MODULE 1 INTRODUCTION TO PYTHON

CHAPTER 1

BASICS OF PYTHON

Python is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language. Python is dynamically-typed and garbage-collected programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

Python supports multiple programming paradigms, including Procedural, Object Oriented and Functional programming language. Python design philosophy emphasizes code readability with the use of significant indentation.

Why to Learn Python?

Python is consistently rated as one of the world's most popular programming languages. Python is fairly easy to learn, so if you are starting to learn any programming language then Python could be your great choice. Today various Schools, Colleges and Universities are teaching Python as their primary programming language. There are many other good reasons which makes Python as the top choice of any programmer:

- Python is Open Source which means its available free of cost.
- Python is simple and so easy to learn
- Python is versatile and can be used to create many different things.
- Python has powerful development libraries include AI, ML etc.
- Python is much in demand and ensures high salary

Python is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning Python:

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

- **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python Online Compiler/Interpreter

Python Online Compiler/Interpreter are available which helps us to **Edit** and **Execute** the code directly from the browser.

Careers with Python

If you know Python nicely, then you have a great career ahead. Here are just a few of the career options where Python is a key skill:

- Game developer
- Web designer
- Python developer
- Full-stack developer
- Machine learning engineer
- Data scientist
- Data analyst
- Data engineer
- DevOps engineer
- Software engineer
- Many more other roles

Characteristics of Python

Following are important characteristics of **Python Programming** −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Applications of Python

The latest release of Python is 3.x. As mentioned before, Python is one of the most widely used language over the web.

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python has a big list of good features, few are listed below −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic

type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

STARTING IDLE

This book uses Mu as an editor and interactive shell. However, you can use any number of editors for writing Python code. The *Integrated Development and Learning Environment (IDLE)* software installs along with Python, and it can serve as a second editor.

- On Windows 7 or later, click the Start icon in the lower-left corner of your screen, enter **IDLE** in the search box, and select **IDLE (Python GUI)**.
- On macOS, open the Finder window, click **Applications**, click **Python 3.8**, and then click the IDLE icon.
- On Ubuntu, select **Applications** ▸ **Accessories** ▸ **Terminal** and then enter **idle3**. (You may also be able to click **Applications** at the top of the screen, select **Programming**, and then click **IDLE 3**.)

THE INTERACTIVE SHELL

When you run Mu, the window that appears is called the *file editor* window. You can open the *interactive shell* by clicking the REPL button. A shell is a program that lets you type instructions into the computer, much like the Terminal or Command Prompt on macOS and Windows, respectively. Python's interactive shell lets you enter instructions for the Python interpreter software to run. The computer reads the instructions you enter and runs them immediately.

For example, enter the following into the interactive shell next to the prompt:

```
>>> print('Hello, world!')
```

After you type that line and press ENTER, the interactive shell should display this in response:

```
>>> print('Hello, world!')
Hello, world!
```

ENTERING EXPRESSIONS INTO THE INTERACTIVE SHELL

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.

A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

We 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

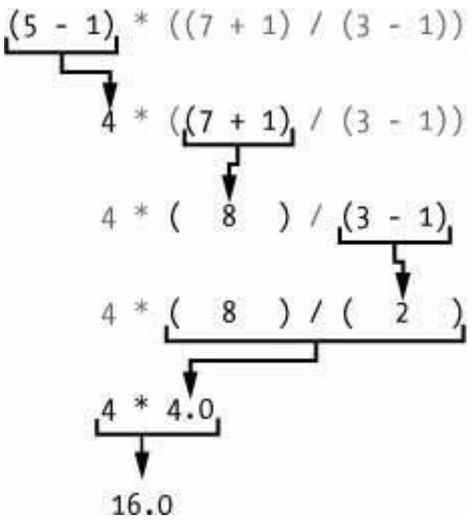
Operator	Operation	Example	Evaluates to . . .
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division /floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

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.

Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2     +          2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Python will keep evaluating parts of the expression until it becomes a single value, as shown here:



If we enter a bad Python instruction, Python won't be able to understand it and will display a Syntax 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
```

THE INTEGER, FLOATING-POINT, AND STRING DATA TYPES

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 or double quote (“) (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*.

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'.

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.

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'
```

STORING VALUES IN VARIABLES

A *variable* is like a box in the computer's memory where you can store a single value.

Assignment Statements

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.

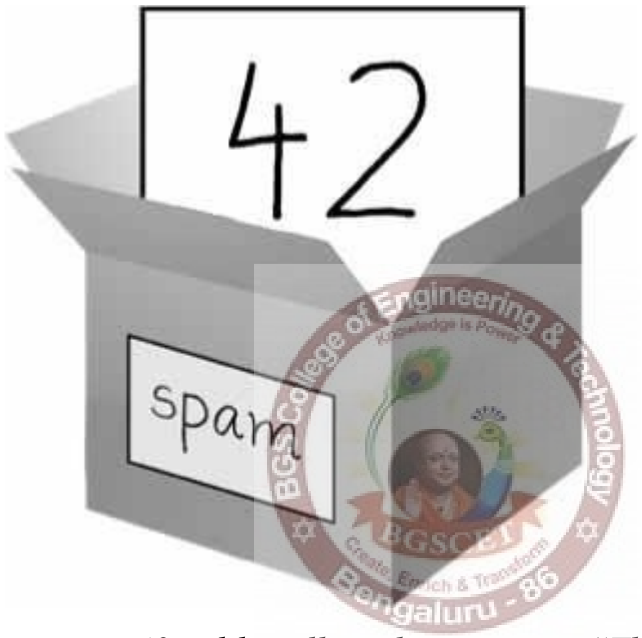


Figure 1-1: spam = 42 is like telling the program, "The variable spam now has the integer value 42 in it."

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'
```

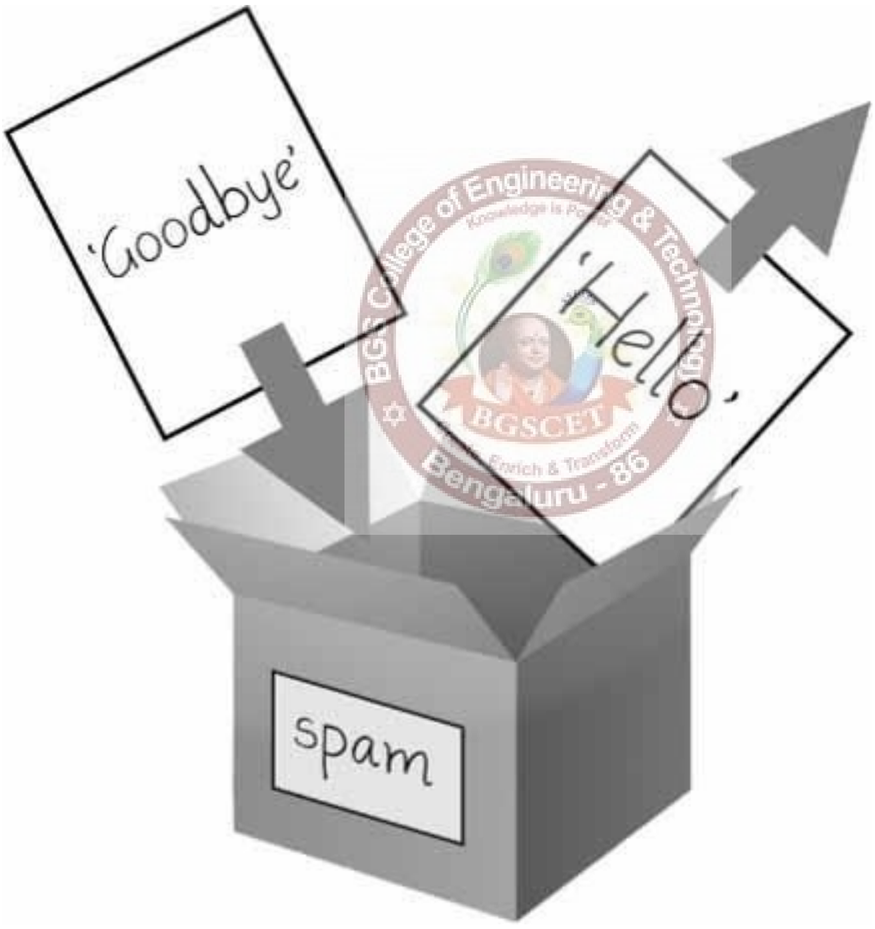


Figure 1-2: When a new value is assigned to a variable, the old one is forgotten.

Variable Names

A good variable name describes the data it contains. 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
current_balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
account4	4account (can't begin with a number)
_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.

Python uses *camelcase* for variable names instead of underscores; that is, variables lookLikeThis instead of looking_like_this.

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.
- 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.')
```

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.
>>>
```

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.

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*.

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'.

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.

```
❺ 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
```

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.

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.

```
>>> 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
```

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

```
>>> 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.

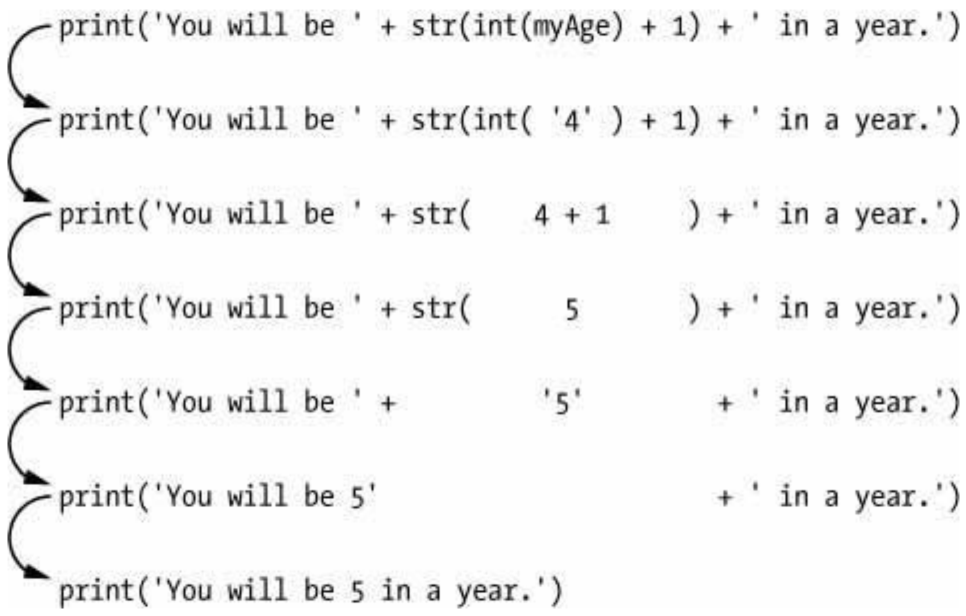
```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: 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
```

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

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.')
```

MODULE 1- CHAPTER 2

FLOW CONTROL

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.

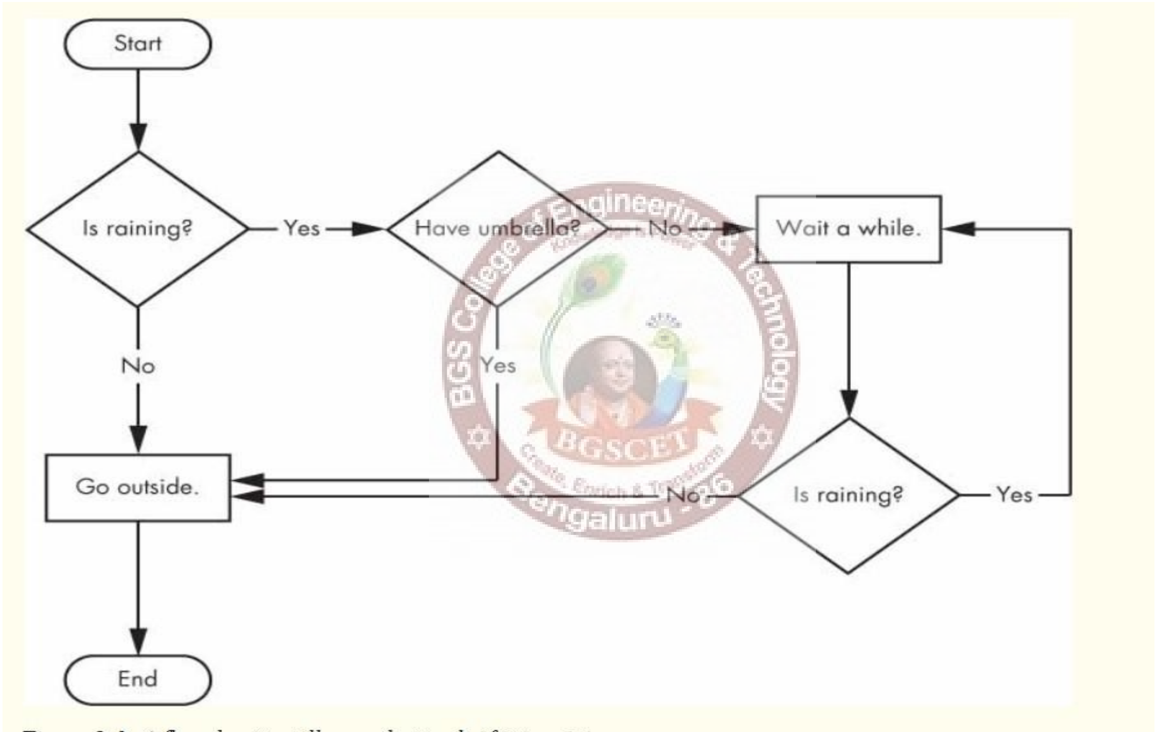


Figure 2-1: A flowchart to tell you what to do if it is raining

BOOLEAN 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.

```
>>> 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
```

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**
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
>>> 'hello' ==
  'hello'True
  >>> 'hello' ==
  'Hello'False
  >>> 'dog' != 'cat'
  True
  >>> True == True
  True
  >>> True != False
  True
  >>> 42 == 42.0
  True
➊ >>> 42 == ‘42’
  False
>>> 42 < 100
  True
  >>> 42 > 100
  False
  >>> 42 < 42
  False
  >>> eggCount = 42
➊ >>> eggCount <= 42
  True
  >>> myAge = 29
➋ >>> myAge >= 10
  True
```

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.

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.

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

Table 2-2: The and Operator’s Truth Table

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

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
```

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 and 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.

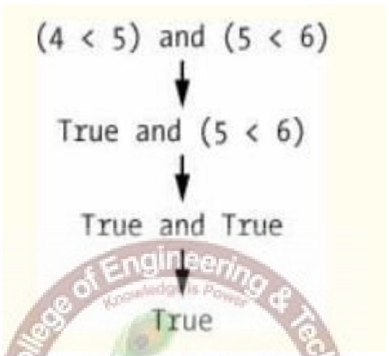
```
 >>> not True
   False
❶ >>> not not not not True
   True
 >>> not True
   False
❶ >>> not not not not True
   True
```

Mixing Boolean and Comparison Operators

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

```
>>> (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:



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

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*.

