Algorithmic Problem Solving with Python

John B. Schneider   Shira Lynn Broschat   Jess Dahmen

February 22, 2019

Contents

1 Introduction **1**
- 1.1 Modern Computers . . . 1
- 1.2 Computer Languages . . . 2
- 1.3 Python . . . 3
- 1.4 Algorithmic Problem Solving . . . 6
- 1.5 Obtaining Python . . . 7
- 1.6 Running Python . . . 8
  - 1.6.1 Interactive Sessions and Comments . . . 9
  - 1.6.2 Running Commands from a File . . . 11
- 1.7 Bugs . . . 12
- 1.8 The **`help()`** Function . . . 13
- 1.9 Comments on Learning New Languages . . . 14
- 1.10 Chapter Summary . . . 15
- 1.11 Review Questions . . . 15

2 Core Basics **19**
- 2.1 Literals and Types . . . 19
- 2.2 Expressions, Arithmetic Operators, and Precedence . . . 22
- 2.3 Statements and the Assignment Operator . . . 24
- 2.4 Cascaded and Simultaneous Assignment . . . 27
- 2.5 Multi-Line Statements and Multi-Line Strings . . . 29
- 2.6 Identifiers and Keywords . . . 30
- 2.7 Names and Namespaces . . . 32
- 2.8 Additional Arithmetic Operators . . . 37
  - 2.8.1 Exponentiation . . . 37
  - 2.8.2 Floor Division . . . 37
  - 2.8.3 Modulo and **`divmod()`** . . . 38
  - 2.8.4 Augmented Assignment . . . 40
- 2.9 Chapter Summary . . . 42
- 2.10 Review Questions . . . 43
- 2.11 Exercises . . . 49

3 Input and Type Conversion **51**
- 3.1 Obtaining Input: **`input()`** . . . 51
- 3.2 Explicit Type Conversion: **`int()`**, **`float()`**, and **`str()`** . . . 53

3.3 Evaluating Strings: **eval()** 57
3.4 Chapter Summary 59
3.5 Review Questions 59
3.6 Exercises 65

4 Functions **67**
4.1 Void Functions and **None** 68
4.2 Creating Void Functions 69
4.3 Non-Void Functions 72
4.4 Scope of Variables 76
4.5 Scope of Functions 78
4.6 **print()** vs. **return** 79
4.7 Using a **main()** Function 81
4.8 Optional Parameters 82
4.9 Chapter Summary 86
4.10 Review Questions 87
4.11 Exercises 92

5 Introduction to Objects **95**
5.1 Overview of Objects and Classes 95
5.2 Creating a Class: Attributes 97
5.3 Creating a Class: Methods 99
5.4 The **dir()** Function 101
5.5 The **__init__()** Method 103
5.6 Operator Overloading 105
5.7 Take-Away Message 106
5.8 Chapter Summary 107

6 Lists and for-Loops **109**
6.1 **list**s 110
6.2 **list** Methods 111
6.3 **for**-Loops 113
6.4 Indexing 115
6.5 **range()** 118
6.6 Mutability, Immutability, and Tuples 121
6.7 Nesting Loops in Functions 123
6.8 Simultaneous Assignment with Lists 126
6.9 Examples 127
6.9.1 Storing Entries in a list 127
6.9.2 Accumulators 129
6.9.3 Fibonacci Sequence 130
6.10 Chapter Summary 132
6.11 Review Questions 133

7 More on `for`-Loops, Lists, and Iterables **143**
- 7.1 `for`-Loops within `for`-Loops . . . 143
- 7.2 **`list`**s of **`list`**s . . . 148
  - 7.2.1 Indexing Embedded **`list`**s . . . 151
  - 7.2.2 Simultaneous Assignment and **`list`**s of **`list`**s . . . 154
- 7.3 References and **`list`** Mutability . . . 157
- 7.4 Strings as Iterables or Sequences . . . 162
- 7.5 Negative Indices . . . 164
- 7.6 Slicing . . . 165
- 7.7 **`list`** Comprehension (optional) . . . 168
- 7.8 Chapter Summary . . . 171
- 7.9 Review Questions . . . 172

8 Modules and `import` Statements **177**
- 8.1 Importing Entire Modules . . . 179
- 8.2 Introduction to Complex Numbers . . . 181
- 8.3 Complex Numbers and the **`cmath`** Module . . . 185
- 8.4 Importing Selected Parts of a Module . . . 187
- 8.5 Importing an Entire Module Using **`*`** . . . 189
- 8.6 Importing Your Own Module . . . 190
- 8.7 Chapter Summary . . . 193
- 8.8 Review Questions . . . 194

9 Strings **195**
- 9.1 String Basics . . . 196
- 9.2 The ASCII Characters . . . 199
- 9.3 Escape Sequences . . . 200
- 9.4 **`chr()`** and **`ord()`** . . . 203
- 9.5 Assorted String Methods . . . 209
  - 9.5.1 **`lower()`**, **`upper()`**, **`capitalize()`**, **`title()`**, and **`swapcase()`** . 209
  - 9.5.2 **`count()`** . . . 210
  - 9.5.3 **`strip()`**, **`lstrip()`**, and **`rstrip()`** . . . 210
  - 9.5.4 **`__repr__()`** . . . 211
  - 9.5.5 **`find()`** and **`index()`** . . . 212
  - 9.5.6 **`replace()`** . . . 215
- 9.6 **`split()`** and **`join()`** . . . 215
- 9.7 Format Strings and the **`format()`** Method . . . 218
  - 9.7.1 Replacement Fields as Placeholders . . . 219
  - 9.7.2 Format Specifier: Width . . . 222
  - 9.7.3 Format Specifier: Alignment . . . 223
  - 9.7.4 Format Specifier: Fill and Zero Padding . . . 224
  - 9.7.5 Format Specifier: Precision (Maximum Width) . . . 225
  - 9.7.6 Format Specifier: Type . . . 226
  - 9.7.7 Format Specifier: Summary . . . 227
  - 9.7.8 A Formatting Example . . . 228

9.8 Chapter Summary 230
9.9 Review Questions 231

10 Reading and Writing Files **239**
10.1 Reading a File 239
10.1.1 **read()**, **close()**, and **tell()** 241
10.1.2 **readline()** 243
10.1.3 **readlines()** 244
10.1.4 File Object Used as an Iterable 245
10.1.5 Using More than One Read Method 247
10.2 Writing to a File 247
10.2.1 **write()** and **print()** 248
10.2.2 **writelines()** 250
10.3 Chapter Summary 251
10.4 Review Questions 252

