If you want your program to free up disk space, use the `os` and `shutil` functions for deleting files and folders. Note that the `send2trash()` function can only send files to the recycle bin; it cannot pull files out of it.

WALKING A DIRECTORY TREE

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Let's look at the `C:\delicious` folder with its contents, shown in Figure 10-1.

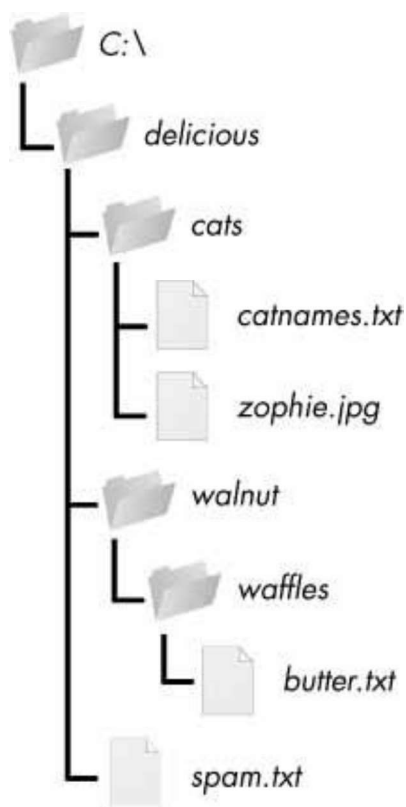


Figure 10-1: An example folder that contains three folders and four files

Here is an example program that uses the `os.walk()` function on the directory tree from Figure 10-1:

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)

    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': ' + filename)

print("")
```

The `os.walk()` function is passed a single string value: the path of a folder. You can use `os.walk()` in a for loop statement to walk a directory tree, much like how you can use the `range()` function to walk over a range of numbers. Unlike `range()`, the `os.walk()` function will return three values on each iteration through the loop:

- A string of the current folder's name
- A list of strings of the folders in the current folder

- A list of strings of the files in the current folder

(By current folder, I mean the folder for the current iteration of the `for` loop. The current working directory of the program is *not* changed by `os.walk()`.)

Just like you can choose the variable name `i` in the code `for i in range(10):`, you can also choose the variable names for the three values listed earlier. I usually use the names `foldername`, `subfolders`, and `filenames`.

When you run this program, it will output the following:

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt
```

```
The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg
```

```
The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles
```

```
The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since `os.walk()` returns lists of strings for the `subfolder` and `filename` variables, you can use these lists in their own `for` loops. Replace the `print()` function calls with your own custom code. (Or if you don't need one or both of them, remove the `for` loops.)

COMPRESSING FILES WITH THE ZIPFILE MODULE

You may be familiar with ZIP files (with the `.zip` file extension), which can hold the compressed contents of many other files. Compressing a file reduces its size, which is useful when transferring it over the internet. And since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an *archive file*, can then be, say, attached to an email.

Your Python programs can create and open (or *extract*) ZIP files using functions in the `zipfile` module. Say you have a ZIP file named `example.zip` that has the contents shown in Figure 10-2.

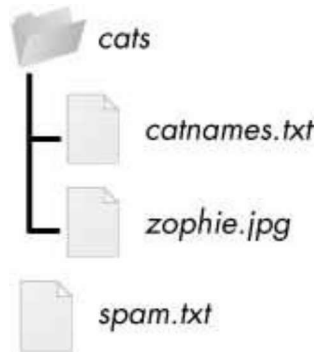


Figure 10-2: The contents of example.zip

You can download this ZIP file from <https://nostarch.com/automatestuff2/> or just follow along using a ZIP file already on your computer.

Reading ZIP Files

To read the contents of a ZIP file, first you must create a `ZipFile` object (note the capital letters *Z* and *F*). `ZipFile` objects are conceptually similar to the `File` objects you saw returned by the `open()` function in the previous chapter: they are values through which the program interacts with the file. To create a `ZipFile` object, call the `zipfile.ZipFile()` function, passing it a string of the `.ZIP` file's filename. Note that `zipfile` is the name of the Python module, and `ZipFile()` is the name of the function.

For example, enter the following into the interactive shell:

```
>>> import zipfile, os

>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> f'Compressed file is {round(spamInfo.file_size / spamInfo
.compress_size, 2)}x smaller!'
)
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

A `ZipFile` object has a `namelist()` method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the `getinfo()` `ZipFile` method to return a `ZipInfo` object about that particular file. `ZipInfo` objects have their own attributes, such as `file_size` and `compress_size` in bytes, which hold integers of the original file size and compressed file size, respectively. While a `ZipFile` object represents an entire archive file, a `ZipInfo` object holds useful information about a *single file* in the archive.

The command at ❶ calculates how efficiently *example.zip* is compressed by dividing the original file size by the compressed file size and prints this information.

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of *example.zip* will be extracted to `C:\`. Optionally, you can pass a folder name to `extractall()` to have it extract the files into a folder other than the current working directory. If the folder passed to the `extractall()` method does not exist, it will be created. For instance, if you replaced the call at ❶ with `exampleZip.extractall('C:\delicious')`, the code would extract the files from *example.zip* into a newly created `C:\delicious` folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
>>> exampleZip.extract('spam.txt')
'C:\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\some\new\folders')
'C:\some\new\folders\spam.txt'
>>> exampleZip.close()
```

The string you pass to `extract()` must match one of the strings in the list returned by `namelist()`. Optionally, you can pass a second argument to `extract()` to extract the file into a folder other than the current working directory. If this second argument is a folder that

doesn't yet exist, Python will create the folder. The value that `extract()` returns is the absolute path to which the file was extracted.

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the `ZipFile` object in *write mode* by passing 'w' as the second argument. (This is similar to opening a text file in write mode by passing 'w' to the `open()` function.)

When you pass a path to the `write()` method of a `ZipFile` object, Python will compress the file at that path and add it into the ZIP file. The `write()` method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to `zipfile.ZIP_DEFLATED`. (This specifies the *deflate* compression algorithm, which works well on all types of data.) Enter the following into the interactive shell:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.

Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to `zipfile.ZipFile()` to open the ZIP file in *append mode*.

PROJECT: RENAMING FILES WITH AMERICAN-STYLE DATES TO EUROPEAN-STYLE DATES

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead.

Here's what the program does:

1. It searches all the filenames in the current working directory for American-style dates.
2. When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

1. Create a regex that can identify the text pattern of American-style dates.
2. Call `os.listdir()` to find all the files in the working directory.
3. Loop over each filename, using the regex to check whether it has a date.
4. If it has a date, rename the file with `shutil.move()`.

For this project, open a new file editor window and save your code as *renameDates.py*.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as TODO makes them easy to find using Mu editor's CTRL-F find feature. Make your code look like the following:

```
#!/ python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
```

```
❶ import shutil, os, re
```

```
# Create a regex that matches files with the American date format.
```

```
❷ datePattern = re.compile(r"^(.*?) # all text before the date
```

```
((0|1)?\d)- # one or two digits for the month
```

```
((0|1|2|3)?\d)- # one or two digits for the day
```

```
((19|20)\d\d) # four digits for the year
```

```
(.*?)$ # all text after the date
```

```
""", re.VERBOSE❸)
```

```
# TODO: Loop over the files in the working directory.
```

```
# TODO: Skip files without a date.
```

```
# TODO: Get the different parts of the filename.
```

```
# TODO: Form the European-style filename.
```

```
# TODO: Get the full, absolute file paths.
```

TODO: Rename the files.

From this chapter, you know the `shutil.move()` function can be used to rename files: its arguments are the name of the file to rename and the new filename. Because this function exists in the `shutil` module, you must import that module ❶.

But before renaming the files, you need to identify which files you want to rename. Filenames with dates such as *spam4-4-1984.txt* and *01-03-2014eggs.zip* should be renamed, while filenames without dates such as *littlebrother.epub* can be ignored.

You can use a regular expression to identify this pattern. After importing the `re` module at the top, call `re.compile()` to create a `Regex` object ❷. Passing `re.VERBOSE` for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

The regular expression string begins with `^(.*?)` to match any text at the beginning of the filename that might come before the date. The `((0|1)?\d)` group matches the month. The first digit can be either 0 or 1, so the regex matches 12 for December but also 02 for February. This digit is also optional so that the month can be 04 or 4 for April. The group for the day is `((0|1|2|3)?\d)` and follows similar logic; 3, 03, and 31 are all valid numbers for days. (Yes, this regex will accept some invalid dates such as 4-31-2014, 2-29-2013, and 0-15-2014. Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

While 1885 is a valid year, you can just look for years in the 20th or 21st century. This will keep your program from accidentally matching nondate filenames with a date-like format, such as *10-10-1000.txt*.

The `(.*?)$` part of the regex will match any text that comes after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from `os.listdir()` and match them against the regex. Any files that do not have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Fill in the first three TODOs in your program with the following code:

```
#!/ python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
```

```
--snip--
```

```
# Loop over the files in the working directory.
```

```
for amerFilename in os.listdir('.'):

```

```
    mo = datePattern.search(amerFilename)

```

```
    # Skip files without a date.

```

```
    ❶ if mo == None:

```

```
        ❷ continue

```

```
    ❸ # Get the different parts of the filename.

```

```
    beforePart = mo.group(1)

```

```
    monthPart = mo.group(2)

```

```
    dayPart = mo.group(4)

```

```
    yearPart = mo.group(6)

```

```
    afterPart = mo.group(8)

```

```
--snip--
```

If the Match object returned from the `search()` method is `None` ❶, then the filename in `amerFilename` does not match the regular expression. The `continue` statement ❷ will skip the rest of the loop and move on to the next filename.

Otherwise, the various strings matched in the regular expression groups are stored in variables named `beforePart`, `monthPart`, `dayPart`, `yearPart`, and `afterPart` ❸. The strings in these variables will be used to form the European-style filename in the next step.

To keep the group numbers straight, try reading the regex from the beginning, and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. Here's an example:

```
datePattern = re.compile(r"^(1) # all text before the date
```

```
    (2 (3) )-          # one or two digits for the month
```

```
    (4 (5) )-          # one or two digits for the day
```

```
    (6 (7) )          # four digits for the year
```

```
    (8)$              # all text after the date
```

```
    """, re.VERBOSE)
```

Here, the numbers **1** through **8** represent the groups in the regular expression you wrote. Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: the date comes before the month. Fill in the three remaining TODOs in your program with the following code:

```
#!/python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format # to European DD-
MM-YYYY.

--snip--

# Form the European-style filename.
❶ euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
    afterPart

# Get the full, absolute file paths.
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)

# Rename the files.
❷ print(f'Renaming "{amerFilename}" to "{euroFilename}"...')
❸ #shutil.move(amerFilename, euroFilename) # uncomment after testing
```

Store the concatenated string in a variable named `euroFilename` ❶. Then, pass the original filename in `amerFilename` and the new `euroFilename` variable to the `shutil.move()` function to rename the file ❸.

This program has the `shutil.move()` call commented out and instead prints the filenames that will be renamed ❷. Running the program like this first can let you double-check that the files are renamed correctly. Then you can uncomment the `shutil.move()` call and run the program again to actually rename the files.

Ideas for Similar Programs

There are many other reasons you might want to rename a large number of files.

- To add a prefix to the start of the filename, such as adding `spam_` to rename `eggs.txt` to `spam_eggs.txt`
- To change filenames with European-style dates to American-style dates
- To remove the zeros from files such as `spam0042.txt`

PROJECT: BACKING UP A FOLDER INTO A ZIP FILE

Say you're working on a project whose files you keep in a folder named *C:\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd like to keep different versions, so you want the ZIP file's filename to increment each time it is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*, *AlsPythonBook_3.zip*, and so on. You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named `backupToZip()`. This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#!/ python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

❶ import zipfile, os

def backupToZip(folder):
    # Back up the entire contents of "folder" into a ZIP file.

    folder = os.path.abspath(folder) # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.

    ❷ number = 1
    ❸ while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    ❹ # TODO: Create the ZIP file.
```



```
# TODO: Walk the entire folder tree and compress the files in each folder.
```

```
print('Done.')
```

```
backupToZip('C:\\delicious')
```

Do the basics first: add the shebang (`#!`) line, describe what the program does, and import the `zipfile` and `os` modules ❶.