Conditions

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.

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

Flow Control Statements

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 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)

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

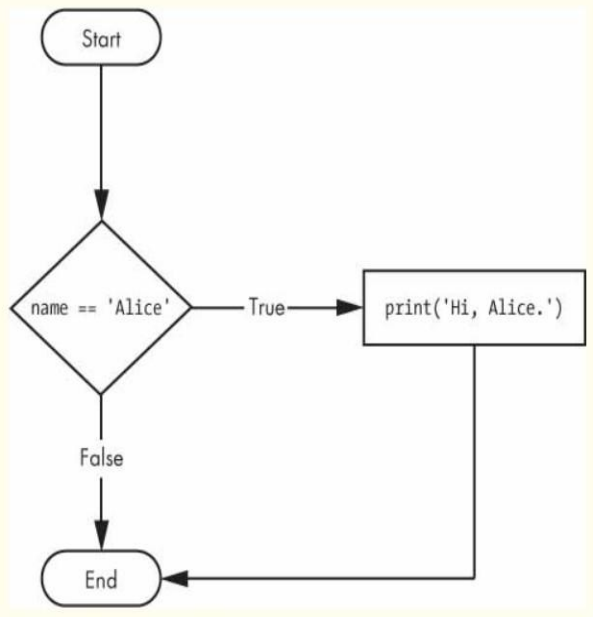


Figure 2-2: The flowchart for an if *statement*

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)

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

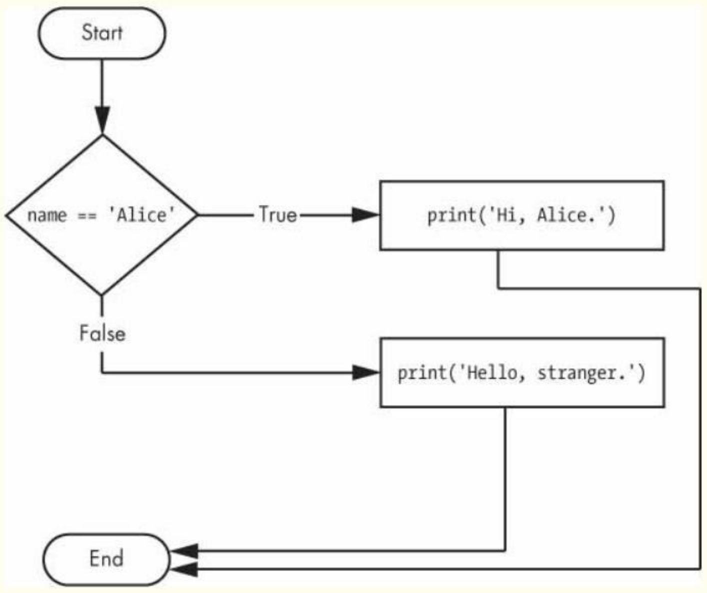


Figure 2-3: The flowchart for an else statement

elif Statements

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.’)
```

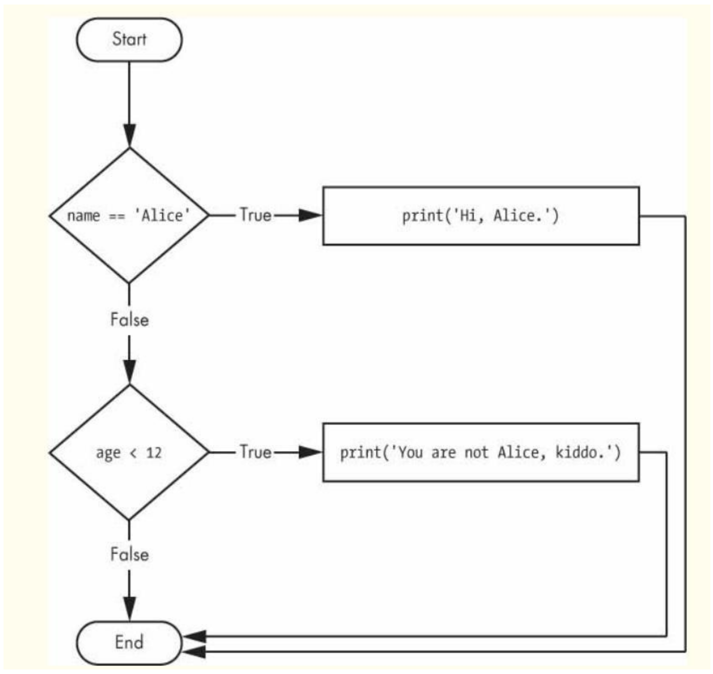
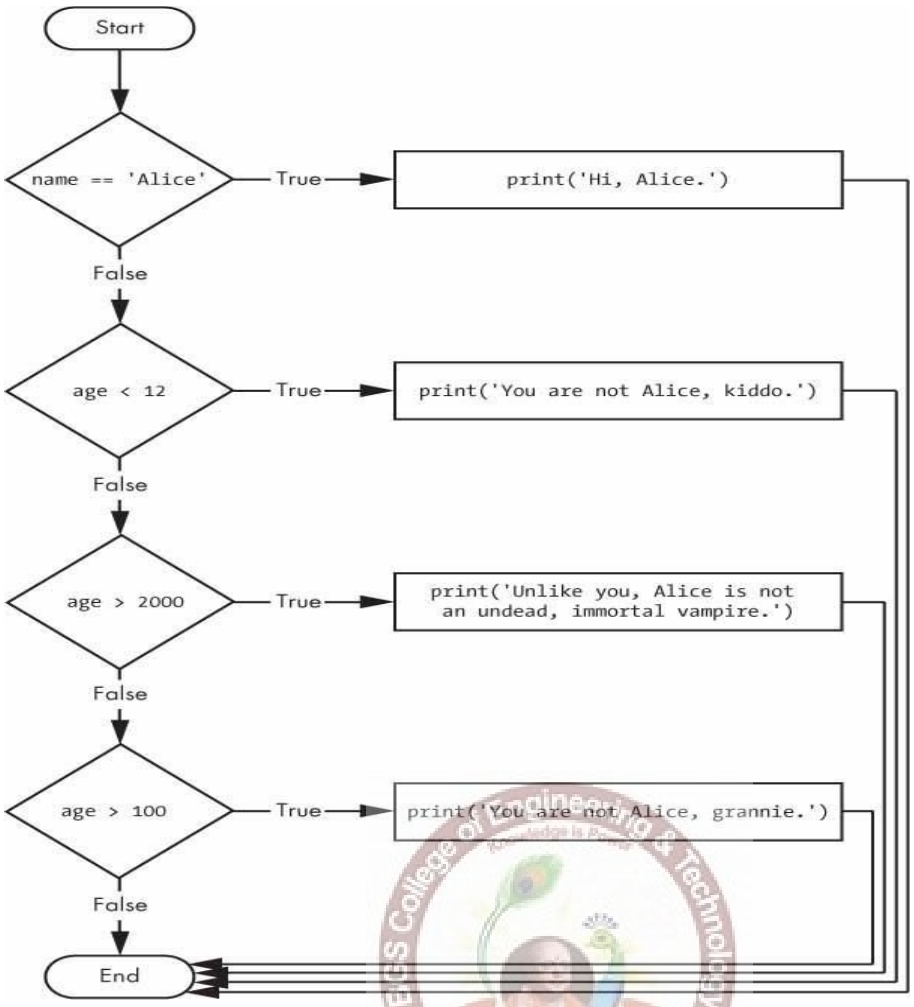
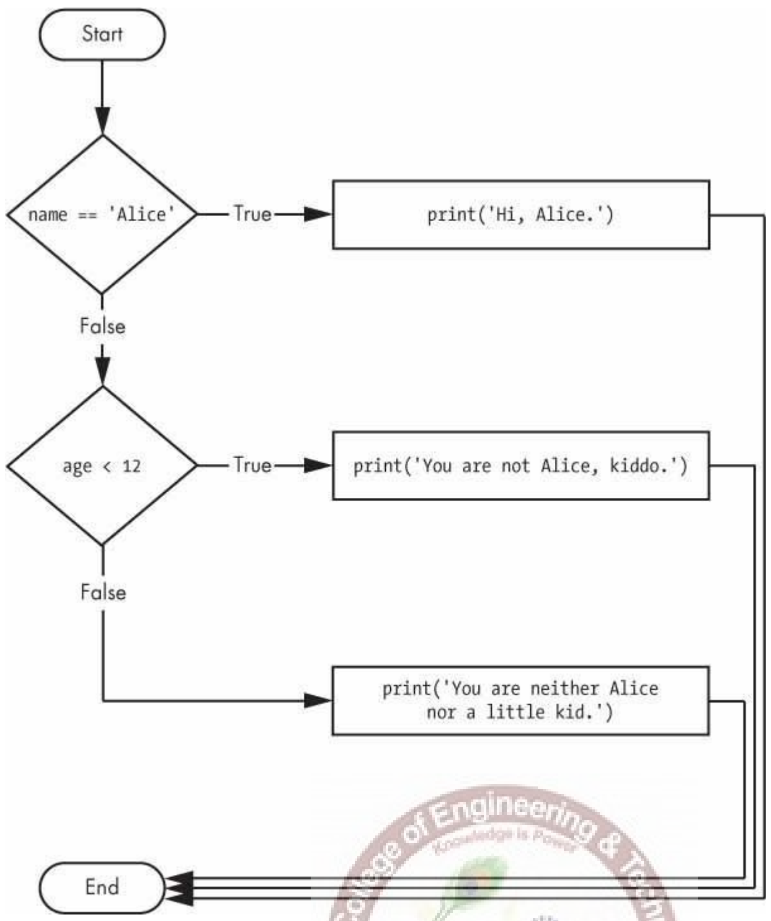


Figure 2-4: The flowchart for an elif *statement*

```
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.')
```


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
age > 100
True
print('You are not Alice, grannie.')
False
End

```
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.')
```


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
print('You are neither Alice nor a little kid.')
End

while Loop Statements

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)

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*.

```
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
```

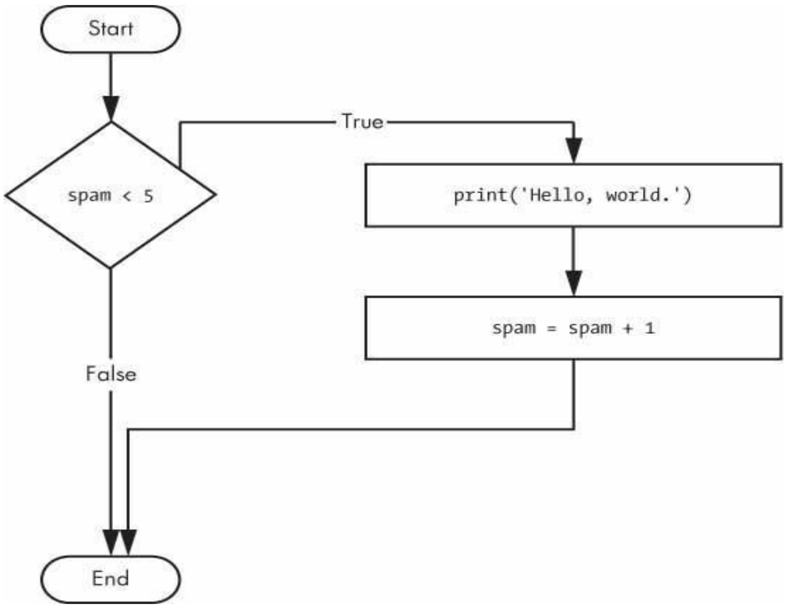


Figure 2-8: The flowchart for the if *statement code*

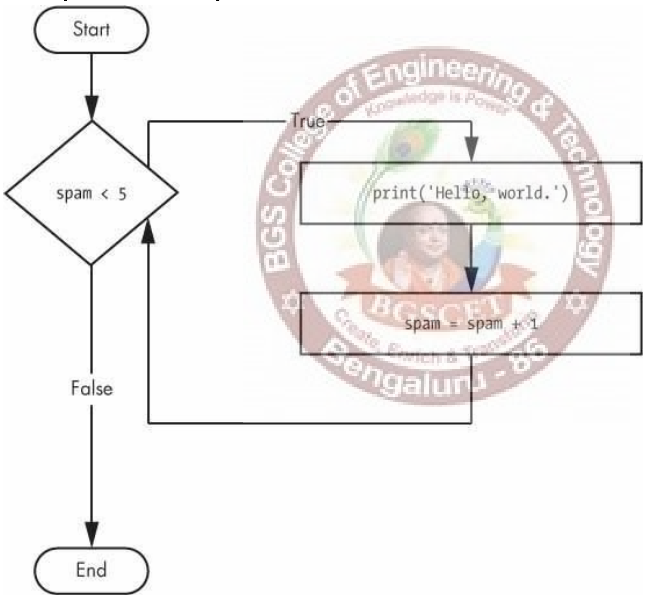
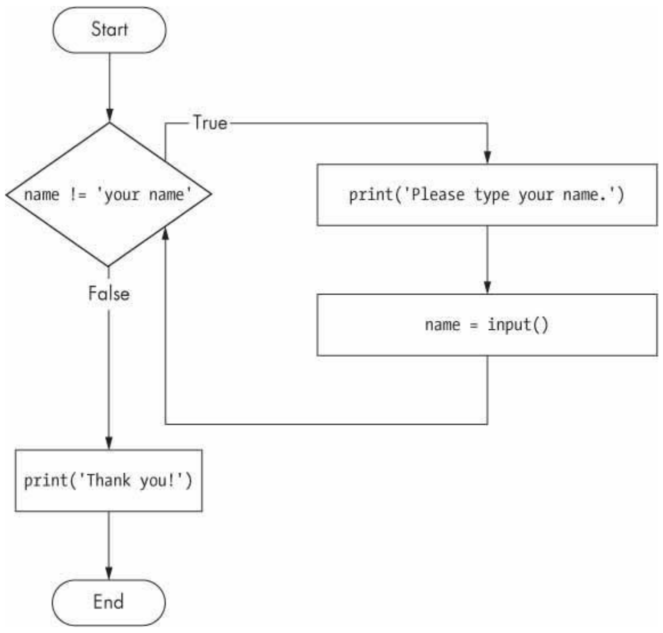


Figure 2-9: The flowchart for the while *statement code*

An Annoying while Loop

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

❹ print('Thank you!')



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.

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

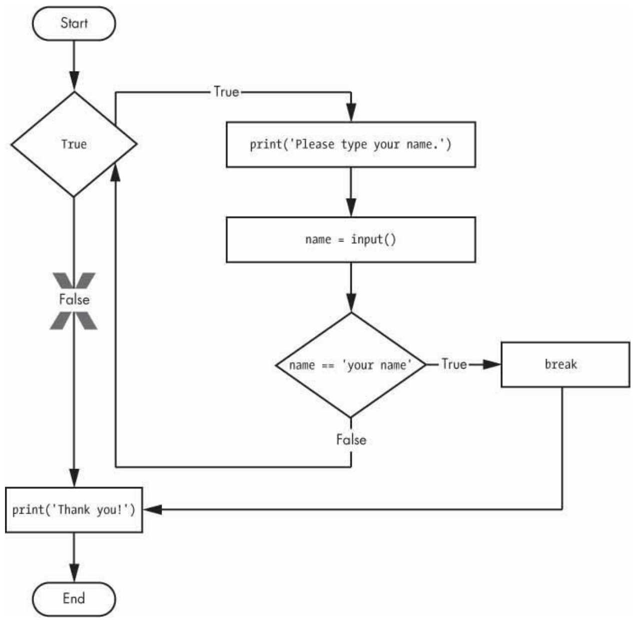


Figure 2-11: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always* True.

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.

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

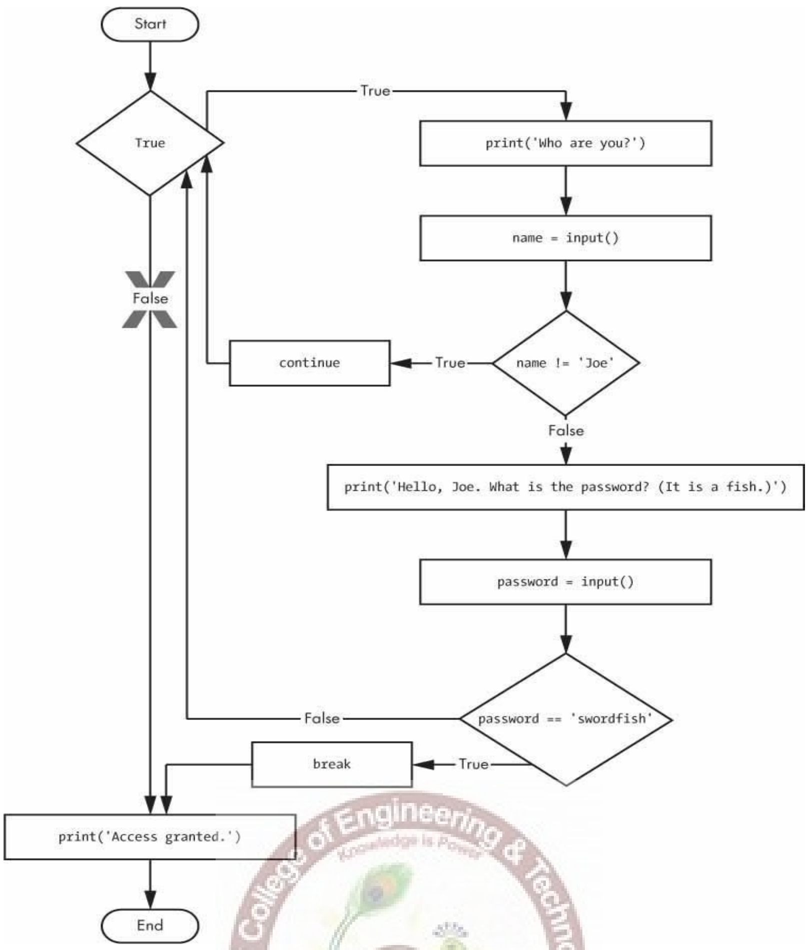


Figure 2-12: A flowchart for swordfish.py. *The X path will logically never happen, because the loop condition is always* True.

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)

A *function* is like a miniprogram within a program.