11 Conditional Statements **255**
11.1 **if** Statements, Boolean Variables, and **bool()** 255
11.2 Comparison Operators 261
11.3 Compound Conditional Statements 267
11.3.1 **if-else** Statements 267
11.3.2 **if-elif-else** Statements 270
11.4 Logical Operators 272
11.5 Multiple Comparisons 275
11.6 **while**-Loops 276
11.6.1 Infinite Loops and **break** 278
11.6.2 **continue** 281
11.7 Short-Circuit Behavior 283
11.8 The **in** Operator 287
11.9 Chapter Summary 289
11.10 Review Questions 290

12 Recursion **297**
12.1 Background 297
12.2 Flawed Recursion 297
12.3 Proper Recursion 299
12.4 Merge Sort 306

13 Turtle Graphics **311**
13.1 Introduction 311
13.2 Turtle Basics 311
13.2.1 Importing Turtle Graphics 312
13.2.2 Your First Drawing 313
13.3 Basic Shapes and Using Iteration to Generate Graphics 317
13.3.1 Controlling the Turtle's Animation Speed 319

13.4 Colors and Filled Shapes 320
13.4.1 Strange Errors 323
13.4.2 Filled Shapes 323
13.5 Visualizing Recursion 324
13.6 Simple GUI Walk-Through 330
13.6.1 Function References 330
13.6.2 Callback functions 332
13.6.3 A simple GUI 332

14 Dictionaries 335
14.1 Dictionary Basics 336
14.2 Cycling through a Dictionary 338
14.3 `get()` 341
14.4 Chapter Summary 343
14.5 Review Questions 344

A ASCII Non-printable Characters 347

Index 349

Chapter 1

Introduction

1.1 Modern Computers

At their core, computers are remarkably simple devices. Nearly all computers today are built using electronic devices called transistors. These transistors serve as switches that behave much like simple light switches—they can be on or they can be off. In a digital computer each *bit* of information (whether input, memory, or output) can be in only one of two states: either off or on, or we might call these states low or high, or perhaps zero or one. When we say "bit," we have in mind the technical definition. A bit is a ***b***inary dig***it*** that can be either 0 or 1 (zero or one). In a very real sense computers only "understand" these two numbers. However, by combining thousands or millions or even billions of these transistor switches we can achieve fantastically complicated behavior. Thus, rather than keeping track of a single binary digit, with computers we may be able to work with a stream of bits of arbitrary length.

For each additional bit we use to represent a quantity, we double the number of possible unique values the quantity can have. One bit can represent only two "states" or values: 0 and 1. This may seem extremely limiting, but a single bit is enough to represent whether the answer to a question is yes or no or a single bit can be used to tell us whether a logical statement evaluates to either true or false. We merely have to agree to interpret values consistently, for example, 0 represents no or false while 1 represents yes or true. Two bits can represent four states which we can write as: 00, 01, 10, and 11 (read this as zero-zero, zero-one, one-zero, one-one). Three bits have eight unique combinations or values: 000, 001, 010, 011, 100, 101, 110, and 111. In general, for n bits the number of unique values is 2^n.

For $n = 7$ bits, there are $2^7 = 128$ unique values. This is already more than the number of all the keys on a standard keyboard, i.e., more than all the letters in the English alphabet (both uppercase and lowercase), plus the digits (0 through 9), plus all the standard punctuation marks. So, by using a mapping (or *encoding*) of keyboard characters to unique combinations of binary digits, we can act as though we are working with characters when, really, we are doing nothing more than manipulating binary numbers.

We can also take values from the (real) continuous world and "digitize" them. Rather than having values such as the amplitude of a sound wave or the color of an object vary continuously, we restrict the amplitude or color to vary between fixed values or levels. This process is also known

From the file: `intro.tex`

as digitizing or quantizing. If the levels of quantization are "close enough," we can fool our senses into thinking the digitized quantity varies continuously as it does in the real world. Through the process of digitizing, we can store, manipulate, and render music or pictures on our computers when we are simply dealing with a collection of zeros and ones.

1.2 Computer Languages

Computers, though remarkably simple at their core, have, nevertheless, truly revolutionized the way we live. They have enabled countless advances in science, engineering, and medicine. They have affected the way we exchange information, how we socialize, how we work, and how we play. To a large degree, these incredible advances have been made possible through the development of new "languages" that allow humans to tell a computer what it should do. These so-called *computer languages* provide a way for us to express what we want done in a way that is more natural to the way we think and yet precise enough to control a computer.

We, as humans, are also phenomenal computing devices, but the way we think and communicate is generally a far cry from the way computers "think" and communicate. Computer languages provide a way of bridging this gap. But, the gap between computers and humans is vast and, for those new to computer programming, these languages can often be tremendously challenging to master. There are three important points that one must keep in mind when learning computer languages.

First, these languages are *not* designed to provide a means for having a two-way dialog with a computer. These languages are more like "instruction sets" where the human specifies what the computer should do. The computer blindly follows these instructions. In some sense, computer languages provide a way for humans to communicate *to* computers and with these languages we also have to tell the computers how we want them to communicate back to us (and it is extremely rare that we want a computer to communicate information back to us in the same language we used to communicate to it).

Second, unlike with natural languages[1], there is no ambiguity in a computer language. Statements in natural languages are often ambiguous while also containing redundant or superfluous content. Often the larger context in which a statement is made serves to remove the ambiguity while the redundant content allows us to make sense of a statement even if we miss part of it. As you will see, there may be a host of different ways to write statements in a computer language that ultimately lead to the same outcome. *However*, the path by which an outcome is reached is precisely determined by the statements/instructions that are provided to the computer. Note that we will often refer to statements in a computer language as "computer code" or simply "code."[2] We will call a collection of statements that serves to complete a desired task a *program*.[3]

The third important point about computer languages is that a computer can never infer meaning or intent. You may have a very clear idea of what you want a computer to do, but if you do not explicitly state your desires using precise *syntax* and *semantics*, the chances of obtaining the desired outcome are exceedingly small. When we say syntax, we essentially mean the rules of grammar

[1] By natural languages we mean languages that humans use with each other.

[2] This has nothing to do with a secret code nor does code in this sense imply anything to do with encryption.

[3] A program that is written specifically to serve the needs of a user is often called an *application* . We will not bother to distinguish between programs and applications.

and punctuation in a language. When writing natural languages, the introduction of a small number of typographical errors, although perhaps annoying to the reader, often does not completely obscure the underlying information contained in the writing. On the other hand, in some computer languages even one small typographical error in a computer program, which may be tens of thousands of lines of code, can often prevent the program from ever running. The computer can't make sense of the entire program so it won't do anything at all.[4] A show-stopping typographical error of syntax, i.e., a syntactic bug, that prevents a program from running is actually often preferable to other kinds of typographical errors that allow the code to run but, as a consequence of the error, the code produces something other than the desired result. Such typographical errors, whether they prevent the program from running or allow the program to run but produce erroneous results, are known as *bugs*.