Define a `backupToZip()` function that takes just one parameter, `folder`. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the folder, and add each of the subfolders and files to the ZIP file. Write `TODO` comments for these steps in the source code to remind yourself to do them later ❷.

The first part, naming the ZIP file, uses the base name of the absolute path of folder. If the folder being backed up is `C:\delicious`, the ZIP file's name should be `delicious_N.zip`, where $N = 1$ is the first time you run the program, $N = 2$ is the second time, and so on.

You can determine what N should be by checking whether `delicious_1.zip` already exists, then checking whether `delicious_2.zip` already exists, and so on. Use a variable named `number` for N ❸, and keep incrementing it inside the loop that calls `os.path.exists()` to check whether the file exists ❹. The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file. Make your program look like the following:

```
#!/python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
while True:
    zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
    if not os.path.exists(zipFilename):
        break
    number = number + 1

# Create the ZIP file.
print(f'Creating {zipFilename}...')
❶ backupZip = zipfile.ZipFile(zipFilename, 'w')
```

```
# TODO: Walk the entire folder tree and compress the files in each folder.  
print('Done.')
```

```
backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the `zipFilename` variable, you can call `zipfile.ZipFile()` to actually create the ZIP file ❶. Be sure to pass 'w' as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the `os.walk()` function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
#!/ python3  
# backupToZip.py - Copies an entire folder and its contents into  
# a ZIP file whose filename increments.  
  
--snip--  
  
# Walk the entire folder tree and compress the files in each folder.  
❶ for foldername, subfolders, filenames in os.walk(folder):  
    print(f'Adding files in {foldername}...')  
    # Add the current folder to the ZIP file.  
    ❷ backupZip.write(foldername)  
  
    # Add all the files in this folder to the ZIP file.  
    ❸ for filename in filenames:  
        newBase = os.path.basename(folder) + '_'  
        if filename.startswith(newBase) and filename.endswith('.zip'):  
            continue # don't back up the backup ZIP files  
        backupZip.write(os.path.join(foldername, filename))  
    backupZip.close()  
    print('Done.')
```

```
backupToZip('C:\\delicious')
```

You can use `os.walk()` in a for loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder.

In the `for` loop, the folder is added to the ZIP file ❷. The nested `for` loop can go through each filename in the `filenames` list ❸. Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

The second time you run it, it will put all the files in `C:\delicious` into a ZIP file named `delicious_2.zip`, and so on.

Ideas for Similar Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you can write programs that do the following:

- Walk a directory tree and archive just files with certain extensions, such as `.txt` or `.py`, and nothing else.
- Walk a directory tree and archive every file except the `.txt` and `.py` ones.
- Find the folder in a directory tree that has the greatest number of files or the folder that uses the most disk space.

SUMMARY

Even if you are an experienced computer user, you probably handle files manually with the mouse and keyboard. Modern file explorers make it easy to work with a few files. But sometimes you'll need to perform a task that would take hours using your computer's file explorer.

The `os` and `shutil` modules offer functions for copying, moving, renaming, and deleting files. When deleting files, you might want to use the `send2trash` module to move files to the recycle bin or trash rather than permanently deleting them. And when writing programs that handle files, it's a good idea to comment out the code that does the actual copy/move/rename/delete and add a `print()` call instead so you can run the program and verify exactly what it will do.

Often you will need to perform these operations not only on files in one folder but also on every folder in that folder, every folder in those folders, and so on. The `os.walk()`

function handles this trek across the folders for you so that you can concentrate on what your program needs to do with the files in them.

The `zipfile` module gives you a way of compressing and extracting files in `.ZIP` archives through Python. Combined with the file-handling functions of `os` and `shutil`, `zipfile` makes it easy to package up several files from anywhere on your hard drive. These `.ZIP` files are much easier to upload to websites or send as email attachments than many separate files.

Previous chapters of this book have provided source code for you to copy. But when you write your own programs, they probably won't come out perfectly the first time. The next chapter focuses on some Python modules that will help you analyze and debug your programs so that you can quickly get them working correctly.

PRACTICE QUESTIONS

1. What is the difference between `shutil.copy()` and `shutil.copytree()`?
2. What function is used to rename files?
3. What is the difference between the delete functions in the `send2trash` and `shutil` modules?
4. `ZipFile` objects have a `close()` method just like `File` objects' `close()` method. What `ZipFile` method is equivalent to `File` objects' `open()` method?

PRACTICE PROJECTS

For practice, write programs to do the following tasks.

Selective Copy

Write a program that walks through a folder tree and searches for files with a certain file extension (such as `.pdf` or `.jpg`). Copy these files from whatever location they are in to a new folder.

Deleting Unneeded Files

It's not uncommon for a few unneeded but humongous files or folders to take up the bulk of the space on your hard drive. If you're trying to free up room on your computer, you'll get the most bang for your buck by deleting the most massive of the unwanted files. But first you have to find them.

Write a program that walks through a folder tree and searches for exceptionally large files or folders—say, ones that have a file size of more than 100MB. (Remember that to

get a file's size, you can use `os.path.getsize()` from the `os` module.) Print these files with their absolute path to the screen.

Filling in the Gaps

Write a program that finds all files with a given prefix, such as *spam001.txt*, *spam002.txt*, and so on, in a single folder and locates any gaps in the numbering (such as if there is a *spam001.txt* and *spam003.txt* but no *spam002.txt*). Have the program rename all the later files to close this gap.