```
Def hello():
   ❷ print('Howdy!')
     print('Howdy!!!')
     print('Hello there.')

❸ hello()
  hello()
  hello()
```

A major purpose of functions is to group code that gets executed multiple times.

DEF STATEMENTS WITH PARAMETERS

```
print('My      name
is')    for    i    in
range(5):
   print('Jimmy  Five  Times  ('  +  str(i)  +
')')My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

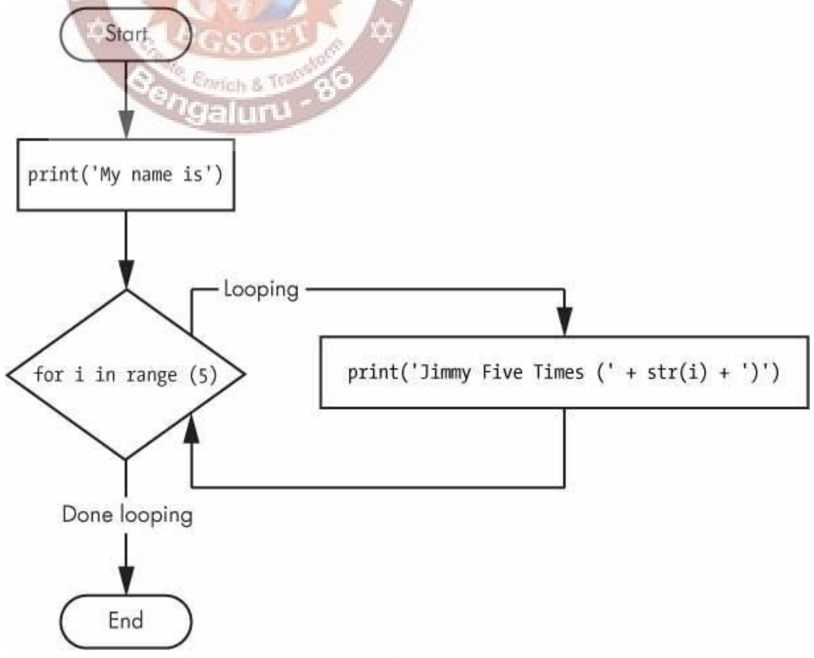


Figure 2-13: The flowchart for fiveTimes.py

```
total = 0
❷ for num in range(101):
    ❸ total = total + num
❹ print(total)
print(‘My name
is’)i = 0
while i < 5:
    print(‘Jimmy Five Times (‘ + str(i) +
    ‘)’)i = i + 1
```

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
```

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

This for loop would have the following output:

```
5
4
3
2
1
0
```

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

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.

```
Import random
for i in range(5):
   print(random.randint(1, 10))
```

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

```
4
1
8
4
1
```

import random, sys, os, math

Now we can use any of the functions in these four modules.

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.

Ending a Program Early with the sys.exit() Function

We 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.

```
Import   sys

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

MODULE 1
CHAPTER 3 FUNCTIONS

A *function* is like a mini program within a program.

```
def hello():
 ❷ print('Howdy!')
    print('Howdy!!!')
    print('Hello there.')

❸ hello()
  hello()
  hello()
```

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

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

DEF STATEMENTS WITH PARAMETERS

When we call the print() or len() function, we pass them values, called arguments, by typing them between the parentheses.

```
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
```

RETURN VALUES AND RETURN STATEMENTS

When we 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 we 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, we 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.

```
    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)
```

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. Just like the Boolean True and False values, None must be typed with a capital *N*.

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

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).

```
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
```

Similarly, when we 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 we 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
```

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.

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.

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.

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()
```

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.

Global Variables Can Be Read from a Local Scope

Consider the following program:

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

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.

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.

```
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
```

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.”

```
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
```

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.
-

```
    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)
```

```
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
```

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))
```

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
```

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
```

A SHORT PROGRAM: GUESS THE NUMBER

```
# 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!
```

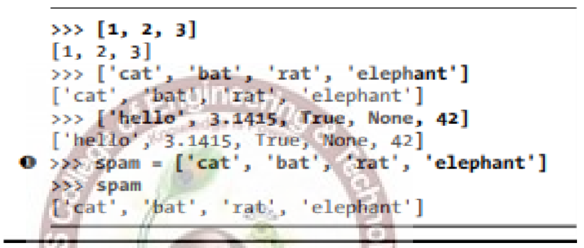
```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + '
guesses!')
else:
       print('Nope. The number I was thinking of was ' + str(secretNumber))
```

```
    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!
```

MODULE-2 CHAPTER 1: LISTS

The List Data Type

- ➢ A list is a value that contains multiple values in an ordered sequence.
- ➢ A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].
- ➢ A list begins with an opening square bracket and ends with a closing square bracket, [].
- ➢ Values inside the list are also called items and are separated with commas.

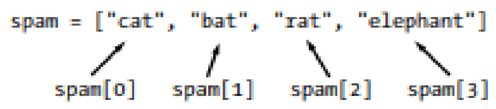


```
  >>> [1, 2, 3]
  [1, 2, 3]
  >>> ['cat', 'bat', 'rat', 'elephant']
  ['cat', 'bat', 'rat', 'elephant']
  >>> ['hello', 3.1415, True, None, 42]
  ['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
  >>> spam
  ['cat', 'bat', 'rat', 'elephant']
```

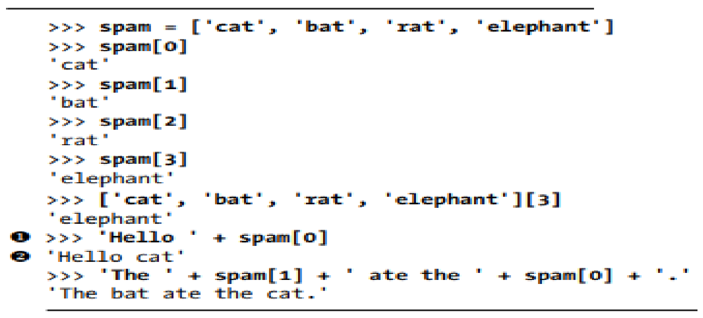
- ➢ The spam variable ❶ is still assigned only one value: the list value(contains multiple values).
- ➢ The value [] is an empty list that contains no values, similar to '', the empty string.

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.

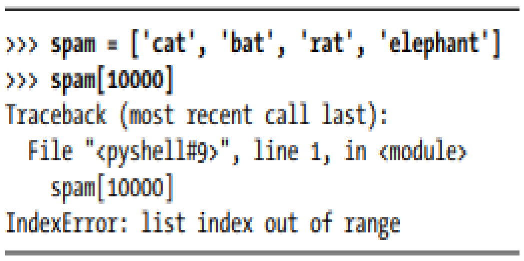


- ➢ The first value in the list is at index 0, the second value is at index 1, and the third value is at index 2, and so on.
- ➢ For example, type the following expressions into the interactive shell.



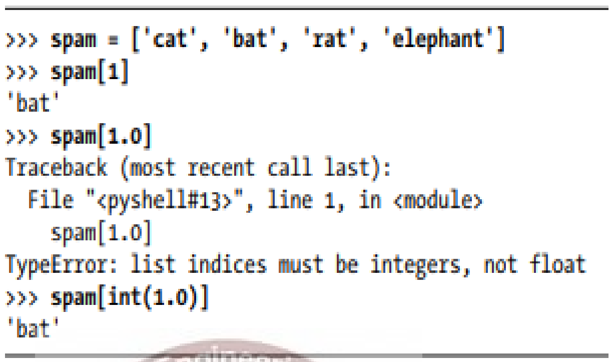
```
  >>> spam = ['cat', 'bat', 'rat', 'elephant']
  >>> spam[0]
  'cat'
  >>> spam[1]
  'bat'
  >>> spam[2]
  'rat'
  >>> spam[3]
  'elephant'
  >>> ['cat', 'bat', 'rat', 'elephant'][3]
  'elephant'
❶ >>> 'Hello ' + spam[0]
❷ 'Hello cat'
  >>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
  'The bat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

- The expression 'Hello ' + spam[0] evaluates to 'Hello ' + 'cat' because spam[0] evaluates to the string 'cat'. This expression in turn evaluates to the string value 'Hello cat'.
- If we use an index that exceeds the number of values in the list value then, python gives IndexError.



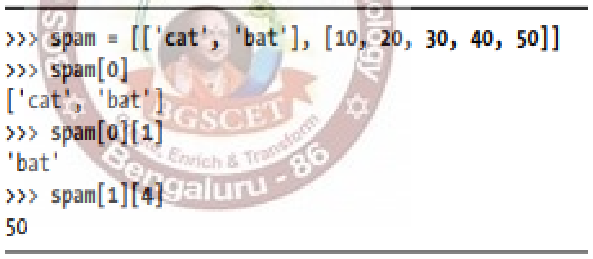
```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

- 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, 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.

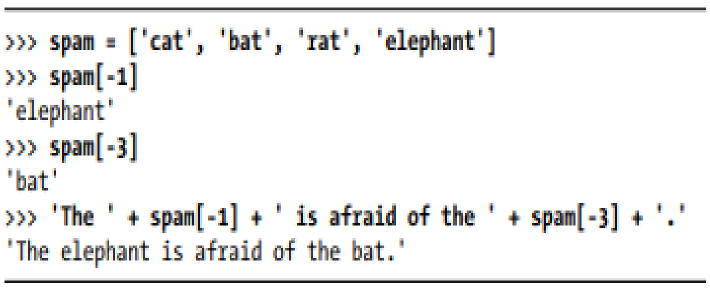


```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

- The first index dictates which list value to use, and the second indicates the value within the list value. **Ex**, spam[0][1] prints 'bat', the second value in the first list.

Negative Indexes

- We 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.



```
>>> 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 Sublists with Slices

- An index will 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.
- **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 (but will not include the value at the second index).

```
>>> 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, we 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.

```
>>> 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 len()

- The len() function will return the number of values that are in a list value.

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

- We can also use an index of a list to change the value at that index. **Ex:** spam[1] = 'aardvark' means "Assign the value at index 1 in the list spam to the string 'aardvark'."

```
>>> 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

- ➢ The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value.
- ➢ The * operator can also be used with a list and an integer value to replicate the list.

```
>>> [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.

```
>>> 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 to delete a variable After deleting if we try to use the variable, we 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 we first begin writing programs, it’s tempting to create many individual variables to store a group of similar values.

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

➢ Which is bad way to write code because it leads to have a duplicate code in the program.

```
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, we can use a single variable that contains a list value.

➢ **For Ex:** The following program uses a single list and it can store any number of cats that the user types in.

➢ Program:

```
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)
```

➢ Output:

```
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
```

➢ The benefit of using a list is that our data is now in a structure, so our program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

➢ A for loop repeats the code block once for each value in a list or list-like value.

Program

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

Output:

```
0
1
2
3
```

➢ A common Python technique is to use range (len(someList)) with a for loop to iterate over the indexes of a list.

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', '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: flame-throwers
Index 3 in supplies is: binders
```

➢ The code in the loop will access the index (as the variable i), the value at that index (as supplies[i]) and range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items it contains.