A program may be written such that it is free of typographical errors and does precisely what the programmer said it should do and yet the output is still not what was desired. In this case the fault lies in the programmer's thinking: the programmer was mistaken about the collection of instructions necessary to obtain the correct result. Here there is an error in the logic or the semantics, i.e., the meaning, of what the programmer wrote. This type of error is still a "bug." The distinction between syntactic and semantic bugs will become more clear as you start to write your own code so we won't belabor this distinction now.

1.3 Python

There are literally thousands of computer languages. There is no single computer language that can be considered the best. A particular language may be excellent for tackling problems of a certain type but be horribly ill-suited for solving problems outside the domain for which it was designed. Nevertheless, the language we will study and use, Python, is unusual in that it does so many things and does them so well. It is relatively simple to learn, it has a rich set of features, and it is quite expressive (so that typically just a few lines of code are required in order to accomplish what would take many more lines of code in other languages). Python is used throughout academia and industry. It is very much a "real" computer language used to address problems on the cutting edge of science and technology. Although it was not designed as a language for teaching computer programming or algorithmic design, Python's syntax and idioms are much easier to learn than those of most other full-featured languages.

When learning a new computer language, one typically starts by considering the code required to make the computer produce the output "`Hello World!`"[5] With Python we must pass our code through the Python *interpreter*, a program that reads our Python statements and acts in accordance with these statements (more will be said below about obtaining and running Python). To have Python produce the desired output we can write the statement shown in Listing 1.1.

---

[4]The computer language we will use, Python, is not like this. Typically Python programs are executed as the lines of code are read, i.e., it is an *interpreted* language. Thus, egregious syntactic bugs may be present in the program and yet the program may run properly if, because of the flow of execution, the flawed statements are not executed. On the other hand, if a bug is in the flow of execution in a Python program, generally all the statements prior to the bug will be executed and then the bug will be "uncovered." We will revisit this issue in Chap. 11.

[5]You can learn more about this tradition at **en.wikipedia.org/wiki/Hello_world_program**.

Listing 1.1 A simple Hello-World program in Python.

```
print("Hello World!")
```

This single statement constitutes the entire program. It produces the following text:

```
Hello World!
```

This text output is terminated with a "newline" character, as if we had hit "return" on the keyboard, so that any subsequent output that might have been produced in a longer program would start on the next line. Note that the Python code shown in this book, as well as the output Python produces, will typically be shown in `Courier` font. The code will be highlighted in different ways as will become more clear later.

If you ignore the punctuation marks, you can read the code in Listing 1.1 aloud and it reads like an English command. Statements in computer languages simply do not get much easier to understand than this. Despite the simplicity of this statement, there are several questions that one might ask. For example: Are the parentheses necessary? The answer is: Yes, they are. Are the double-quotation marks necessary? Here the answer is yes and no. We do need to quote the desired output but we don't necessarily have to use double-quotes. In our code, when we surround a string of characters, such as `Hello World!`, in quotation marks, we create what is known as a *string literal*. (Strings will be shown in a bold green **`Courier`** font.) Python subsequently treats this collection of characters as a single group. As far as Python is concerned, there is a single argument, the string "`Hello World!`", between parentheses in Listing 1.1. We will have more to say about quotation marks and strings in Sec. 2.5 and Chap. 9.

Another question that might come to mind after first seeing Listing 1.1 is: Are there other Python programs that can produce the same output as this program produces? The answer is that there are truly countless programs we could write that would produce the same output, but the program shown in Listing 1.1 is arguably the simplest. However, let us consider a couple of variants of the Hello-World program that produce the exact same output as the previous program.[6] First consider the variant shown in Listing 1.2.

Listing 1.2 A variant of the Hello-World program that uses a single `print()` statement but with two arguments.

```
print("Hello", "World!")
```

In both Listings 1.1 and 1.2 we use the `print()` function that is provided with Python to obtain the desired output. Typically when referring to a function in this book (as opposed to in the code itself), we will provide the function name (in this case `print`) followed by empty parentheses. The parentheses serve to remind us that we are considering a function. What we mean in Python

[6]We introduce these variants because we want to emphasize that there's more than one way of writing code to generate the same result. As you'll soon see, it is not uncommon for one programmer to write code that differs significantly in appearance from that of another programmer. In any case, don't worry about the details of the variants presented here. They are merely presented to illustrate that seeming different code can nevertheless produce identical results.

when we say *function* and the significance of the parentheses will be discussed in more detail in Chap. 4.

The `print()` function often serves as the primary means for obtaining output from Python, and there are a few things worth pointing out now about `print()`. First, as Listing 1.1 shows, `print()` can take a single argument or parameter,[7] i.e., as far as Python is concerned, between the parentheses in Listing 1.1, there is a single argument, the string `Hello World!`. However, in Listing 1.2, the `print()` function is provided with two parameters, the string `Hello` and the string `World!`. These parameters are separated by a comma. The `print()` function permits an arbitrary number of parameters. It will print them in sequence and, by default, separate them by a single blank spaces. Note that in Listing 1.2 there are no spaces in the string literals (i.e., there are no blank spaces between the matching pairs of quotes). The space following the comma in Listing 1.2 has no significance. We can write:

```
print("Hello","World!")
```

or

```
print("Hello",                    "World!")
```

and obtain the same output. The mere fact that there are two parameters supplied to `print()` will ensure that, by default, `print()` will separate the output of these parameters by a single space.

Listing 1.3 uses two `print()` statements to obtain the desired output. Here we have added line numbers to the left of each statement. These numbers provide a convenient way to refer to specific statements and are not actually part of the program.

Listing 1.3 Another variant of the Hello-World program that uses two `print()` statements.

```
1 print("Hello", end=" ")
2 print("World!")
```

In line 1 of Listing 1.3 we see the string literal `Hello`. This is followed by a comma and the word `end` which is not in quotes. `end` is an *optional parameter* that specifies what Python should do at the end of a `print()` statement. If we do not add this optional parameter, the default behavior is that a line of output is terminated with a newline character so that subsequent output appears on a new line. We override this default behavior via this optional parameter by specifying what the `end` of the output should be. In the `print()` statement in the first line of Listing 1.3 we tell Python to set `end` equal to a blank space. Thus, subsequent output will start on the same line as the output produced by the `print()` statement in line 1 but there will be a space separating the subsequent output from the original output. The second line of Listing 1.3 instructs Python to write `World!`.[8]

We will show another Hello-World program but this one will be positively cryptic. Even most seasoned Python programmers would have some difficulty precisely determining the output produced by the code shown in Listing 1.4.[9] So, don't worry that this code doesn't make sense to you. It is, nevertheless, useful for illustrating two facts about computer programming.