As an added challenge, write another program that can insert gaps into numbered files so that a new file can be added.

CSE, HIT, Nidasoshi

DEBUGGING

CSE, HIT, Nidasoshi

To paraphrase an old joke among programmers, writing code accounts for 90 percent of programming. Debugging code accounts for the other 90 percent.

Your computer will do only what you tell it to do; it won't read your mind and do what you *intended* it to do. Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem.

Fortunately, there are a few tools and techniques to identify what exactly your code is doing and where it's going wrong. First, you will look at logging and assertions, two features that can help you detect bugs early. In general, the earlier you catch bugs, the easier they will be to fix.

Second, you will look at how to use the debugger. The debugger is a feature of Mu that executes a program one instruction at a time, giving you a chance to inspect the values in variables while your code runs, and track how the values change over the course of your program. This is much slower than running the program at full speed, but it is helpful to see the actual values in a program while it runs, rather than deducing what the values might be from the source code.

RAISING EXCEPTIONS

Python raises an exception whenever it tries to execute invalid code. In Chapter 3, you read about how to handle Python's exceptions with `try` and `except` statements so that your program can recover from exceptions that you anticipated. But you can also raise your own exceptions in your code. Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the `except` statement."

Exceptions are raised with a `raise` statement. In code, a `raise` statement consists of the following:

- The `raise` keyword
- A call to the `Exception()` function
- A string with a helpful error message passed to the `Exception()` function

For example, enter the following into the interactive shell:

```
>>> raise Exception('This is the error message.')
```

```
Traceback (most recent call last):
```

```
File "<pyshell#191>", line 1, in <module>
```

```
raise Exception('This is the error message.')
```

```
Exception: This is the error message.
```

If there are no `try` and `except` statements covering the `raise` statement that raised the exception, the program simply crashes and displays the exception's error message.

Often it's the code that calls the function, rather than the function itself, that knows how to handle an exception. That means you will commonly see a raise statement inside a function and the try and except statements in the code calling the function. For example, open a new file editor tab, enter the following code, and save the program as *boxPrint.py*:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
        ❶ raise Exception('Symbol must be a single character string.')
    if width <= 2:
        ❷ raise Exception('Width must be greater than 2.')
    if height <= 2:
        ❸ raise Exception('Height must be greater than 2.')

    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

for sym, w, h in (('!', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
    ❹ except Exception as err:
        ❺ print('An exception happened: ' + str(err))
```

You can view the execution of this program at <https://autbor.com/boxprint>. Here we've defined a `boxPrint()` function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height. This box shape is printed to the screen.

Say we want the character to be a single character, and the width and height to be greater than 2. We add if statements to raise exceptions if these requirements aren't satisfied. Later, when we call `boxPrint()` with various arguments, our try/except will handle invalid arguments.

This program uses the `except Exception as err` form of the except statement ❹. If an Exception object is returned from `boxPrint()` ❶ ❷ ❸, this except statement will store it in a variable named `err`. We can then convert the Exception object to a string by passing it to `str()` to produce a user-friendly error message ❺. When you run this *boxPrint.py*, the output will look like this:


```
****
* *
* *
****
```

```
OOOOOOOOOOOOOOOOOOOOOOOO
```

```
O      O
O      O
O      O
```

```
OOOOOOOOOOOOOOOOOOOOOOOO
```

An exception happened: Width must be greater than 2.

An exception happened: Symbol must be a single character string.

Using the `try` and `except` statements, you can handle errors more gracefully instead of letting the entire program crash.

GETTING THE TRACEBACK AS A STRING

When Python encounters an error, it produces a treasure trove of error information called the *traceback*. The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error. This sequence of calls is called the *call stack*.

Open a new file editor tab in Mu, enter the following program, and save it as *errorExample.py*:

```
def spam():
    bacon()

def bacon():
    raise Exception('This is the error message.')

spam()
```

When you run *errorExample.py*, the output will look like this:

```
Traceback (most recent call last):
  File "errorExample.py", line 7, in <module>
    spam()
  File "errorExample.py", line 2, in spam
    bacon()
  File "errorExample.py", line 5, in bacon
```

```
raise Exception('This is the error message.')
```

Exception: This is the error message.

From the traceback, you can see that the error happened on line 5, in the `bacon()` function. This particular call to `bacon()` came from line 2, in the `spam()` function, which in turn was called on line 7. In programs where functions can be called from multiple places, the call stack can help you determine which call led to the error.

Python displays the traceback whenever a raised exception goes unhandled. But you can also obtain it as a string by calling `traceback.format_exc()`. This function is useful if you want the information from an exception's traceback but also want an `except` statement to gracefully handle the exception. You will need to import Python's `traceback` module before calling this function.

For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a text file and keep your program running. You can look at the text file later, when you're ready to debug your program. Enter the following into the interactive shell:

```
>>> import traceback
>>> try:
...     raise Exception('This is the error message.')
except:
...     errorFile = open('errorInfo.txt', 'w')
...     errorFile.write(traceback.format_exc())
...     errorFile.close()
...     print('The traceback info was written to errorInfo.txt.')
```

111

The traceback info was written to errorInfo.txt.

The 111 is the return value from the `write()` method, since 111 characters were written to the file. The traceback text was written to *errorInfo.txt*.

Traceback (most recent call last):

File "<pyshell#28>", line 2, in <module>

Exception: This is the error message.

In “Logging” on page 255, you’ll learn how to use the `logging` module, which is more effective than simply writing this error information to text files.

ASSERTIONS

An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by `assert` statements. If the sanity check fails, then an `AssertionError` exception is raised. In code, an `assert` statement consists of the following:

- The `assert` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A comma
- A string to display when the condition is `False`

In plain English, an `assert` statement says, "I assert that the condition holds true, and if not, there is a bug somewhere, so immediately stop the program." For example, enter the following into the interactive shell:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.sort()
>>> ages
[15, 17, 22, 26, 47, 54, 57, 73, 80, 92]
>>> assert
ages[0] <= ages[-1] # Assert that the first age is <= the last age.
```

The `assert` statement here asserts that the first item in `ages` should be less than or equal to the last one. This is a sanity check; if the code in `sort()` is bug-free and did its job, then the assertion would be true.