The in and not in Operators

➢ We can determine whether a value is or isn't in a list with the in and not in 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 and these expressions will evaluate to a Boolean value.

```
>>> '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
```

➢ 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.

Program

```
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.')
```

Output

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

The Multiple Assignment Trick

➤ The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code.

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

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

➤ Instead of left-side program we could type the right-side program to assignment multiple variables but the number of variables and the length of the list must be exactly equal, or Python will give you a ValueError:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

Augmented Assignment Operators

➤ When assigning a value to a variable, we will frequently use the variable itself.

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

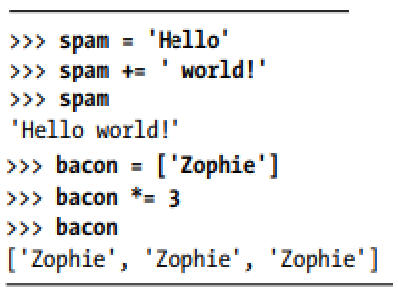
```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

➤ Instead of left-side program we could use right-side program i.e., with the augmented assignment operator += to do the same thing as a shortcut.

➤ The Augmented Assignment Operators are listed in the below table:

Augmented assignment statement	Equivalent assignment statement
spam = spam + 1	spam += 1
spam = spam - 1	spam -= 1
spam = spam * 1	spam *= 1
spam = spam / 1	spam /= 1
spam = spam % 1	spam %= 1

➤ The += operator can also do string and list concatenation, and the *= operator can do string and list replication.



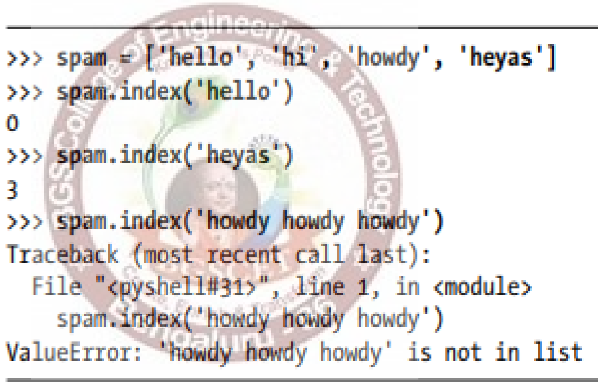
```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Methods

- A method is same as a function, except it is “called on” a value.
- The method part comes after the value, separated by a period.
- Each data type has its own set of methods.
- The list data type has several useful methods for finding, adding, removing, and manipulating values in a list.

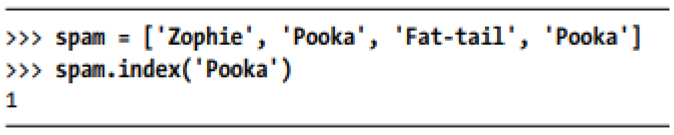
Finding a Value in a List with the index() Method

- 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.



```
>>> 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
```

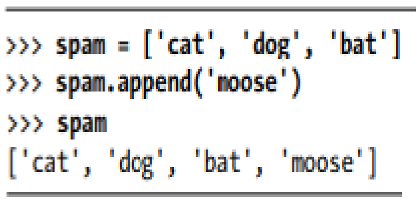
- When there are duplicates of the value in the list, the index of its first appearance is returned.



```
>>> 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.
- The append() method call adds the argument to the end of the list.



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

- 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.

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

- 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.

```
>>> 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')
AttributeError: 'int' object has no attribute 'insert'
```

<u>Removing Values from Lists with remove()</u>

- The remove() method is passed the value to be removed from the list it is called on.

```
>>> 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.

```
>>> 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.

```
>>> 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 good 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.

```
>>> 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.

```
>>> 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 return value by writing code like spam = spam.sort().
  - **Second,** we cannot sort lists that have both number values and string values in them.

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

  - **Third,** sort() uses "ASCIIbetical order(upper case)" rather than actual alphabetical order(lower case) for sorting strings.

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

- If we 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']
```

Example Program: Magic 8 Ball with a List

- We can write a much more elegant version of the Magic 8 Ball program. Instead of several lines of nearly identical elif statements, we can create a single list.

```
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)])
```

- The expression you use as the index into 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.

Exceptions to Indentation Rules in Python

- The amount of indentation for a line of code tells Python what block it is in.
- lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished.

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

- We can also split up a single instruction across multiple lines using the \ line continuation character at the end.

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

List-like Types: Strings and Tuples

- Lists aren't the only data types that represent ordered sequences of values.
- **Ex,** we can also do these with strings: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators.

```
>>> 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

String

- However, a string is immutable: It cannot be changed. Trying to reassign a single character in a string results in a TypeError error.

```
>>> 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.

```
>>> 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.

List

- A list value is a mutable data type: It can have values added, removed, or changed.

```
>>> 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]).

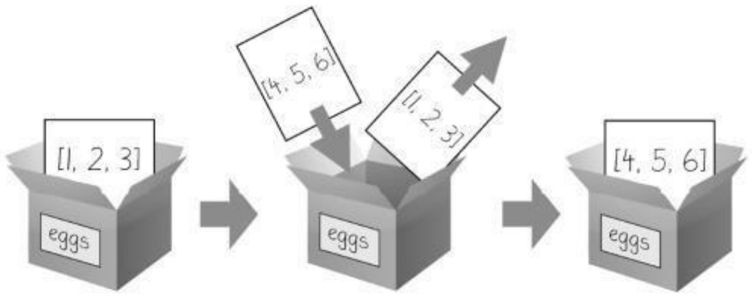


Figure: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

- If we want to modify the original list in eggs to contain [4, 5, 6], you would have to delete the items in that and then add items to it.

```
>>> 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]
```

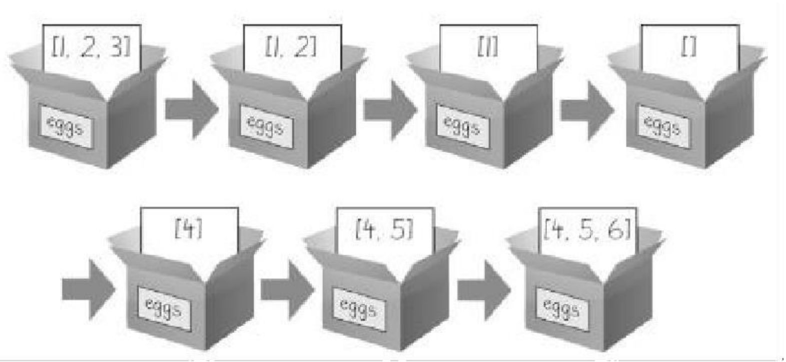


Figure: The del statement and the append() method modify the same list value in place.

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].

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

- **Second**, benefit of using tuples instead of lists is that, because they are immutable and their contents don't change. Tuples cannot have their values modified, appended, or removed.

```
>>> 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.

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

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

- The functions list() and tuple() will return list and tuple versions of the values passed to them.

```
>>> 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 we've seen, variables store strings and integer values.

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

- We assign 42 to the spam variable, and then we copy the value in spam and assign it to the variable cheese. When we later change the value in spam to 100, this doesn't affect the value in cheese. This is because spam and cheese are different variables that store different values.
- But lists works differently. When we assign a list to a variable, we are actually assigning a list reference to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list.

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

- When we create the list ❶, we assign a reference to it in the spam variable. But the next line 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 we modify the first element of cheese, we are modifying the same list that spam refers to.
- List variables don't actually contain lists—they contain references to lists.

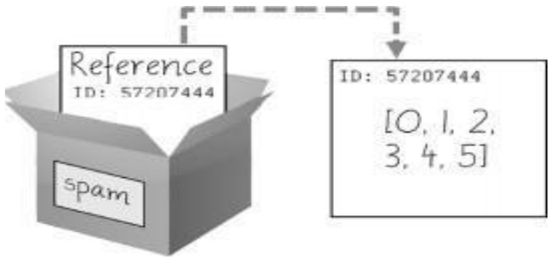


Figure: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

- The reference in spam is copied to cheese. Only a new reference was created and stored in cheese, not a new list.

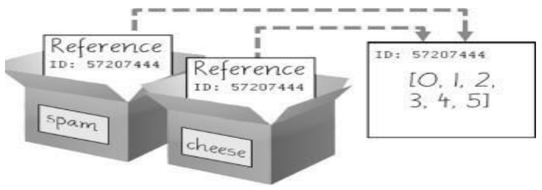


Figure: spam = cheese copies the reference, not the list

- When we 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.

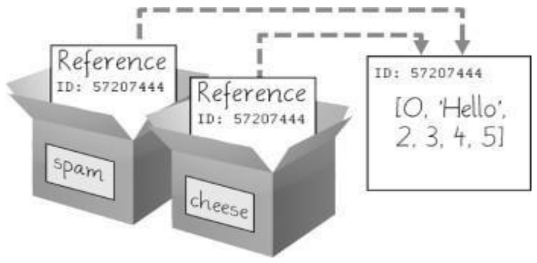


Figure: cheese[1] = 'Hello!' modifies the list that both variables refer to

- Variables will contain references to list values rather than list values themselves.
- But for strings and integer values, variables will contain the string or integer value.
- Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself.

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.

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

```
[1, 2, 3, 'Hello']
```

Program **Output**

- when eggs() is called, a return value is not used to assign a new value to spam.
- 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.

The copy Module's copy() and deepcopy() Functions

- If the function modifies the list or dictionary that is passed, we 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.
- **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.
- 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.
- The reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

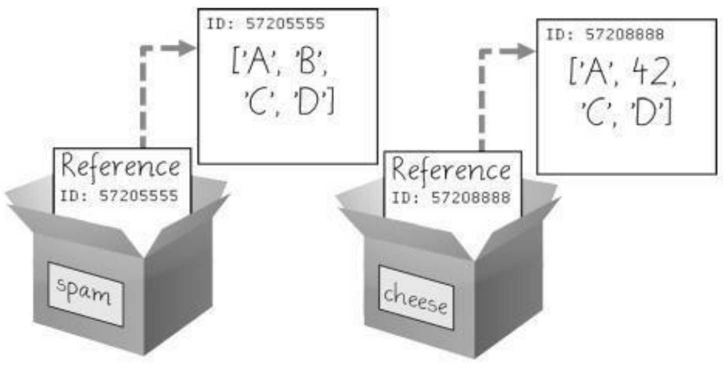


Figure: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

- 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.

MODULE 2
CHAPTER2: DICTIONARIES AND STRUCTURING DATA

1. The Dictionary Data Type
2. Pretty Printing
3. Using Data Structures to Model Real-World Things.

The Dictionary Data Type

- A dictionary is a collection of many values. 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.
- A dictionary is typed with braces, {}.

```
>>> 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, 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.

```
>>> 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
```

- 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.

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

- We can have arbitrary values for the keys that allows us to organize our data in powerful ways.
- **Ex:** we want to store data about our friends’ birthdays. We can use a dictionary with the names as keys and the birthdays as values.

```
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:
❸         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()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

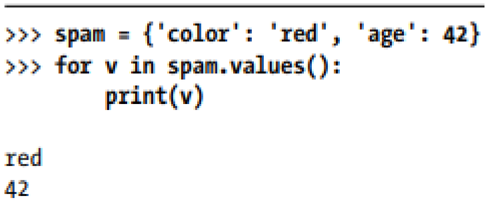
```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Program **Output**

- We create an initial dictionary and store it in birthdays **1**.
- We can see if the entered name exists as a key in the dictionary with the in keyword **2**.
- If the name is in the dictionary, we access the associated value using square brackets **3**; if not, we can add it using the same square bracket syntax combined with the assignment operator **4**.

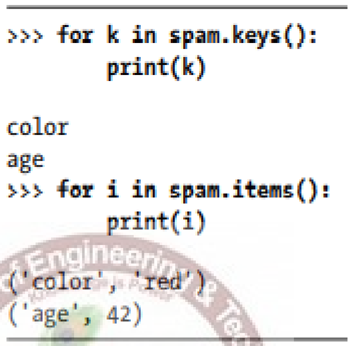
The keys(), values(), and items() Methods

- 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().
- Data types (dict_keys, dict_values, and dict_items, respectively) can be used in for loops.



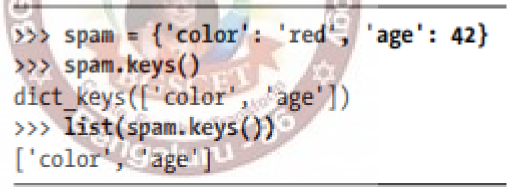
```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

- A for loop can iterate over the keys, values, or key-value pairs in a dictionary by using keys(), values(), and items() methods.
- The values in the dict_items value returned by the items() method are tuples of the key and value.



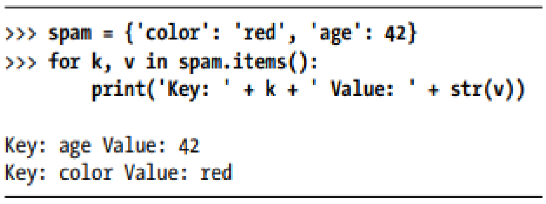
```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

- If we want a true list from one of these methods, pass its list-like return value to the list() function.



```
>>> 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'].
- We can also use the multiple assignment trick in a for loop to assign the key and value to separate variables.



```
>>> 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

➢ We can use the **in** and **not in** operators to see whether a certain key or value exists in a dictionary.

```
>>> 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
```

The get() Method

➢ 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.

```
>>> 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.'
```

The setdefault() Method

➢ To set a value in a dictionary for a certain key only if that key does not already have a value.

```
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.

➢ Setdeafault() takes 2 arguments:

- The first argument 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.

```
>>> 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'.

- **Ex:** program that counts the number of occurrences of each letter in a string.

```
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

print(count)
```

- The program loops over each character in the message variable's string, counting how often each character appears.
- The setdefault() method call 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.

Output:

```
{' ': 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}
```

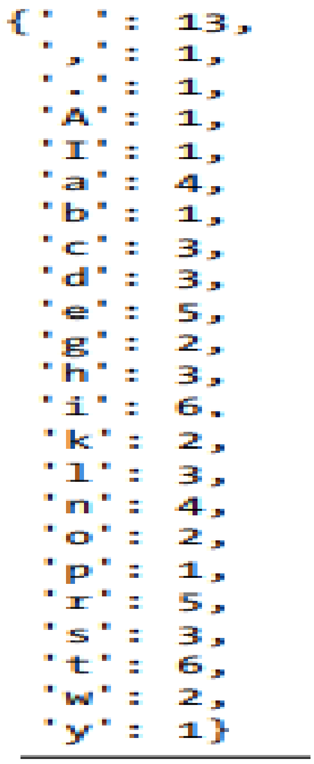
Pretty Printing

- Importing pprint module will provide access to the pprint() and pformat() functions that will "pretty print" a dictionary's values.
- This is helpful when we want a cleaner display of the items in a dictionary than what print() provides and also it is helpful when the dictionary itself contains nested lists or dictionaries..