[7] We will use the terms argument and parameter synonymously. As with arguments for a mathematical function, by "arguments" or "parameters" we mean the values that are *supplied* to the function, i.e., enclosed within parentheses.

[8] We will say more about this listing and the ways in which Python can be run in Sec. 1.6.

[9] The reason **`for`** and **`in`** appear in a bold blue font is because they are *keywords* as discussed in more detail in Sec. 2.6.

Listing 1.4 Another Hello-World program. The binary representation of each individual character is given as a numeric literal. The program prints them, as characters, to obtain the desired output.

```
for c in [0b1001000, 0b1100101, 0b1101100, 0b1101100,
 0b1101111, 0b0100000, 0b1010111, 0b1101111, 0b1110010,
 0b1101100, 0b1100100, 0b0100001, 0b0001010]:
   print(chr(c), end="")
```

Listing 1.4 produces the exact same output as each of the previous programs. However, while Listing 1.1 was almost readable as simple English, Listing 1.4 is close to gibberish. So, the first fact this program illustrates is that, although there may be many ways to obtain a solution (or some desired output as is the case here), clearly some implementations are better than others. This is something you should keep in mind as you begin to write your own programs. What constitutes the "best" implementation is not necessarily obvious because you, the programmer, may be contending with multiple objectives. For example, the code that yields the desired result most quickly (i.e., the fastest code) may not correspond to the code that is easiest to read, understand, or maintain.

In the first three lines of Listing 1.4 there are 13 different terms that start with `0b` followed by seven binary digits. These binary numbers are actually the individual representations of each of the characters of `Hello World!`. `H` corresponds to `1001000`, `e` corresponds to `1100101`, and so on.[10] As mentioned previously, the computer is really just dealing with zeros and ones. This brings us to the second fact Listing 1.4 serves to illustrate: it reveals to us some of the underlying world of a computer's binary thinking. But, since we don't think in binary numbers, this is often rather useless to us. We would prefer to keep binary representations hidden in the depths of the computer. Nevertheless, we have to agree (together with Python) how a collection of binary numbers should be interpreted. Is the binary number `1001000` the letter `H` or is it the integer number 72 or is it something else entirely? We will see later how we keep track of these different interpretations of the same underlying collection of zeros and ones.

1.4 Algorithmic Problem Solving

A computer language provides a way to tell a computer what we want it to do. We can consider a computer language to be a technology or a tool that aids us in constructing a solution to a problem or accomplishing a desired task. A computer language is not something that is timeless. It is exceedingly unlikely that the computer languages of today will still be with us 100 years from now (at least not in their current forms). However, at a more abstract level than the code in a particular language is the *algorithm*. An algorithm is the set of rules or steps that need to be followed to perform a calculation or solve a particular problem. Algorithms can be rather timeless. For example, the algorithm for calculating the greatest common denominator of two integers dates back thousands of years and will probably be with us for thousands of years more. There are efficient algorithms for sorting lists and performing a host of other tasks. The degree to which these algorithms are considered optimum is unlikely to change: many of the best algorithms of today are

[10] The space between `Hello` and `World!` has its own binary representation (`0100000`) as does the newline character that is used to terminate the output (`0001010`).

likely to be the best algorithms of tomorrow. Such algorithms are often expressed in a way that is independent of any particular computer language because the language itself is not the important thing—performing the steps of the algorithm is what is important. The computer language merely provides a way for us to tell the computer how to perform the steps in the algorithm.

In this book we are not interested in examining the state-of-the-art algorithms that currently exist. Rather, we are interested in developing your computer programming skills so that you can translate algorithms, whether yours or those of others, into a working computer program. As mentioned, we will use the Python language. Python possesses many useful features that facilitate learning and problem solving, but much of what we will do with Python mirrors what we would do in the implementation of an algorithm in any computer language. The algorithmic constructs we will consider in Python, such as looping structures, conditional statements, and arithmetic operations, to name just a few, are key components of most algorithms. Mastering these constructs in Python should enable you to more quickly master the same things in another computer language.

At times, for pedagogic reasons, we will not exploit all the tools that Python provides. Instead, when it is instructive to do so, we may implement our own version of something that Python provides. Also at times we will implement some constructs in ways that are not completely "Pythonic" (i.e., not the way that somebody familiar with Python would implement things). This will generally be the case when we wish to illustrate the way a solution would be implemented in languages such as C, C++, or Java.

Keep in mind that computer science and computer programming are much more about problem solving and algorithmic thinking (i.e., systematic, precise thinking) than they are about writing code in a particular language. Nevertheless, to make our problem-solving concrete and to be able to implement real solutions (rather than just abstract descriptions of a solution), we need to program in a language. Here that language is Python. But, the reader is cautioned that this book is *not* intended to provide an in-depth Python reference. On many occasions only as much information will be provided as is needed to accomplish the task at hand.

1.5 Obtaining Python

Python is open-source software available for free. You can obtain the latest version for Linux/Unix, Macintosh, and Windows via the download page at **`python.org`**. As of this writing, the current version of Python is 3.2.2. You should install this (or a newer version if one is available). There is also a 2.x version of Python that is actively maintained and available for download, but it is not compatible with Python 3.x and, thus, you should not install it.[11] Mac and Linux machines typically ship with Python pre-installed but it is usually version 2.x. Because this book is for version 3.x of Python, you must have a 3.x version of Python.

Computer languages provide a way of describing what we want the computer to do. Different implementations may exist for translating statements in a computer language into something that actually performs the desired operations on a given computer. There are actually several dif-

[11] When it comes to versions of software, the first digit corresponds to a major release number. Incremental changes to the major release are indicated with additional numbers that are separated from the major release with a "dot." These incremental changes are considered minor releases and there can be incremental changes to a minor release. Version 3.2.2 of Python is read as "version three-point-two-point-two" (or some people say "dot" instead of "point"). When we write version 3.x we mean any release in the version 3 series of releases.

ferent Python implementations available. The one that we will use, i.e., the one available from **python.org**, is sometimes called CPython and was written in the C programming language. Other implementations that exist include IronPython (which works with the Microsoft .NET framework), Jython (which was written in Java), and PyPy (which is written in Python). The details of how these different implementations translate statements from the Python language into something the computer understands is not our concern. However, it is worthwhile to try to distinguish between compilers and interpreters.