Because the `ages[0] <= ages[-1]` expression evaluates to `True`, the `assert` statement does nothing.

However, let's pretend we had a bug in our code. Say we accidentally called the `reverse()` list method instead of the `sort()` list method. When we enter the following in the interactive shell, the `assert` statement raises an `AssertionError`:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.reverse()
>>> ages
[73, 47, 80, 17, 15, 22, 54, 92, 57, 26]
>>> assert ages[0] <= ages[-1] # Assert that the first age is <= the last age.
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

`AssertionError`

Unlike exceptions, your code should *not* handle assert statements with try and except; if an assert fails, your program *should* crash. By “failing fast” like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the bug’s cause.

Assertions are for programmer errors, not user errors. Assertions should only fail while the program is under development; a user should never see an assertion error in a finished program. For errors that your program can run into as a normal part of its operation (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement. You shouldn’t use assert statements in place of raising exceptions, because users can choose to turn off assertions. If you run a Python script with `python -O myscript.py` instead of `python myscript.py`, Python will skip assert statements. Users might disable assertions when they’re developing a program and need to run it in a production setting that requires peak performance. (Though, in many cases, they’ll leave assertions enabled even then.)

Assertions also aren’t a replacement for comprehensive testing. For instance, if the previous ages example was set to `[10, 3, 2, 1, 20]`, then the `assert ages[0] <= ages[-1]` assertion wouldn’t notice that the list was unsorted, because it just happened to have a first age that was less than or equal to the last age, which is the only thing the assertion checked for.

Using an Assertion in a Traffic Light Simulation

Say you’re building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary with keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street. To start the project, you want to write a `switchLights()` function, which will take an intersection dictionary as an argument and switch the lights.

At first, you might think that `switchLights()` should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'. The code to implement this idea might look like this:

```
def switchLights(stoplight):
    for key in stoplight.keys():
```

```
if stoplight[key] == 'green':
    stoplight[key] = 'yellow'
elif stoplight[key] == 'yellow':
    stoplight[key] = 'red'
elif stoplight[key] == 'red':
    stoplight[key] = 'green'
```

```
switchLights(market_2nd)
```

You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it. When you finally do run the simulation, the program doesn't crash—but your virtual cars do!

Since you've already written the rest of the program, you have no idea where the bug could be. Maybe it's in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the `switchLights()` function.

But if while writing `switchLights()` you had added an assertion to check that *at least one of the lights is always red*, you might have included the following at the bottom of the function:

```
assert 'red' in stoplight.values(), 'Neither light is red!' + str(stoplight)
```

With this assertion in place, your program would crash with this error message:

Traceback (most recent call last):

```
File "carSim.py", line 14, in <module>
```

```
    switchLights(market_2nd)
```

```
File "carSim.py", line 13, in switchLights
```

```
    assert 'red' in stoplight.values(), 'Neither light is red!' +
```

```
    str(stoplight)
```

```
❶ AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

The important line here is the `AssertionError` ❶. While your program crashing is not ideal, it immediately points out that a sanity check failed: neither direction of traffic has a red light, meaning that traffic could be going both ways. By failing fast early in the program's execution, you can save yourself a lot of future debugging effort.

LOGGING

If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of *logging* to debug your code. Logging is a great way to understand what's happening in your program and in what order it's

happening. Python's logging module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

Using the logging Module

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the `#!/python shebang` line):

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)
s - %(message)s')
```

You don't need to worry too much about how this works, but basically, when Python logs an event, it creates a `LogRecord` object that holds information about that event. The logging module's `basicConfig()` function lets you specify what details about the `LogRecord` object you want to see and how you want those details displayed.

Say you wrote a function to calculate the *factorial* of a number. In mathematics, factorial 4 is $1 \times 2 \times 3 \times 4$, or 24. Factorial 7 is $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$, or 5,040. Open a new file editor tab and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as *factorialLog.py*.

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s
- %(message)s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total
```

```
print(factorial(5))
logging.debug('End of program')
```

Here, we use the `logging.debug()` function when we want to print log information. This `debug()` function will call `basicConfig()`, and a line of information will be printed. This information will be in the format we specified in `basicConfig()` and will include the messages we passed to `debug()`. The `print(factorial(5))` call is part of the original program, so the result is displayed even if logging messages are disabled.

The output of this program looks like this:

```
2019-05-23 16:20:12,664 - DEBUG - Start of program
2019-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2019-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2019-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2019-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2019-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2019-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2019-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2019-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2019-05-23 16:20:12,684 - DEBUG - End of program
```

The `factorial()` function is returning 0 as the factorial of 5, which isn't right. The `for` loop should be multiplying the value in `total` by the numbers from 1 to 5. But the log messages displayed by `logging.debug()` show that the `i` variable is starting at 0 instead of 1. Since zero times anything is zero, the rest of the iterations also have the wrong value for `total`. Logging messages provide a trail of breadcrumbs that can help you figure out when things started to go wrong.

Change the `for i in range(n + 1):` line to `for i in range(1, n + 1):`, and run the program again. The output will look like this:

```
2019-05-23 17:13:40,650 - DEBUG - Start of program
2019-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2019-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2019-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2019-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2019-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2019-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2019-05-23 17:13:40,661 - DEBUG - End of factorial(5)
```

The `factorial(5)` call correctly returns 120. The log messages showed what was going on inside the loop, which led straight to the bug.

You can see that the `logging.debug()` calls printed out not just the strings passed to them but also a timestamp and the word *DEBUG*.

Don't Debug with the print() Function

Typing `import logging` and `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` is somewhat unwieldy. You may want to use `print()` calls instead, but don't give in to this temptation! Once you're done debugging, you'll end up spending a lot of time removing `print()` calls from your code for each log message. You might even accidentally remove some `print()` calls that were being used for nonlog messages. The nice thing about log messages is that you're free to fill your program with as many as you like, and you can always disable them later by adding a single `logging.disable(logging.CRITICAL)` call. Unlike `print()`, the logging module makes it easy to switch between showing and hiding log messages.

Log messages are intended for the programmer, not the user. The user won't care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that. For messages that the user will want to see, like *File not found* or *Invalid input, please enter a number*, you should use a `print()` call. You don't want to deprive the user of useful information after you've disabled log messages.

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 11-1 from least to most important. Messages can be logged at each level using a different logging function.

Table 11-1: Logging Levels in Python

Level	Logging function	Description
DEBUG	<code>logging.debug()</code>	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.

Level	Logging function	Description
INFO	logging.info()	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	logging.warning()	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	logging.error()	Used to record an error that caused the program to fail to do something.
CRITICAL	logging.critical()	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Your logging message is passed as a string to these functions. The logging levels are suggestions. Ultimately, it is up to you to decide which category your log message falls into. Enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2019-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2019-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2019-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2019-05-18 19:05:07,737 - ERROR - An error has occurred.
```

```
>>> logging.critical('The program is unable to recover!')
```

```
2019-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

The benefit of logging levels is that you can change what priority of logging message you want to see. Passing `logging.DEBUG` to the `basicConfig()` function's level keyword argument will show messages from all the logging levels (`DEBUG` being the lowest level). But after developing your program some more, you may be interested only in errors. In that case, you can set `basicConfig()`'s level argument to `logging.ERROR`. This will show only `ERROR` and `CRITICAL` messages and skip the `DEBUG`, `INFO`, and `WARNING` messages.

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand. You simply pass `logging.disable()` a logging level, and it will suppress all log messages at that level or lower. So if you want to disable logging entirely, just add `logging.disable(logging.CRITICAL)` to your program. For example, enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format='%(asctime)s -
%(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2019-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since `logging.disable()` will disable all messages after it, you will probably want to add it near the `import logging` line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a filename keyword argument, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='
%(asctime)s - %(levelname)s - %(message)s')
```

The log messages will be saved to *myProgramLog.txt*. While logging messages are helpful, they can clutter your screen and make it hard to read the program's output. Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program. You can open this text file in any text editor, such as Notepad or TextEdit.

MU'S DEBUGGER

The *debugger* is a feature of the Mu editor, IDLE, and other editor software that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program “under the debugger” like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.

To run a program under Mu's debugger, click the **Debug** button in the top row of buttons, next to the Run button. Along with the usual output pane at the bottom, the Debug Inspector pane will open along the right side of the window. This pane lists the current value of variables in your program. In Figure 11-1, the debugger has paused the execution of the program just before it would have run the first line of code. You can see this line highlighted in the file editor.

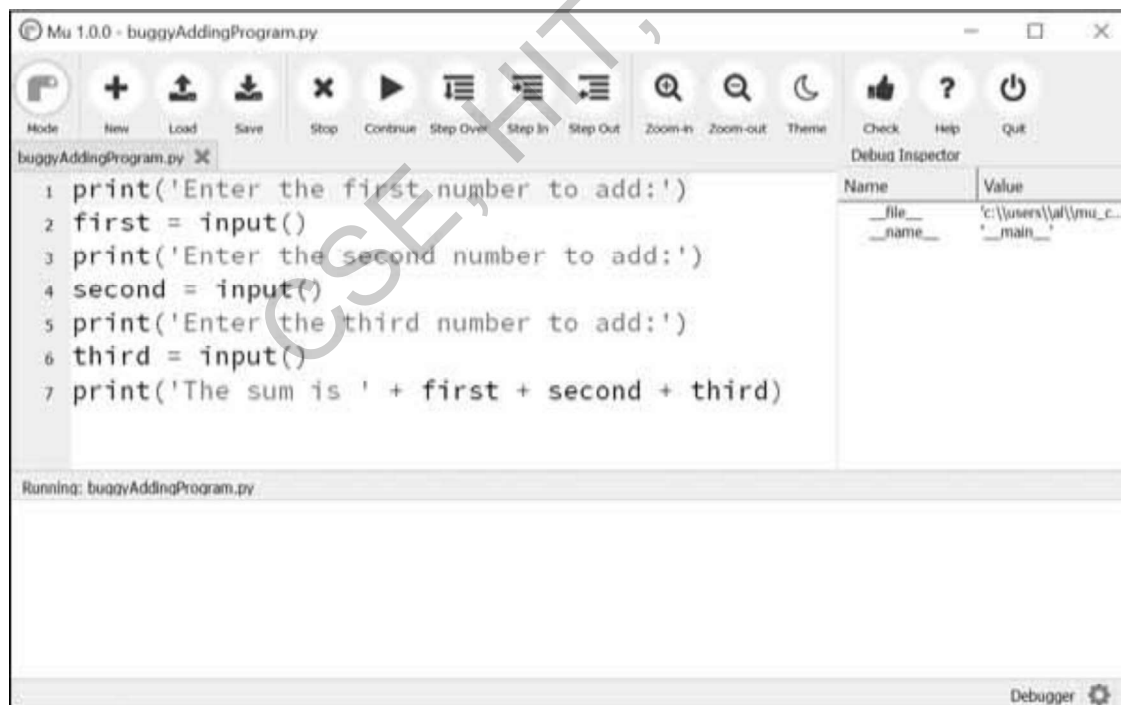


Figure 11-1: Mu running a program under the debugger

Debugging mode also adds the following new buttons to the top of the editor: Continue, Step Over, Step In, and Step Out. The usual Stop button is also available.

Continue

Clicking the Continue button will cause the program to execute normally until it terminates or reaches a *breakpoint*. (I will describe breakpoints later in this chapter.) If you are done debugging and want the program to continue normally, click the Continue button.