Program: counts the number of occurrences of each letter in a string.

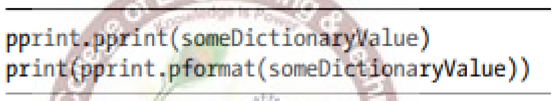
```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Output:



```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

- If we want to obtain the prettified text as a string value instead of displaying it on the screen, call pprint.pformat().



```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

Using Data Structures to Model Real-World Things

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, we can assign each slot a string-value key as shown in below figure.

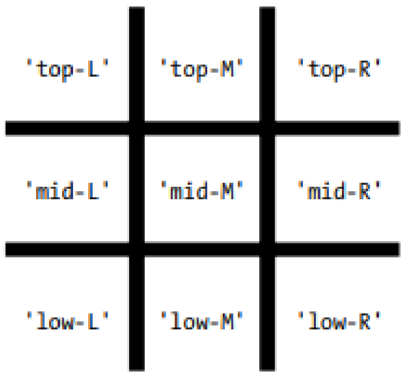


Figure: The slots of a tic-tactoe board with their corresponding keys

- We can use string values to represent what’s in each slot on the board: 'X', 'O', or ' ' (a space character).
- To store nine strings. We 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.
- Store this board-as-a-dictionary in a variable named theBoard.

```
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-tactoe board in the below Figure.

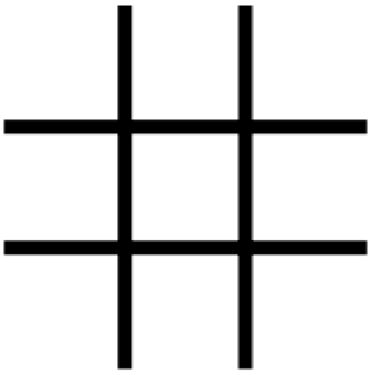


Figure: An empty tic-tac-toe board

- Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary as shown below:

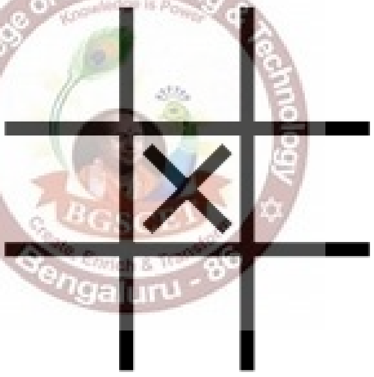


Figure: A first move

- 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 the below Figure.

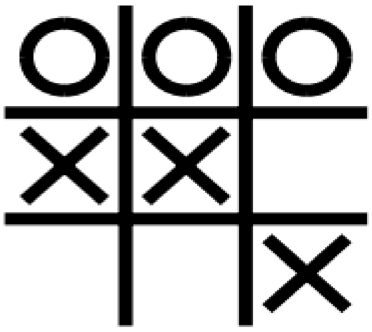


Figure: Player O wins.

➢ The player sees only what is printed to the screen, not the contents of variables.

➢ The tic-tac-toe program is updated as below.

```
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)
```

Output:

```
 | |
-+-+-
 | |
-+-+-
 | |
```

➢ The printBoard() function can handle any tic-tac-toe data structure you pass it.

Program

```
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)
```

Output:

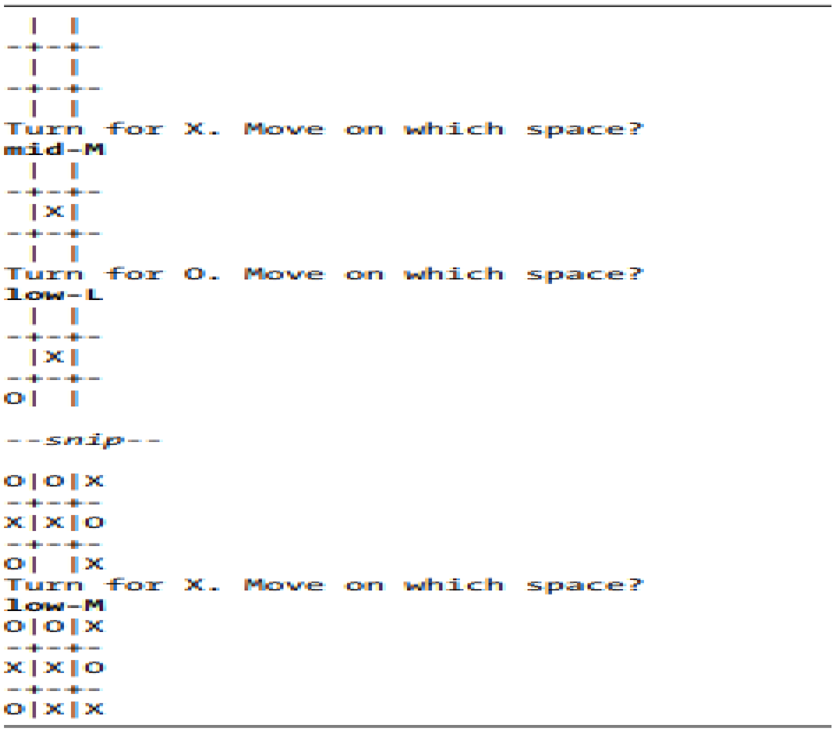
```
O|O|O
-+-+-
X|X|
-+-+-
 | |X
```

➢ Now we created a data structure to represent a tic-tac-toe board and wrote code in printBoard() to interpret that data structure, we now have a program that “models” the tic-tac-toe board.

Program: allows the players to enter their moves.

```
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)
```

Output:



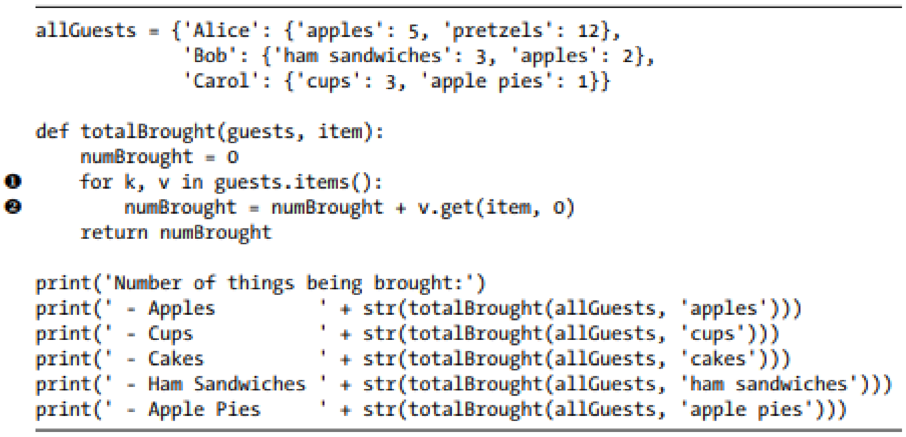
```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L
 | |
-+-+-
 |X|
-+-+-
O| |

--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
```

- The new code prints out the board at the start of each new turn **1**, gets the active player's move **2**, updates the game board accordingly **3**, and then swaps the active player **4** before moving on to the next turn.

Nested Dictionaries and Lists

- We can have program that contains dictionaries and lists which in turn contain other dictionaries and lists.
- Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.

Program: which contains nested dictionaries in order to see who is bringing what to a picnic.

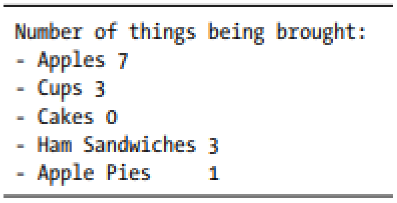


```
  allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
               'Bob': {'ham sandwiches': 3, 'apples': 2},
               'Carol': {'cups': 3, 'apple pies': 1}}

  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')))
```

- Inside the totalBrought() function, the for loop iterates over the keyvalue pairs in guests **1.**
- 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, it's value (the quantity) is added to numBrought **2**.

- If it does not exist as a key, the get() method returns 0 to be added to numBrought.

Output:



```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```

Module 3

CHAPTER-1: MANIPULATING STRINGS

1. Working with Strings
2. Useful String Methods
3. Project: Password Locker
4. Project: Adding Bullets to Wiki Markup

Working with strings

String Literals

- ➢ String values begin and end with a single quote.
- ➢ But we want to use either double or single quotes within a string then we have a multiple ways to do it as shown below.

Double Quotes

- ➢ One benefit of using double quotes is that the string can have a single quote character in it.

```
>>> 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.

Escape Characters

- ➢ If you need to use both single quotes and double quotes in the string, you'll need to use escape characters.
- ➢ An escape character consists of a backslash (\) followed by the character you want to add to the string.

```
>>> spam = 'Say hi to Bob\'s mother.'
```

- ➢ Python knows that the single quote in Bob\'s has a backslash, it is not a single quote meant to end the string value. The escape characters \' and \" allows to put single quotes and double quotes inside your strings, respectively.

Ex:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

- ➢ The different special escape characters can be used in a program as listed below in a table.

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

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

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Multiline Strings with Triple Quotes

- 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.

Program

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

Output

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

- 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
burglary, and extortion.\n\nSincerely,\nBob')
```

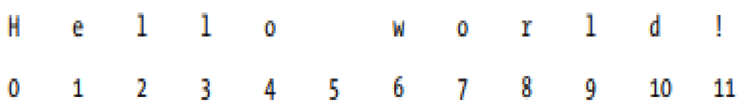
Multiline Comments

- 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.

```
"""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!')
```

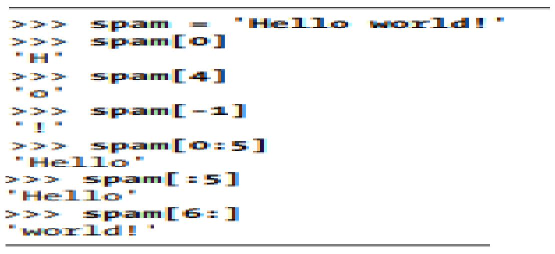
Indexing and Slicing Strings

- Strings use indexes and slices the same way lists do. We can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.



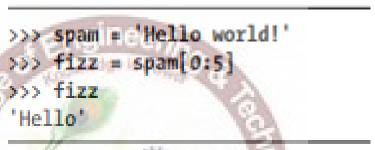
```
H   e   l   l   o       w   o   r   l   d   !
0   1   2   3   4   5   6   7   8   9   10  11
```

- The space and exclamation point are included in the character count, so 'Hello world!' is 12 characters long.
- If we specify an index, you'll get the character at that position in the string.



```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

- If we specify a range from one index to another, the starting index is included and the ending index is not.



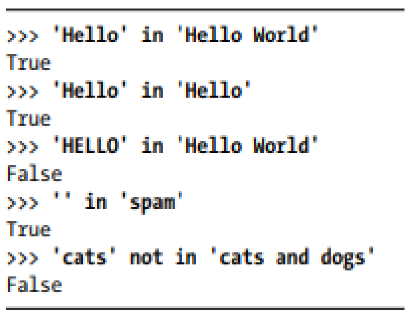
```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

- The substring we get from spam[0:5] will include everything from spam[0] to spam[4], leaving out the space at index 5.

Note: slicing a string does not modify the original string.

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.



```
>>> '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.

Useful String Methods

- Several string methods analyze strings or create transformed string values.

The upper(), lower(), isupper(), and islower() String 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.

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

- These methods do not change the string itself but return new string values.
- If we want to change the original string, we have to call upper() or lower() on the string and then assign the new string to the variable where the original was stored.
- The upper() and lower() methods are helpful if we need to make a case-insensitive comparison.
- 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.')
```

Program

```
How are you?
GREat
I feel great too.
```

Output

- 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.

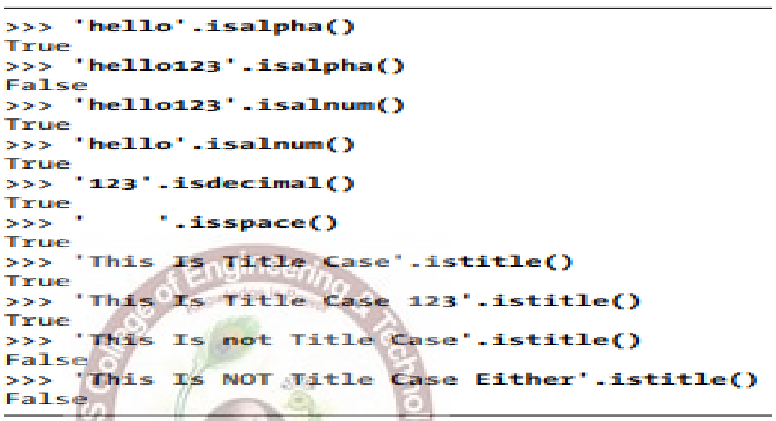
```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

- 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.

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

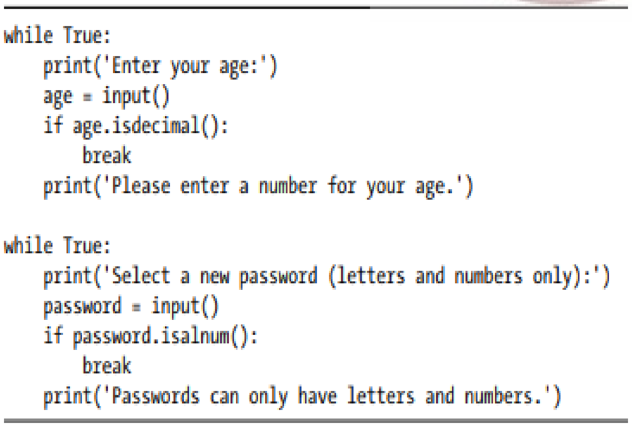
The isX String Methods

- There are several 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 is not 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.



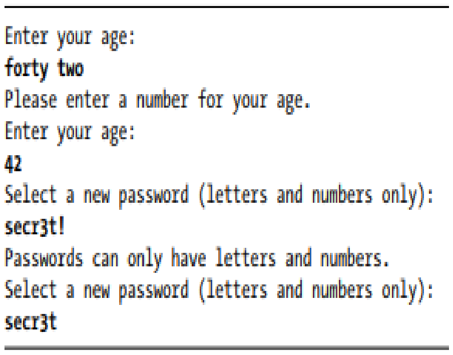
```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '     '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> '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.