Some computer languages, such as FORTRAN, C, and C++, typically require that you write a program, then you *compile* it (i.e., have a separate program known as a compiler translate your program into executable code the computer understands), and finally you run it. The CPython implementation of Python is different in that we can write statements and have the Python *interpreter* act on them immediately. In this way we can instantly see what individual statements do. The instant feedback provided by interpreters, such as CPython, is useful in learning to program. An interpreter is a program that is somewhat like a compiler in that it takes statements that we've written in a computer language and translates them into something the computer understands. However, with an interpreter, the translation is followed immediately by execution. Each statement is executed "on the fly." [12]

1.6 Running Python

With Python we can use interactive sessions in which we enter statements one at a time and the interpreter acts on them. Alternatively, we can write all our commands, i.e., our program, in a file that is stored on the computer and then have the interpreter act on that stored program. In this case some compilation may be done behind the scenes, but Python will still not typically provide speeds comparable to a true compiled language.[13] We will discuss putting programs in files in Sec. 1.6.2. First, we want to consider the two most common forms of interactive sessions for the Python interpreter.

Returning to the statements in Listing 1.3, if they are entered in an interactive session, it is difficult to observe the behavior that was described for that listing because the `print()` statements have to be entered one at a time and output will be produced immediately after each entry. In Python we can have multiple statements on a single line if the statements are separated by a semicolon. Thus, if you want to verify that the code in Listing 1.3 is correct, you should enter it as shown in Listing 1.5.

[12] Compiled languages, such as C++ and Java, typically have an advantage in speed over interpreted languages such as Python. When speed is truly critical in an application, it is unlikely one would want to use Python. However, in most applications Python is "fast enough." Furthermore, the time required to develop a program in Python is typically much less than in other languages. This shorter development time can often more than compensate for the slower run-time. For example, if it takes one day to write a program in Python but a week to write it in Java, is it worth the extra development time if the program takes one second to run in Java but two seconds to run in Python? Sometimes the answer to this is definitely yes, but this is more the exception rather than the rule. Although it is beyond the scope of this book, one can create programs that use Python together with code written in C. This approach can be used to provide execution speeds that exceed the capabilities of programs written purely in Python.

[13] When the CPython interpreter runs commands from a file for the first time, it compiles a "bytecode" version of the code which is then run by the interpreter. The bytecode is stored in a file with a `.pyc` extension. When the file code is rerun, the Python interpreter actually uses the bytecode rather than re-interpreting the original code as long as the Python statements have not been changed. This speeds up execution of the code.

Listing 1.5 A Hello-World program similar to Listing 1.3 except that both `print()` statements are given on a single line. This form of the program is suitable for entry in an interactive Python session.

```
print("Hello", end=" "); print("World!")
```

1.6.1 Interactive Sessions and Comments

When you install Python, an application called IDLE will be installed on your system. On a Mac, this is likely to be in the folder `/Applications/Python 3.2`. On a Windows machine, click the `Start` button in the lower left corner of the screen. A window should pop up. If you don't see any mention of Python, click `All Programs`. You will eventually see a large listing of programs. There should be an entry that says `Python 3.2`. Clicking `Python 3.2` will bring up another list in which you will see `IDLE (Python GUI)` (GUI stands for Graphical User Interface).

IDLE is an integrated development environment (IDE). It is actually a separate program that stands between us and the interpreter, but it is not very intrusive—the commands we enter are still sent to the interpreter and we can obtain on-the-fly feedback. After starting IDLE, you should see (after a bit of boilerplate information) the Python interactive prompt which is three greater-than signs (>>>). At this point you are free to issue Python commands. Listing 1.6 demonstrates how the window will appear after the code from Listing 1.1 has been entered. For interactive sessions, programmer input will be shown in bold **`Courier`** font although, as shown in subsequent listings, comments will be shown in a slanted, orange *`Courier`* font.

Listing 1.6 An IDLE session with a Hello-World statement. Programmer input is shown in bold. The information on the first three lines will vary depending on the version and system.

```
1 Python 3.2.2 (v3.2.2:137e45f15c0b, Sep  3 2011, 17:28:59)
2 [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
3 Type "copyright", "credits" or "license()" for more information.
4 >>> print("Hello World!")
5 Hello World!
6 >>>
```

To execute the `print()` statement shown on line 4, one merely types in the statement as shown and then hits the enter (or return) key.

An alternative way of running an interactive session with the Python interpreter is via the *command line*.[14] To accomplish this on a Mac, go to the folder `/Applications/Utilities` and open the application `Terminal`. After `Terminal` has started, type `python3` and hit return.

[14]IDLE is built using a graphics package known as tkinter which also comes with Python. When you use tkinter graphics commands, sometimes they can interfere with IDLE so it's probably best to open an interactive session using the command line instead of IDLE.

For Windows, click the `Start` button and locate the program `Python (command line)` and click on it.

Listing 1.7 shows the start of a command-line based interactive session. An important part of programming is including comments for humans. These comments are intended for those who are reading the code and trying to understand what it does. As you write more and more programs, you will probably discover that the comments you write will often end up aiding you in trying to understand what *you* previously wrote! The programmer input in Listing 1.7 starts with four lines of comments which are shown in a slanted, orange `Courier` font. (One would usually not include comments in an interactive session, but they are appropriate at times—especially in a classroom setting!)

Listing 1.7 A command-line session with a Hello-World statement. Here lines 4 through 7 are purely comments. Comment statements will be shown in a slanted, `Courier` font (instead of bold).

```
1  Python 3.2.2 (v3.2.2:137e45f15c0b, Sep  3 2011, 17:28:59)
2  [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
3  Type "help", "copyright", "credits" or "license" for more information.
4  >>> # This is a comment.  The interpreter ignores everything
5  ... # after the "#" character.  In the command-line environment
6  ... # the prompt will change to "..." if "#" is the first
7  ... # character of the previous line.
8  ... print("Hello", "World!") # Comment following a statement.
9  Hello World!
10 >>>
```

Python treats everything after the character # as a comment, i.e., it simply ignores this text as it is intended for humans and not the computer. The character # is called pound, hash, number sign, or (rarely) octothorp. As line 8 demonstrates, a comment can appear on the same line as a statement. (The # character does not indicate a comment if it is embedded in a string literal.) A hash is used to indicate a comment whether using the Python interpreter or writing Python code in a file. (String literals can also be used as comments as discussed in connection with doc strings in Sec. 4.3.)

As Listing 1.7 shows, sometimes the Python prompt changes to three dots (`...`). This happens in the command-line environment when Python is expecting more input (we will see later the situations in which Python expects more input). In the command-line environment, when a line starts with a comment, Python will change the prompt in the following line to three dots. However, as shown in line 8 in Listing 1.7, a statement entered after the three dots will be executed as usual. Things behave slightly differently in IDLE: the prompt will remain >>> in a line following a line of comment. In this book, when showing an interactive session, we will typically adopt the IDLE convention in which the prompt following a comment is still >>>.