Step In

Clicking the Step In button will cause the debugger to execute the next line of code and then pause again. If the next line of code is a function call, the debugger will “step into” that function and jump to the first line of code of that function.

Step Over

Clicking the Step Over button will execute the next line of code, similar to the Step In button. However, if the next line of code is a function call, the Step Over button will “step over” the code in the function. The function’s code will be executed at full speed, and the debugger will pause as soon as the function call returns. For example, if the next line of code calls a `spam()` function but you don’t really care about code inside this function, you can click Step Over to execute the code in the function at normal speed, and then pause when the function returns. For this reason, using the Over button is more common than using the Step In button.

Step Out

Clicking the Step Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step In button and now simply want to keep executing instructions until you get back out, click the Out button to “step out” of the current function call.

Stop

If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Stop button. The Stop button will immediately terminate the program.

Debugging a Number Adding Program

Open a new file editor tab and enter the following code:

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
```

```
third = input()
print("The sum is " + first + second + third)
```

Save it as *buggyAddingProgram.py* and run it first without the debugger enabled. The program will output something like this:

Enter the first number to add:

5

Enter the second number to add:

3

Enter the third number to add:

42

The sum is 5342

The program hasn't crashed, but the sum is obviously wrong. Run the program again, this time under the debugger.

When you click the Debug button, the program pauses on line 1, which is the line of code it is about to execute. Mu should look like Figure 10-1.

Click the **Step Over** button once to execute the first `print()` call. You should use Step Over instead of Step In here, since you don't want to step into the code for the `print()` function. (Although Mu should prevent the debugger from entering Python's built-in functions.) The debugger moves on to line 2, and highlights line 2 in the file editor, as shown in Figure 11-2. This shows you where the program execution currently is.

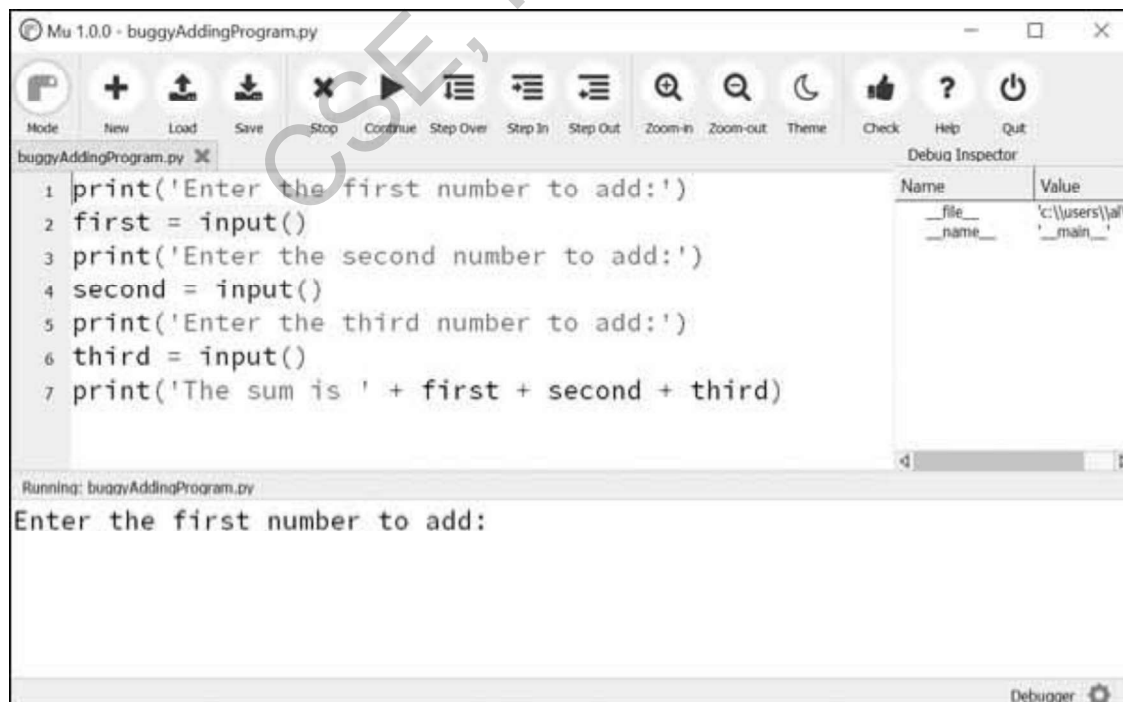


Figure 11-2: The Mu editor window after clicking Step Over

Click **Step Over** again to execute the `input()` function call. The highlighting will go away while Mu waits for you to type something for the `input()` call into the output pane. Enter 5 and press ENTER. The highlighting will return.

Keep clicking **Step Over**, and enter 3 and 42 as the next two numbers. When the debugger reaches line 7, the final `print()` call in the program, the Mu editor window should look like Figure 11-3.

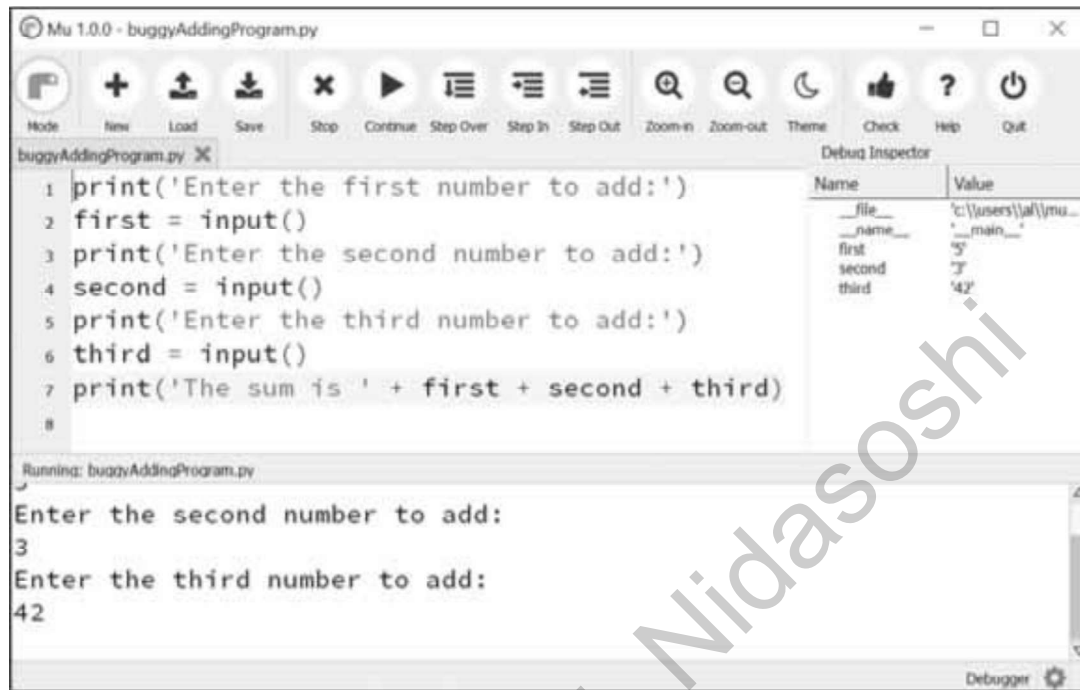


Figure 11-3: The Debug Inspector pane on the right side shows that the variables are set to strings instead of integers, causing the bug.

In the Debug Inspector pane, you should see that the `first`, `second`, and `third` variables are set to string values `'5'`, `'3'`, and `'42'` instead of integer values 5, 3, and 42. When the last line is executed, Python concatenates these strings instead of adding the numbers together, causing the bug.

Stepping through the program with the debugger is helpful but can also be slow. Often you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

Breakpoints

A *breakpoint* can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line. Open a new file editor tab and enter the following program, which simulates flipping a coin 1,000 times. Save it as `coinFlip.py`.