```
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.')
```

Program

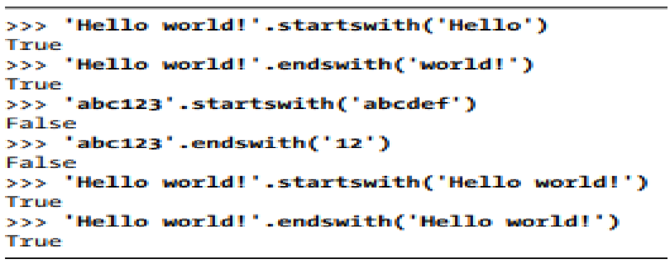


```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

output

The startswith() and endswith() String 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.



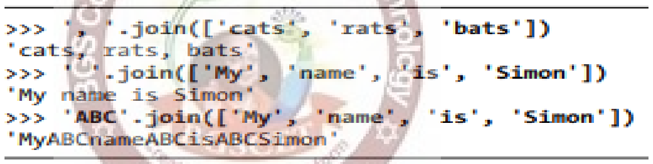
```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

- These methods are useful alternatives to the == equals operator if we 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() String Methods

Join()

- The join() method is useful when we 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.

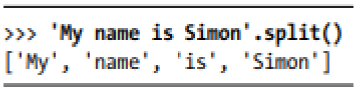


```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

- string join() calls on is inserted between each string of the list argument.
  - **Ex:** when join(['cats', 'rats', 'bats']) is called on the ', ' string, the returned string is 'cats, rats, bats'.
  - join() is called on a string value and is passed a list value.

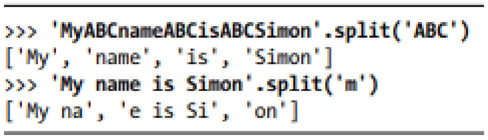
Split()

- The split() method is called on a string value and returns a list of strings.



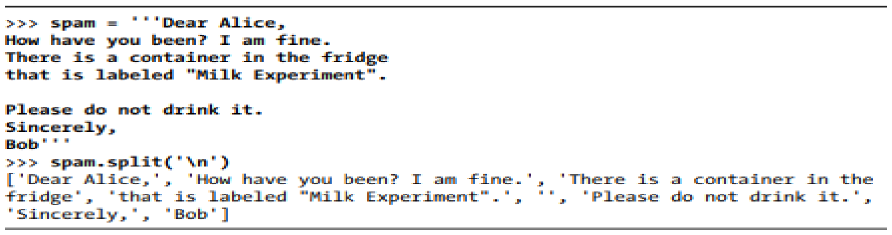
```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

- We can pass a delimiter string to the split() method to specify a different string to split upon.



```
>>> '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.

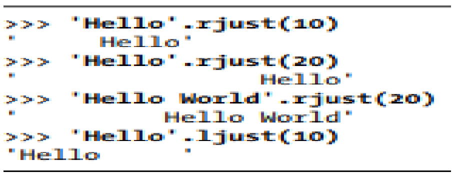


```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

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.

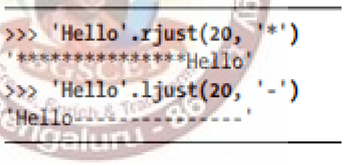
Justifying Text with rjust(), ljust(), and center()

- 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.



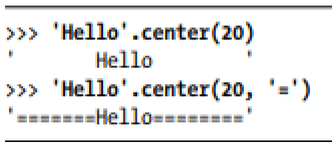
```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

- '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.



```
>>> '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.



```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

- These methods are especially useful when you need to print tabular data that has the correct spacing.
- In the below 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 8000 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.

```
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)
```

```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches..........     4
apples..............    12
cups................     4
cookies.............  8000
```

Program **output**

Removing Whitespace with strip(), rstrip(), and lstrip()

- 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.

```
>>> 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.

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

- Passing strip() the argument 'ampS' will tell it to strip occurences 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').

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.

```
>>> 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.

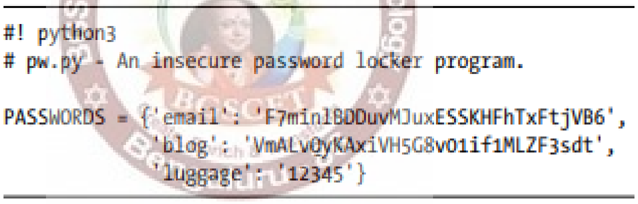
```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

Project: Password Locker

- ➢ We probably have accounts on many different websites.
- ➢ It’s a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts.
- ➢ It’s best to use password manager software on your computer that uses one master password to unlock the password manager.
- ➢ Then you can copy any account password to the clipboard and paste it into the website’s Password field
- ➢ The password manager program you’ll create in this example isn’t secure, but it offers a basic demonstration of how such programs work.

Step 1: Program Design and Data Structures

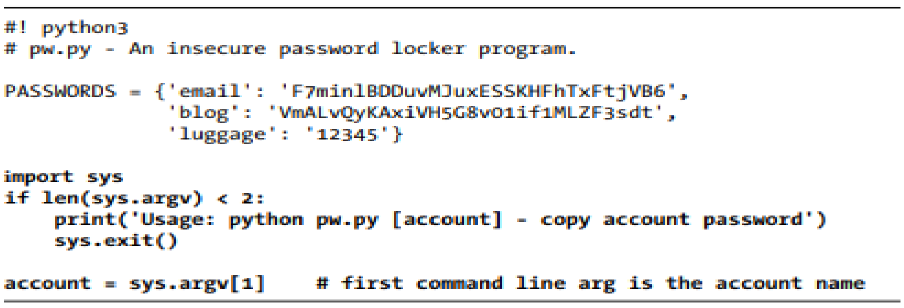
- ➢ We have to run this program with a command line argument that is the account’s name--for instance, email or blog. That account’s password will be copied to the clipboard so that the user can paste it into a Password field. The user can have long, complicated passwords without having to memorize them.
- ➢ We need to start the program with a #! (shebang) line and should also write a comment that briefly describes the program. Since we want to associate each account’s name with its password, we can store these as strings in a dictionary.



```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

- ➢ The command line arguments will be stored in the variable sys.argv.
- ➢ The **first** item in the sys.argv list should always be a string containing the program’s filename ('pw.py'), and the **second** item should be the first command line argument.



```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name
```

Step 3: Copy the Right Password

- ➢ The account name is stored as a string in the variable account, you need to see whether it exists in the PASSWORDS dictionary as a key. If so, you want to copy the key’s value to the clipboard using pyperclip.copy().

```
#! python3
# pw.py - An insecure password locker program.
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

- This new code looks in the PASSWORDS dictionary for the account name. If the account name 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 account with that name.
- On Windows, you can create a batch file to run this program with the win-R Run window. Type the following into the file editor and save the file as pw.bat in the C:\Windows folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

- With this batch file created, running the password-safe program on Windows is just a matter of pressing win-R and typing pw <account name>.

Project: Adding Bullets to Wiki Markup

- When editing a Wikipedia article, we can create a bulleted list by putting each list item on its own line and placing a star in front.
- But say we have a really large list that we want to add bullet points to. We could just type those stars at the beginning of each line, one by one. Or we 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.
- **Ex:**

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

Program

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

output

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

- Steps 1 and 3 are pretty straightforward and involve the pyperclip.copy() and pyperclip.paste() functions. saving the following program as bulletPointAdder.py:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the star
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

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.
- The \n newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard.
- We 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.

```
#! 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. 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.
- pyperclip.copy() is expecting a single string value, 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.

```
#! 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.

MODULE 3

CHAPTER 2-READING AND WRITING FILES

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.

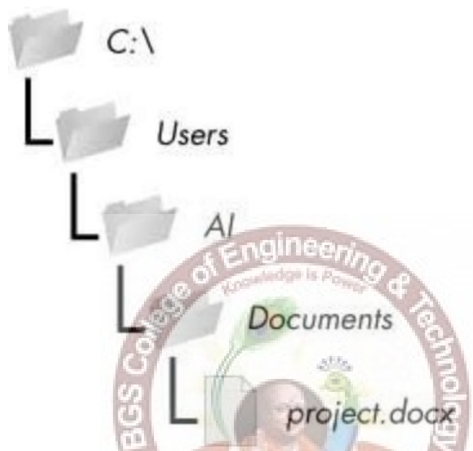


Figure 9-1: A file in a hierarchy of folders

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\Al', filename))
C:\Users\Al\accounts.txt
C:\Users\Al\details.csv
C:\Users\Al\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\Al'
>>> subFolder = 'spam'
>>> homeFolder + '\\' + subFolder
'C:\\Users\\Al\\spam'
>>> '\\'.join([homeFolder, subFolder])
'C:\\Users\\Al\\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.

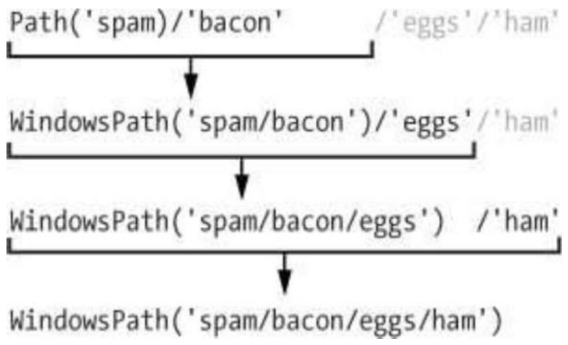
The pathlib module solves these problems by reusing the / math division operator to join paths correctly, no matter what operating system your code is running on. The following example uses this strategy to join the same paths as in the previous example:

```
>>> homeFolder = Path('C:/Users/Al')
>>> subFolder = Path('spam')
>>> homeFolder / subFolder
WindowsPath('C:/Users/Al/spam')
>>> str(homeFolder / subFolder)
'C:\\Users\\Al\\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.


Path('spam)/'bacon'
/'eggs'/'ham'
WindowsPath('spam/bacon')/'eggs'
/'ham'
WindowsPath('spam/bacon/eggs')
/'ham'
WindowsPath('spam/bacon/eggs/ham')

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.

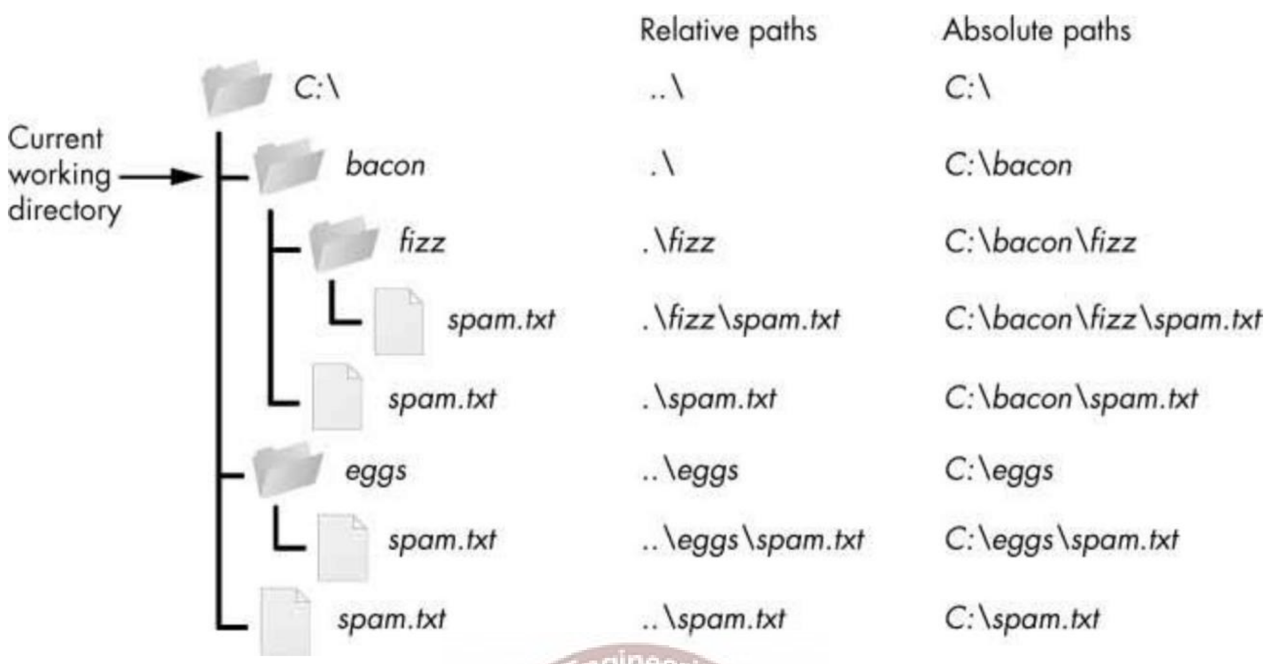


Figure 9-2: The relative paths for folders and files in the working directory C:\bacon

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.

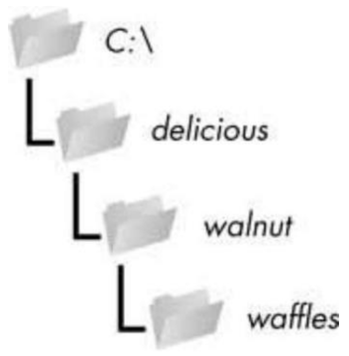


Figure 9-3: The result of os.makedirs('C:\\delicious\\walnut\\waffles')

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\Al\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/Al/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/Al/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.

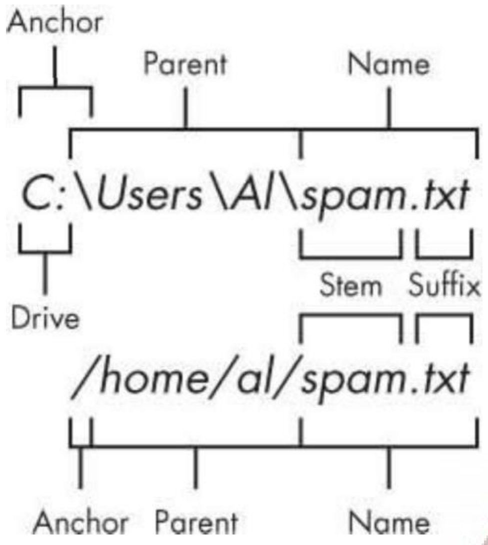


Figure 9-4: The parts of a Windows (top) and macOS/Linux (bottom) file path

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.

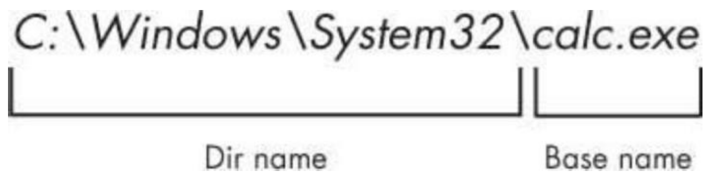


Figure 9-5: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

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.