There is one important feature of the interactive environment that, though useful, can lead to confusion for those new to Python. The interactive environment will display the result of expressions (what we mean by an expression will be discussed further in Chap. 2) and will echo a literal that is entered. So, for example, in the interactive environment, if we want to print `Hello`

World!, we don't need to use a print() statement. We can merely enter the string literal and hit return. Listing 1.8 illustrates this where, on line 1, the programmer entered the literal string and on line 2 Python echoed the message back. However, note that, unlike in Listing 1.7, the output is surrounded by single quotes. We will have more to say about this below and in the next chapter.

Listing 1.8 When a literal is entered in the interactive environment, Python echoes the literal back to the programmer.[15]

```
>>> "Hello World!"
'Hello World!'
>>>
```

1.6.2 Running Commands from a File

There are various ways you can store commands in a file and then have the Python interpreter act on them. Here we will just consider how this can be done using IDLE. After starting IDLE, the window that appears with the interactive prompt is titled Python Shell. Go to the File menu and select New Window. Alternatively, on a Mac you can type command-N, while on a Windows machine you would type control-N. Henceforth, when we refer to a keyboard shortcut such as C-N, we mean command-N on a Mac and control-N under Windows. The letter following "C-" will vary depending on the shortcut (although this trailing letter will be written in uppercase, it is not necessary to press the shift key).

After selecting New Window or typing C-N, a new window will appear with the title Untitled. No interactive prompt appears. You can enter Python statements in this window but the interpreter will not act on these statements until you explicitly ask it to. Once you start typing in this window the title will change to *Untitled*. The asterisks indicate that the contents of this window have changed since the contents were last saved to a file.

Before we can run the statements in the window we opened, we must save the file. To do this, either select Save from the File menu or type C-S. This will bring up a "Save" window where you indicate the folder and the file name where you want the contents of the window to be saved. Whatever file name you choose, you should save it with an extension of ".py" which indicates this is a Python file. Once you have saved the file, the title of the Window will change to reflect the new file name (and the folder where it is stored).

Once the file has been saved, it can be run through the Python interpreter. To do this, you can either go to the Run menu and select Run Module or you can type F5 (function key 5—on a Mac laptop you will have to hold down the fn key, too). To illustrate what happens now, assume a programmer has entered and saved the two lines of code shown in Listing 1.9.

Listing 1.9 Two lines of code that we assume have been saved to a file via IDLE. (This code is not entered directly in the interactive environment.)

[15] If an *expression* is entered in the interactive environment, Python displays the result of the expression. Expressions are discussed in Chap. 2.

```
"Hello World!"
print("Have we said enough hellos?")
```

When this is run, the focus will switch back to the Python Shell window. The window will contain the output shown in Listing 1.10.

Listing 1.10 The output that is produced by running the code in Listing 1.9

```
>>> =========================== RESTART ============================
>>>
Have we said enough hellos?
>>>
```

The output shown in the first two lines is not something our code produced. Rather, whenever IDLE runs the contents of a file, it restarts the Python interpreter (thus anything you previously defined, such as variables and functions, will be lost—this provides a clean start for running the code in the file). This restart is announced as shown in line 1; it is followed by a "blank line," i.e., a line with the interactive prompt but nothing else. Then, in line 3 of Listing 1.10, we see the output produced by the `print()` statement in line 2 of Listing 1.9. However, note that *no* output was produced by the `Hello World!` literal on line 1 of Listing 1.9. In the interactive environment, `Hello World!` is echoed to the screen, but when we put statements in a file, we have to explicitly state what we want to show up on the screen.

If you make further changes to the file, you must save the contents before running the file again.[16] To run the file you can simply type C-S (the save window that appeared when you first type C-S will not reappear—the contents will be saved to the file you specified previously) and then F5.

1.7 Bugs

You should keep in mind that, for now, you cannot hurt your computer with any bugs or errors you may write in your Python code. Furthermore, any errors you make will not crash the Python interpreter. Later, when we consider opening or manipulating files, we will want to be somewhat cautious that we don't accidentally delete a file, but for now you shouldn't hesitate to experiment with code. If you ever have a question about whether something will or won't work, there is no harm in trying it out to see what happens.

Listing 1.11 shows an interactive session in which a programmer wanted to find out what would happen when entering modified versions of the Hello-World program. In line 2, the programmer wanted to see if `Print()` could be use instead of `print()`. In line 7 the programmer attempted to get rid of the parentheses. And, in line 13, the programmer tried to do away with the quotation marks. Code that produces an error will generally be shown in red.

[16] Note that we will say "run the file" although it is more correct to say "run the program contained in the file."

Listing 1.11 Three buggy attempts at a Hello-World program. (Code shown in red produces an error.)

```
>>> # Can I write Print()?
>>> Print("Hello World!")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'Print' is not defined
>>> # Can I get rid of the parentheses?
>>> print "Hello World!"
  File "<stdin>", line 2
    print "Hello World!"
                       ^
SyntaxError: invalid syntax
>>> # Do I need the quotation marks?
>>> print(Hello World!)
  File "<stdin>", line 2
    print(Hello World!)
                     ^
SyntaxError: invalid syntax
```

For each of the attempts, Python was unable to perform the task that the programmer seemingly intended. Again, the computer will never guess what the programmer intended. We, as programmers, have to state precisely what we want.

When Python encounters errors such as these, i.e., syntactic errors, it *raises* (or *throws*) *an exception*. Assuming we have not provided special code to handle an exception, an error message will be printed and the execution of the code will halt. Unfortunately, these error messages are not always the most informative. Nevertheless, these messages should give you at least a rough idea where the problem lies. In the code in Listing 1.11 the statement in line 2 produced a `NameError` exception. Python is saying, in line 5, that `Print` is not defined. This seems clear enough even if the two lines before are somewhat cryptic. The statements in lines 7 and 13 resulted in `SyntaxError` exceptions (as stated in lines 11 and 17). Python uses a caret (^) to point to where it thinks the error may be in what was entered, but one cannot count on this to truly show where the error is.

1.8 The `help()` Function

The Python interpreter comes with a `help()` function. There are two ways to use `help()`. First, you can simply type `help()`. This will start the online help utility and the prompt will change to `help>`. You then get help by typing the name of the thing you are interested in learning about. Thus far we have only considered one built-in function: `print()`. Listing 1.12 shows the message provided for the `print()` function. To exit the help utility, type `quit`.

Listing 1.12 Information provided by the online help utility for the `print()` function.

```
help> print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
```

When you are just interested in obtaining help for one particular thing, often you can provide that thing as an argument to the `help()` function. For example, at the interactive prompt, if one types `help(print)`, Python will return the output shown in Listing 1.12. (When used this way, you cannot access the other topics that are available from within the help utility.)