```
import random
heads = 0
```

```
for i in range(1, 1001):
    ❶ if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
        ❷ print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

The `random.randint(0, 1)` call ❶ will return 0 half of the time and 1 the other half of the time. This can be used to simulate a 50/50 coin flip where 1 represents heads. When you run this program without the debugger, it quickly outputs something like the following:

```
Halfway done!
Heads came up 490 times.
```

If you ran this program under the debugger, you would have to click the Step Over button thousands of times before the program terminated. If you were interested in the value of `heads` at the halfway point of the program's execution, when 500 of 1,000 coin flips have been completed, you could instead just set a breakpoint on the line `print('Halfway done!')` ❷. To set a breakpoint, click the line number in the file editor to cause a red dot to appear, marking the breakpoint like in Figure 11-4.



Figure 11-4: Setting a breakpoint causes a red dot (circled) to appear next to the line number.

You don't want to set a breakpoint on the `if` statement line, since the `if` statement is executed on every single iteration through the loop. When you set the breakpoint on the code in the `if` statement, the debugger breaks only when the execution enters the `if` clause.

The line with the breakpoint will have a red dot next to it. When you run the program under the debugger, it will start in a paused state at the first line, as usual. But if you click Continue, the program will run at full speed until it reaches the line with the

breakpoint set on it. You can then click Continue, Step Over, Step In, or Step Out to continue as normal.

If you want to remove a breakpoint, click the line number again. The red dot will go away, and the debugger will not break on that line in the future.

SUMMARY

Assertions, exceptions, logging, and the debugger are all valuable tools to find and prevent bugs in your program. Assertions with the Python `assert` statement are a good way to implement “sanity checks” that give you an early warning when a necessary condition doesn’t hold true. Assertions are only for errors that the program shouldn’t try to recover from and should fail fast. Otherwise, you should raise an exception.

An exception can be caught and handled by the `try` and `except` statements. The logging module is a good way to look into your code while it’s running and is much more convenient to use than the `print()` function because of its different logging levels and ability to log to a text file.

The debugger lets you step through your program one line at a time. Alternatively, you can run your program at normal speed and have the debugger pause execution whenever it reaches a line with a breakpoint set. Using the debugger, you can see the state of any variable’s value at any point during the program’s lifetime.

These debugging tools and techniques will help you write programs that work. Accidentally introducing bugs into your code is a fact of life, no matter how many years of coding experience you have.

PRACTICE QUESTIONS

1. Write an `assert` statement that triggers an `AssertionError` if the variable `spam` is an integer less than 10.
2. Write an `assert` statement that triggers an `AssertionError` if the variables `eggs` and `bacon` contain strings that are the same as each other, even if their cases are different (that is, `'hello'` and `'hello'` are considered the same, and `'goodbye'` and `'GOODbye'` are also considered the same).
3. Write an `assert` statement that *always* triggers an `AssertionError`.
4. What are the two lines that your program must have in order to be able to call `logging.debug()`?
5. What are the two lines that your program must have in order to have `logging.debug()` send a logging message to a file named `programLog.txt`?
6. What are the five logging levels?

7. What line of code can you add to disable all logging messages in your program?
8. Why is using logging messages better than using `print()` to display the same message?
9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?
10. After you click Continue, when will the debugger stop?
11. What is a breakpoint?
12. How do you set a breakpoint on a line of code in Mu?

PRACTICE PROJECT

For practice, write a program that does the following.

Debugging Coin Toss

The following program is meant to be a simple coin toss guessing game. The player gets two guesses (it's an easy game). However, the program has several bugs in it. Run through the program a few times to find the bugs that keep the program from working correctly.

```
import random
guess = ""
while guess not in ('heads', 'tails'):
    print('Guess the coin toss! Enter heads or tails:')
    guess = input()
toss = random.randint(0, 1) # 0 is tails, 1 is heads
if toss == guess:
    print('You got it!')
else:
    print('Nope! Guess again!')
    guessss = input()
    if toss == guess:
        print('You got it!')
    else:
        print('Nope. You are really bad at this game.')
```