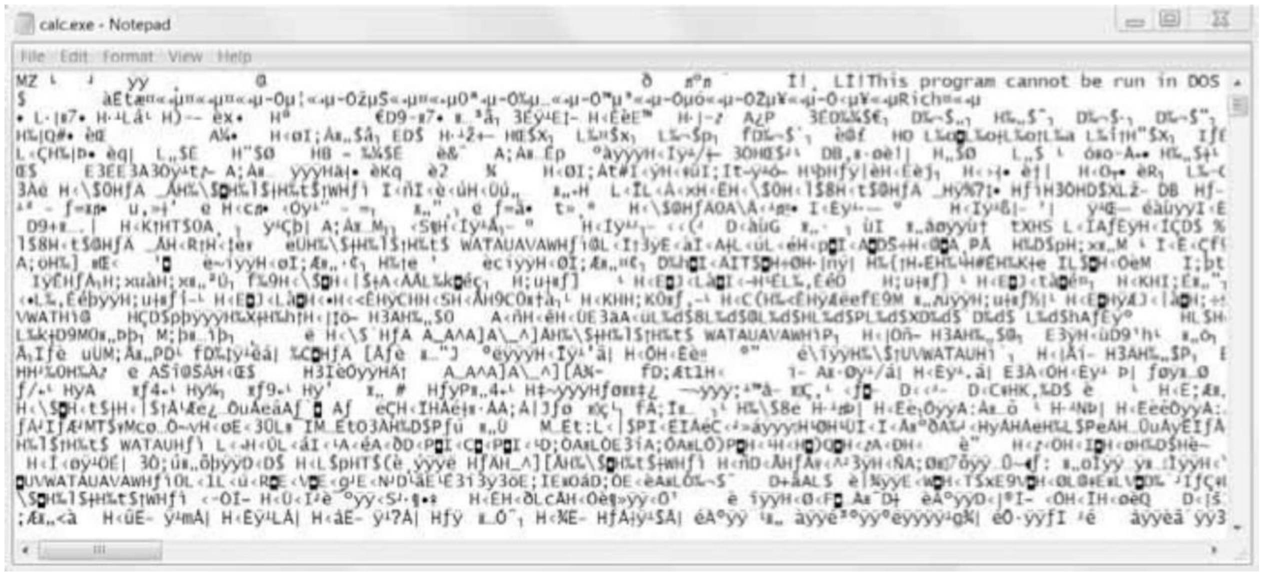


Figure 9-6: The Windows calc.exe program opened in Notepad

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 beweep 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 beweep 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')
  ❷answerKeyFile = open(f'capitalsquiz_answers{quizNum + 1}.txt', 'w')
    # Write out the header for the quiz.
  ❸quizFile.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())
  ❹random.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 ❶and ❷, with 'w' as the second argument to open them in write mode.

The write() statements at ❸create 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)
```

```
❺ answerOptions = wrongAnswers + [correctAnswer]
❻ 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 dictionary❶. 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 ❷ deleting the correct answer ❸ and selecting three random values from this list ❹The 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 .❺Finally, the answers need to be randomized ❻ 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')
    ❶ for i in range(4):
       ❷ 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 line❸, 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 modules❷ 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') / 'Al' evaluate to on Windows?
4. What does 'C:/Users' / 'Al' 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.
```

MODULE 4

CHAPTER 1- ORGANIZING FILES

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()
➊ >>> shutil.copy(p / 'spam.txt', p / 'some_folder')
  'C:\\Users\\Al\\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 destination❶, the original *spam.txt* filename is used for the new, copied file's filename. The second shutil.copy() call❷ 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.

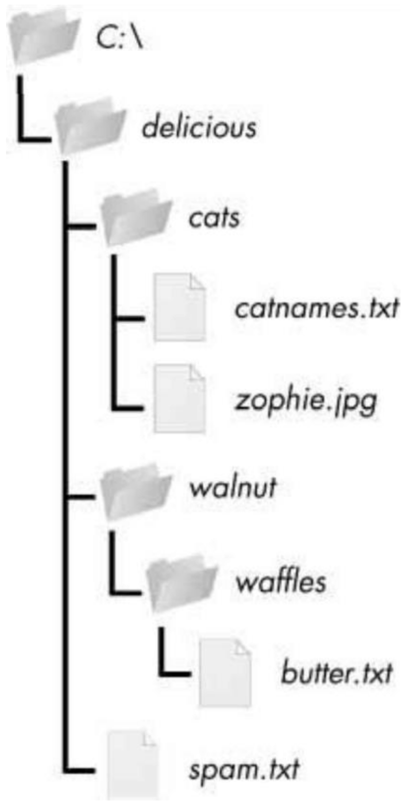


Figure 10-1: An example folder that contains three folders and four files

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.

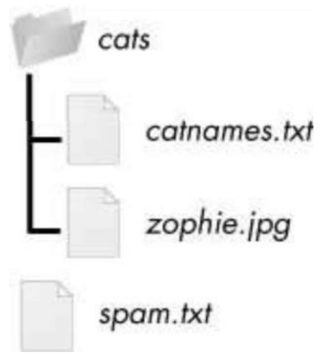


Figure 10-2: The contents of example.zip

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 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 handLet'swrite 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 None❶, 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 afterPart❸ 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 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 ❹.

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* ❷, and keep incrementing it inside the loop that calls os.path.exists() to check whether the file exists ❸. 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.
     ❷backupZip.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, itâs 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.

MODULE 4
CHAPTER 2-DEBUGGING

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 w ill 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 statement❹. 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:

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
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 AssertionError❶. 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 × 2 × 3 × 4, or 24. Factorial 7 is 1 × 2 × 3 × 4 × 5 × 6 × 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)
```

```
2019-05-23 17:13:40,666 - DEBUG - End of program
```

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	logging.debug()	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:

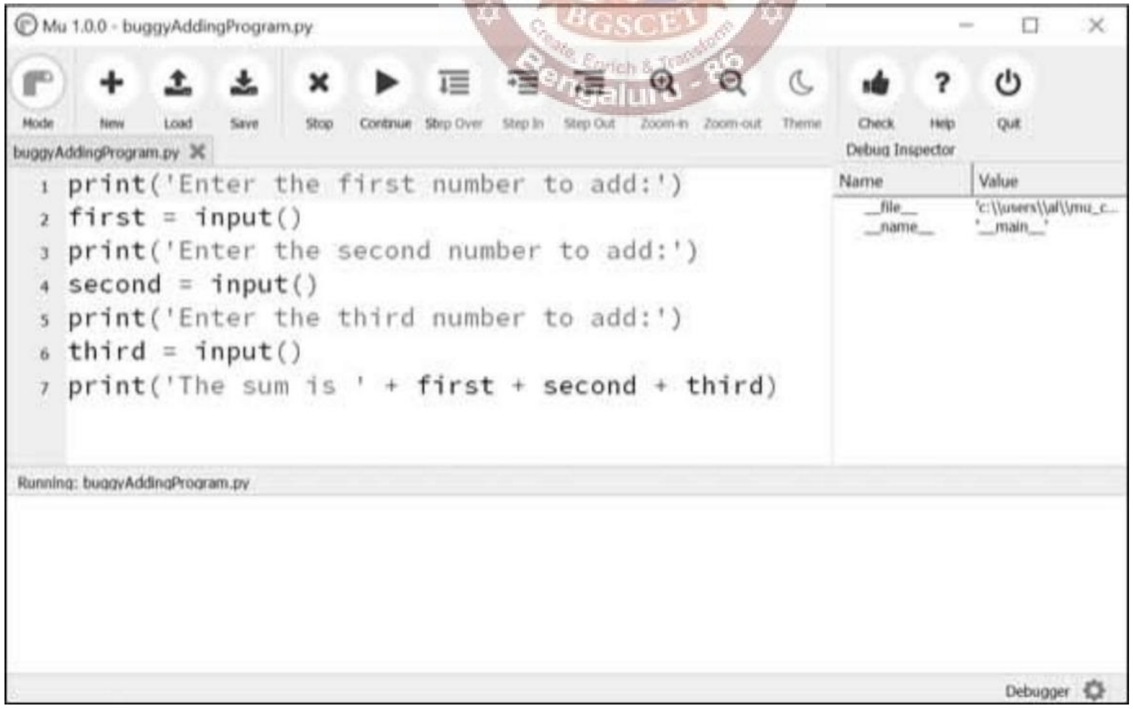
```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='
%(asctime)s - %(levelname)s - %(message)s')
```

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.



```
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)
```

Name	Value
__file__	'c:\\users\\al\\mu_c...
__name__	'__main__'

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)
```

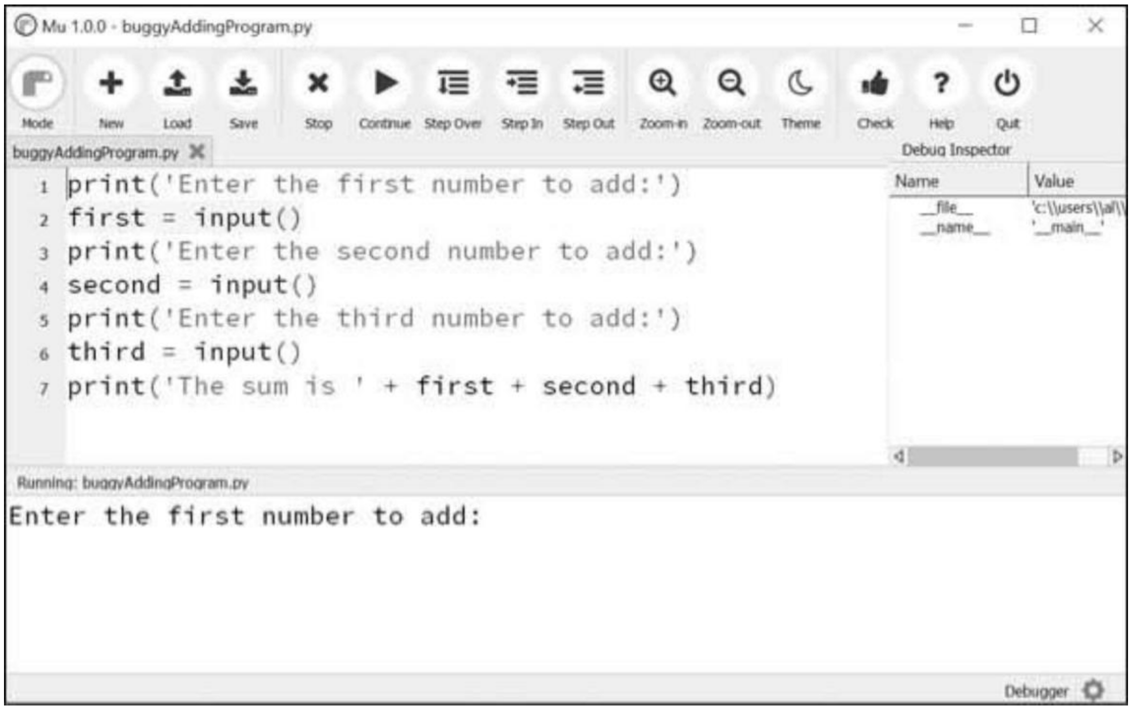
Save it as *buggyAddingProgram.py* and run it first without the debugger enabled. The program will output something like this:

```
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.



```
1 print('Enter the first number to add:')
2 first = input()
3 print('Enter the second number to add:')
4 second = input()
5 print('Enter the third number to add:')
6 third = input()
7 print('The sum is ' + first + second + third)
```

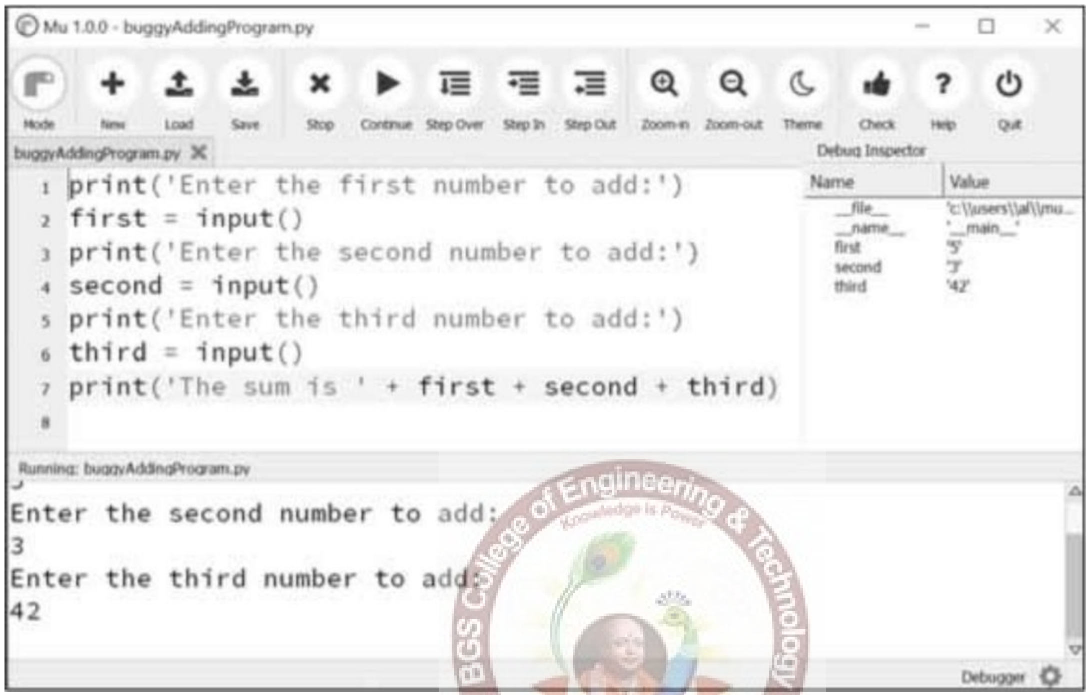
Name	Value
__file__	'c:\\users\\al\\
__name__	'__main__'

```
Enter the first number to add:
```

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.



```
1 print('Enter the first number to add:')
2 first = input()
3 print('Enter the second number to add:')
4 second = input()
5 print('Enter the third number to add:')
6 third = input()
7 print('The sum is ' + first + second + third)
8
```

Name	Value
__file__	'c:\\users\\al\\mu...
__name__	'__main__'
first	'5'
second	'3'
third	'42'

```
Enter the second number to add:
3
Enter the third number to add:
42
```

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*.