1.9 Comments on Learning New Languages

When learning a new skill, it is often necessary to practice over and over again. This holds true for learning to play an instrument, play a new sport, or speak a new language. If you have ever studied a foreign language, as part of your instruction you undoubtedly had to say certain things over and over again to help you internalize the pronunciation and the grammar.

Learning a computer language is similar to learning any new skill: You must actively practice it to truly master it. As with natural languages, there are two sides to a computer language: the ability to comprehend the language and the ability to speak or write the language. Comprehension (or analysis) of computer code is *much* easier than writing (or synthesis of) computer code. When reading this book or when watching somebody else write code, you may be able to easily follow what is going on. This comprehension may lead you to think that you've "got it." However, when it comes to *writing* code, at times you will almost certainly feel completely lost concerning something that you thought you understood. To minimize such times of frustration, it is vitally important that you practice what has been presented. Spend time working through assigned exercises, but also experiment with the code yourself. Be an active learner. As with learning to play the piano, you can't learn to play merely by watching somebody else play!

You should also keep in mind that you can learn quite a bit from your mistakes. In fact, in some ways, the more mistakes you make, the less likely you are to make mistakes in the future. Spending time trying to decipher error messages that are produced in connection with relatively simple code will provide you with the experience to more quickly decipher bugs in more complicated code. Pixar Animation Studios has combined state-of-the-art technology and artistic talent to produce several of the most successful movies of all time. The following quote is from Lee Unkrich, a director at Pixar, who was describing the philosophy they have at Pixar.[17] You would do well to adopt this philosophy as your own in your approach to learning to program:

[17] From *Imagine: How Creativity Works*, by Jonah Lehrer, Houghton Mifflin Harcourt, 2012, pg. 169.

> We know screwups are an essential part of what we do here. That's why our goal is simple: We just want to screw up as quickly as possible. We want to fail fast. And then we want to fix it.
>
> — Lee Unkrich

1.10 Chapter Summary

Comments are indicated by a hash sign # (also known as the pound or number sign). Text to the right of the hash sign is ignored. (But, hash loses its special meaning if it is part of a string, i.e., enclosed in quotes.)

`print()`: is used to produce output. The optional arguments `sep` and `end` control what appears between values and how a line is terminated, respectively.

Code may contain syntactic bugs (errors in grammar) or semantic bugs (error in meaning). Generally, Python will only raise, or throw, an exception when the interpreter encounters a syntactic bug.

`help()`: provides help. It can be used interactively or with a specific value specified as its argument.

1.11 Review Questions

Note: Many of the review questions are meant to be challenging. At times, the questions probe material that is somewhat peripheral to the main topic. For example, questions may test your ability to spot subtle bugs. Because of their difficulty, you should not be discouraged by incorrect answers to these questions but rather use challenging questions (and the understanding a correct answer) as opportunities to strengthen your overall programming skills.

1. True or False: When Python encounters an error, it responds by raising an exception.

2. A comment in Python is indicated by a:

   (a) colon (`:`)

   (b) dollar sign (`$`)

   (c) asterisk (`*`)

   (d) pound sign (`#`)

3. What is the output produced by `print()` in the following code?

   ```
   print("Tasty organic", "carrots.")
   ```

   (a) `"Tasty organic", "carrots."`

   (b) `"Tasty organic carrots."`

   (c) `Tasty organic carrots.`

   (d) `Tasty organic", "carrots.`

4. What is the output produced by `print()` in the following code?

```
print("Sun ripened ","tomatoes.")
```

(In the following, ␣ indicates a single blank space.)

(a) `Sun ripened␣tomatoes.`
(b) `"Sun ripened␣","tomatoes."`
(c) `"Sun ripened␣",␣"tomatoes."`
(d) `Sun ripened␣␣tomatoes.`

5. What is the output produced by `print()` in the following code?

```
print("Grass fed ","beef.", end="")
```

(In the following, ␣ indicates a single blank space.)

(a) `Grass fed␣beef.`
(b) `"Grass fed␣","beef."`
(c) `"Grass fed␣",␣"beef."`
(d) `Grass fed␣␣beef.`

6. What is the output produced by the following code? (In the following, ␣ indicates a single blank space.)

```
print("Free range␣", end="␣␣")
print("chicken.")
```

(a) `Free range␣␣␣chicken.`
(b) `Free range␣␣␣`
`chicken.`
(c) `"Free range"␣"␣␣"␣"chicken."`
(d) `Free range␣␣␣␣␣chicken.`
(e) `Free range"␣␣"chicken.`

7. What is the output produced by the following code? (In the following, ␣ indicates a single blank space.)

```
print("Free range␣", end="␣␣"); print("chicken.")
```

(a) `Free range␣␣␣chicken.`
(b) `Free range␣␣␣`
`chicken.`

(c) `"Free range"␣"␣␣"␣"chicken."`

(d) `Free range␣␣␣␣␣chicken.`

(e) `Free range"␣␣"chicken.`

8. The follow code appears in a *file*:

```
"Hello"
print(" world!")
```

What output is produced when this code is interpreted? (In the following, ␣ indicates a single blank space.)

(a) `Hello`
`␣world!`

(b) `Hello␣world!`

(c) `␣world!`

(d) `world!`

ANSWERS: 1) True; 2) d; 3) c; 4) d; 5) a; 6) a; 7) a; 8) c.

Chapter 2

Core Basics

In this chapter we introduce some of the basic concepts that are important for understanding what is to follow. We also introduce a few of the fundamental operations that are involved in the vast majority of useful programs.

2.1 Literals and Types

Section 1.3 introduced the string literal. A string literal is a collection of characters enclosed in quotes. In general, a *literal* is code that exactly represents a value. In addition to a string literal, there are numeric literals.

There are actually a few different types of numeric values in Python. There are integer values which correspond to whole numbers (i.e., the countable numbers that lack a fractional part). In Python, an integer is known as an `int`. A number that can have a fractional part is called a `float` (a `float` corresponds to a real number). For example, `42` is an `int` while `3.14` is a `float`. The presence of the decimal point makes a number a `float`—it doesn't matter if the fractional part is zero. For example, the literal `3.0`, or even just `3.`, are also `float`s[1] despite the fact that these numbers have fractional parts of zero.[2]

There are some important differences between how computers work with `int`s and `float`s. For example, `int`s are stored exactly in the computer while, in general, `float`s are only stored approximately. This is a consequence of the fact that we are entering numbers as decimal numbers, i.e., numbers in the base 10 counting system, but the computer stores these numbers internally as a finite number of binary (base 2) digits. For integers we can represent a number exactly in any counting system. However, for numbers with fractional parts, this is no longer true. Consider the number one-third, i.e., 1/3. To represent this as a decimal number (base 10) requires an infinite number of digits: $0.3333\cdots$. But, if one uses the base 3 (ternary) counting system, one-third is

From the file: `core-basics.tex`

[1]We will write the plural of certain nouns that have meaning in Python as a mix of fonts. The noun itself will be in `Courier` and the trailing "s" will be in Times-Roman, e.g., `float`s.