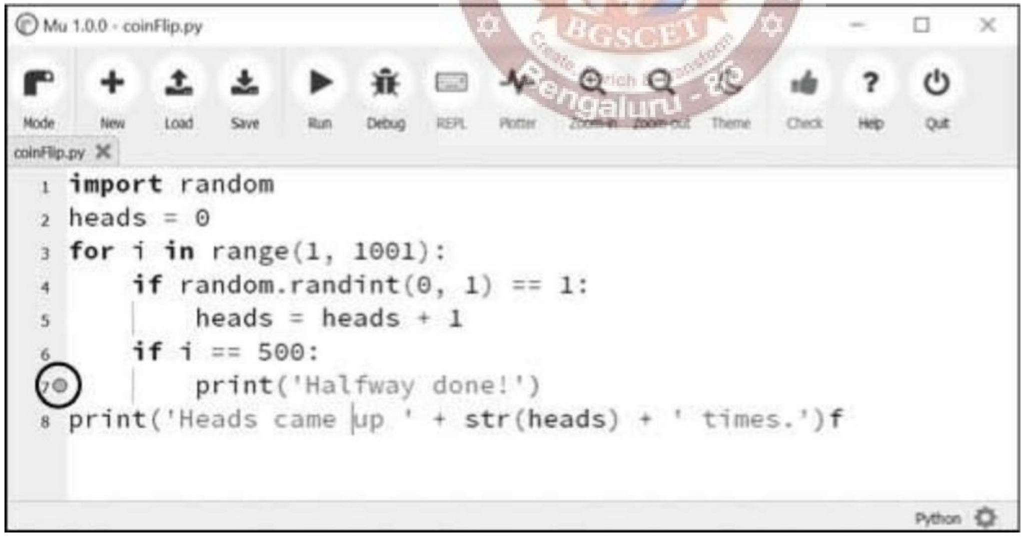
```
import random
heads = 0
```

```
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.



```
import random
heads = 0
for i in range(1, 1001):
    if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
        print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')f
```

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!')
    guesss = input()
    if toss == guess:
        print('You got it!')
    else:
        print('Nope. You are really bad at this game.')
```

MODULE 5

CHAPTER 01

CLASSES AND OBJECTS

1. Programmer-defined types

- We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two-dimensional space.
- In mathematical notation, points are often written in parentheses with a comma separating the coordinates.
- For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.
- There are several ways we might represent points in Python:
  1. We could store the coordinates separately in two variables, x and y.
  2. We could store the coordinates as elements in a list or tuple.
  3. We could create a new type to represent points as objects.
- Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.
- A programmer-defined type is also called a class. A class definition looks like this:

```
class Point:
"""Represents a point in 2-D space."""
```

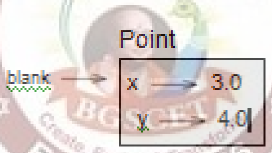


Figure 15.1: Object diagram.

- The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.
- Defining a class named Point creates a class object.

```
>>> Point
            <class '__main__.Point'>
```

- Because Point is defined at the top level, its "full name" is __main__.Point.
- The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

```
blank = Point()
blank
<__main__.Point object at 0xb7e9d3ac>
```

- The return value is a reference to a Point object, which we assign to blank.
- Creating a new object is called instantiation, and the object is an instance of the class.
- When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

2. **Attributes**

- You can assign values to an instance using dot notation:

```
blank.x = 3.0
blank.y = 4.0
```

- This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace .
- In this case, though, we are assigning values to named elements of an object. These elements are called attributes.
- A state diagram that shows an object and its attributes is called an object diagram; see Figure 15.1.
- The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.
- You can read the value of an attribute using the same syntax:

blank.y 4.0	x = blank.x x 3.0

- The expression blank.x means, "Go to the object blank refers to and get the value of x." In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.
- You can use dot notation as part of any expression. For example:

```
'(%g, %g)' % (blank.x, blank.y)
 '(3.0, 4.0)'
```

```
distance = math.sqrt(blank.x**2 + blank.y**2)
distance
5.0
```

- You can pass an instance as an argument in the usual way. For example:

```
def   print_point(p):
print('(%g, %g)' % (p.x, p.y))
```

- print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
print_point(blank) (3.0, 4.0)
```

- Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

3. **Rectangles**

- Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions.

- For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle?
- You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.
- There are at least two possibilities:
    1. You could specify one corner of the rectangle (or the center), the width, and the height.
    2. You could specify two opposing corners.
- At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.
- Here is the class definition:

```
class Rectangle:
"""Represents a rectangle.
attributes: width, height, corner."""
```

- The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.
- To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

- The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."
- Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

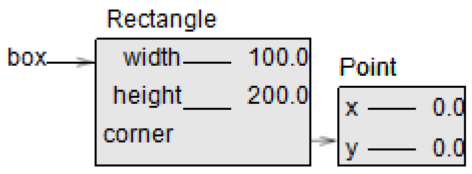


Figure 15.2: Object diagram.

4. Instances as return values

- Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y + rect.height/2
return p
```

➢ Here is an example that passes box as an argument and assigns the resulting Point to center:

```
center = find_center(box)
print_point(center)
(50, 100)
```

5. Objects are mutable

➢ You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

➢ You can also write functions that modify objects.

➢ For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
```

➢ Here is an example that demonstrates the effect:

```
box.width, box.height (150.0, 300.0)
grow_rectangle(box, 50, 100)
box.width, box.height
(200.0, 400.0)
```

➢ Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

6. Copying

➢ Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place.

➢ It is hard to keep track of all the variables that might refer to a given object.

➢ Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0

import copy
p2 = copy.copy(p1)
```

➢ p1 and p2 contain the same data, but they are not the same Point.

```
print_point(p1)
(3, 4)
print_point(p2)
(3, 4)
p1 is p2
False
p1 == p2
False
```

- The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data.
- In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.
- That’s because for programmer-defined types, Python doesn’t know what should be considered equivalent. At least, not yet.

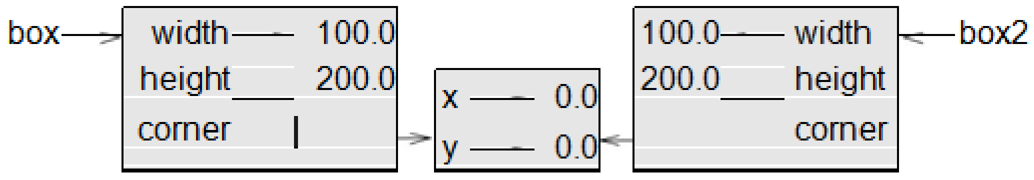


Figure 15.3: Object diagram.

- If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
box2 = copy.copy(box)
box2 is box
False
box2.corner is box.corner
True
```

- Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.
- For most applications, this is not what you want.
- In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behavior is confusing and error-prone.
- Fortunately, the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy.**

```
box3 = copy.deepcopy(box)
box3 is box
False
box3.corner is box.corner
False
```

- box3 and box are completely separate objects.

CHAPTER 02
CLASSES AND FUNCTIONS

1. Time

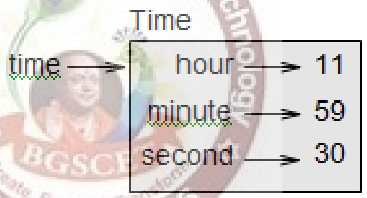
- As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second """
```

- We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time = Time( )
time.hour = 11
time.minute    =  59
time.second    =  30
```

- The state diagram for the Time object looks like Figure below.



2. Pure functions

- In the next few sections, we'll write two functions that add time values.
- They demonstrate two kinds of functions: pure functions and modifiers.
- They also demonstrate a development plan I'll call prototype and patch, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.
- Here is a simple prototype of add_time:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

- The function creates a new Time object, initializes its attributes, and returns a reference to the new object.
- This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

- To test this function, let us create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes.
- add_time figures out when the movie will be done.

```
>>>  start  = Time()
>>>  start.hour  = 9
>>>   start.minute  = 45
>>>  start.second  =    0

>>>   duration   = Time()
>>>   duration.hour   = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

- The result, 10:80:00 might not be what you were hoping for.
- The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty.
- When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.
- Here's an improved version:

```
def add_time(t1, t2):
sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

if sum.second >= 60: sum.second -= 60
    sum.minute += 1

if sum.minute >= 60: sum.minute -= 60
    sum.hour += 1

    return sum
```

3. Modifiers

- Sometimes it is useful for a function to modify the objects it gets as parameters.
- In that case, the changes are visible to the caller. Functions that work this way are called modifiers.
- increment, which adds a given number of seconds to a Time object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
time.second += seconds

if time.second >= 60: time.second -= 60
time.minute += 1

if time.minute >= 60: time.minute -= 60
time.hour += 1
```

- The first line performs the basic operation; the remainder deals with the special cases we saw before.
- Is this function correct? What happens if seconds is much greater than sixty?
- In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty.
- One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient.
- Anything that can be done with modifiers can also be done with pure functions.

4. Prototyping versus planning

- The development plan, i.e. demonstrating is called “prototype and patch”. For each function, we wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.
- This approach can be effective, especially if you don’t yet have a deep understanding of the problem.
- But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.
- Here is a function that converts Times to integers:

```
def time_to_int(time):
minutes = time.hour * 60 +time.minute
seconds = minutes * 60 + time.second return seconds
```

- And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
 time = Time()
minutes, time.second = divmod(seconds, 60)
time.hour, time.minute = divmod(minutes, 60)
return time
```

- Once we are convinced they are correct, you can use them to rewrite:

```
def add_time(t1, t2):
seconds = time_to_int(t1) + time_to_int(t2)
return int_to_time(seconds)
```

- This version is shorter than the original, and easier to verify.

MODULE5

CHAPTER 03
CLASSES AND METHODS

1. Object-Oriented Features

- Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:

  - Programs include class and method definitions.
  - Most of the computation is expressed in terms of operations on objects.
  - Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

- A method is a function that is associated with a particular class.
- Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

2. Printing Objects

- We already defined a class named and also wrote a function named print_time:

```
class Time:
"""Represents the time of day."""

def print_time(time):
print('%.2d:%.2d:%.2d'      %  (time.hour,  time.minute,  time.second))
```

- To call this function, we have to pass a Time object as an argument:

```
>>>  start  = Time()
>>>  start.hour  = 9
>>>   start.minute  = 45
>>>   start.second  = 00
>>> print_time(start)
09:45:00
```

- To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d'   %  (time.hour,  time.minute,  time.second))
```

- Now there are two ways to call print_time. The first (and less common) way is to use function syntax:

```
>>>Time.print_time(start)

09:45:00
```

- In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.
- The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

- In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the subject.
- Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.
- Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.
- By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d'   %  (self.hour,  self.minute,  self.second))
```

- The reason for this convention is an implicit metaphor:

  - The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, “Hey print_time! Here’s an object for you to print.”
  - In object-oriented programming, the objects are the active agents. A method invocation like

start.print_time() says “Hey start! Please print yourself.”

3. Another Example

- Here’s a version of increment rewritten as a method:

```
# inside class Time:

def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

- This version assumes that time_to_int is written as a method. Also, note that it is a pure function, not a modifier.
- Here’s how you would invoke increment:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

- The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.
- This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

- The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that’s three.
- By the way, a positional argument is an argument that doesn’t have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

- parrot and cage are positional, and dead is a keyword argument.

4. A More Complicated Example

- Rewriting is_after is slightly more complicated because it takes two Time objects as parameters.
- In this case it is conventional to name the first parameter self and the second parameter other:

```
# inside class Time:

def is_after(self, other):
return self.time_to_int() > other.time_to_int()
```

- To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)

True
```

5. The init Method

- The init method (short for "initialization") is a special method that gets invoked when an object is instantiated.
- Its full name is __init__(two underscore characters, followed by init, and then two more underscores).
- An init method for the Time class might look like this:

```
# inside class Time:
def _init_(self, hour=0, minute=0, second=0):
    self.hour = hour
    self.minute = minute
    self.second = second
```

- It is common for the parameters of___init___to have the same names as the attributes.
- The statement

```
self.hour = hour
```

- stores the value of the parameter hour as an attribute of self.
- The parameters are optional, so if you call Time with no arguments, you get the default values:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

- If we provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

- If we provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

- And if we provide three arguments, they override all three default values

6. The str Method

- __str__is a special method, like __init_, that is supposed to return a string representa- tion of an object.
- For example, here is a str method for Time objects:

```
# inside class Time:
def__str_(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

- When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

7. Operator Overloading

- By defining other special methods, you can specify the behavior of operators on programmer-defined types.
- For example, if we define a method named __add___for the Time class, you can use the + operator on Time objects.
- Here is what the definition might look like:

```
def _add_(self,other):
    seconds=self.time_to_int()+other.time_to_int()
    return int_to_time(seconds)
```

- And here is how we could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

- When you apply the + operator to Time objects, Python invokes __add_.
- When you print the result, Python invokes __str_. So there is a lot happening behind the scenes!
- Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading.
- For every operator in Python there is a corresponding special method, like __add_.

8. Type-Based Dispatch

- The following is the version of _add_ that checks the type of other and invokes either add_time or increment:

```
def_add_(self,other):
      if isintance(other, Time):
            return self.add_time(other)
     else:
           return self.increment(other)

def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
         return int_to_time(seconds)

def increment(self,  seconds):
            seconds += self.time_to_int()
            return int_to_time(seconds)
```

- The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class.
- If other is a Time object, __add___invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment.
- This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments.
- Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

- Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int'    and 'instance'
```

- The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how.
- But there is a clever solution for this problem: the special method __radd_, which stands for "right-side add".
- This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:
def __radd_(self, other):
    return self._add_(other)
```

- And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

9. **Polymorphism**

- Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.
- Many of the functions we wrote for strings also work for other sequence types. For example, we used histogram to count the number of times each letter appears in a word.

```
def
histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

- This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d:

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

- Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse.
- For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 31)
>>> t3 = Time(7, 37)
>>> total = sum(t1, t2, t3)
>>> print(total)
23:01:00
```

- In general, if all of the operations inside a function work with a given type, the function works with that type.
- The best kind of polymorphism is the unintentional kind, where you discover that a func- tion you already wrote can be applied to a type you never planned for.

10. Interface and implementation

- One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

- A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented. For example, in this chapter we developed a class that represents a time of day.

- Methods provided by this class include time_to_int, is_after, and add_time. We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a Time object are hour, minute, and second.

- As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like is_after, easier to write, but it makes some methods harder.

- After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface. But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don’t have to change.

- Keeping the interface separate from the implementation means that you have to hide the attributes. Code in other parts of the program (outside the class definition) should use methods to read and modify the state of the object. They should not access the attributes directly. This principle is called information hiding; see http://en.wikipedia.org/wiki/ Information_hiding.