[2]Other numeric data types that Python provides are `complex`, `Decimal`, and `Fraction`. Of these, only `complex` numbers can be entered directly as literals. They will be considered in Chap. 8. `Decimal`s provide a way of representing decimal numbers in a manner that is consist with how humans think of decimal numbers and do not suffer from the approximation inherent in `float`s. `Fraction`s are rational numbers where the numerator and denominator are integers.

represented with a single digit to the right of the decimal sign, i.e., 0.1_3.[3] This illustrates that when converting from one counting system to another (i.e., one base to another), the fractional part of the number that could be represented by a finite number of digits in one counting system may require an infinite number of digits in the other counting system. Because computers cannot store an infinite number of digits, the finite number of digits that are stored for a `float` is an approximation of the true number.[4]

As mentioned in Sec. 1.6.1, when we enter a literal in the interactive environment, Python merely echoes the literal back to us. This is true of both numeric and string literals as Listing 2.1 illustrates.

Listing 2.1 Entry of numeric and string literals in the interactive environment.

```
1  >>> 7          # int literal.
2  7
3  >>> 3.14       # float literal.
4  3.14
5  >>> -2.00001 # Negative numbers are indicated with a negative sign.
6  -2.00001
7  >>> +2.00      # Can explicitly use positive sign, but not needed.
8  2.0
9  >>> 1.23e3     # Exponential notation: positive exponent.
10 1230.0
11 >>> 1.23e-3    # Exponential notation: negative exponent.
12 0.00123
13 >>> 1e0        # Exponential notation: decimal point not required.
14 1.0
15 >>> "Literally unbelievable." # String in double quotes.
16 'Literally unbelievable.'
17 >>> 'Foo the Bar'              # String in single quotes.
18 'Foo the Bar'
```

Line 5 shows the entry of a negative number. One can use the positive sign to indicate a positive number, as is done in line 7, but if no sign is provided, the number is assumed to be positive.

Lines 9, 11, and 13 demonstrate the entry of a number using exponential notation. From your calculator you may already be familiar with the exponential notation in which the letter `e` is used as shorthand for "$\times 10$" and the number to the right of `e` is the exponent of 10. Thus, when we write `1.23e-3` we mean 1.23×10^{-3} or 0.00123. Note think in line 13 we have entered `1e0` which is equivalent to $1 \times 10^0 = 1$. You might think that since there is no decimal point, `1e0` should be an integer. However, the output of `1.0` shown on line 14 indicates that it is actually a `float`. So, when a decimal point appears in a number *or* when `e` appears in a number, the number is a `float`.

In lines 15 and 17 string literals are entered. In line 15 the string is enclosed in double quotes but what Python echoes back in line 16 is enclosed in single quotes. Line 17 shows that we can

[3]The subscript 3 indicates this is a number in base 3. If you are unfamiliar with counting in numbering systems other than 10, it is not a concern. We will revisit this issue later.

[4]For a `float` one obtains about 15 decimal digits of accuracy. Thus, although a `float` may be an approximation of the true number, 15 digits of accuracy is typically sufficient to meet the needs of an application.

also use single quotes to enter a string literal. In line 18 Python echoes this back and again uses single quotes.

The *type* of various things, including literals, can be determined using the `type()` function.[5] In finished programs it is rare that one needs to use this function, but it can be useful for debugging and helping to understand code. Listing 2.2 illustrates what happens when various literals are passed as an argument to `type()`.

Listing 2.2 Using the `type()` function to determine the type of various literals.

```
1  >>> type(42)
2  <class 'int'>
3  >>> type(-27.345e14)
4  <class 'float'>
5  >>> type("Hello World!")
6  <class 'str'>
7  >>> type("A") # Single character, double quotes.
8  <class 'str'>
9  >>> # Single character, single quotes; looks like int but isn't!
10 >>> type('2')
11 <class 'str'>
12 >>> type("")  # Empty string.
13 <class 'str'>
14 >>> type('-27.345e14')  # Looks like float but isn't!
15 <class 'str'>
```

In line 1 the programmer asks for the type of the literal `42`. Line 2 reports that it is an `int`. The response is a little more involved than simply saying this is an `int`—the word `class` indicates that `42` is actually an instance of the `int` class. The notion of an "instance of a class" relates to object oriented programming concepts and is something that doesn't concern us yet. Suffice it simply to say that "`42` is an `int`."

In line 3 a `float` is passed as an argument to `type()` while in line 5 the argument is a string. Line 6 shows that Python identifies a string as a `str`. We will take string to be synonymous with `str` and integer to be synonymous with `int`. (For `floats` we will simply write `float`.)

Lines 7 through 11 show that a single character, regardless of the type of quotation marks that enclose it, is considered a string.[6] Lines 12 and 13 show the type of an *empty string*: there are no characters enclosed between the quotation marks in line 12 and yet this is still considered a string. An empty string may seem a rather useless literal, but we will see later that it plays a useful role in certain applications.

Lines 10 and 14 of Listing 2.2 are of interest in that, to a human, the characters in the string represent a numeric value; *nevertheless*, the literal is, in fact, a string. Keep in mind that just because something might appear to be a numeric value doesn't mean it is. To hint at how this may lead to some confusion, consider the code in Listing 2.3. In line 1 the `print()` function has two

[5] Instead of saying "various things," it would be more correct to say the *objects* in our code. We discuss what objects are in Chap. 5.

[6] This is not the case in some other languages such as C and C++.

arguments: an `int` and a `str`. However, in line 2 the output produced for each of these arguments is identical.

Listing 2.3 Printing numeric values and strings that look like numeric values.

```
1 >>> print(42, "42") # An int and a str that looks like an int.
2 42 42
3 >>> print('3.14')    # A str that looks like a float.
4 3.14
5 >>> print(3.14)      # A float.
6 3.14
```

In line 3 a `str` is printed while in line 5 a `float` is printed. The output of these two statements is identical. But, there are things that we can do with numeric values that we cannot do with strings and, conversely, there are things we can do with strings that we cannot do with numeric values. Thus, we need to be somewhat mindful of type. When bugs are encountered, often the best tool for debugging is the `print()` function. However, as the code in Listing 2.3 demonstrates, if we ever encounter problems related to type, the `print()` function, by itself, might not be the best tool for sorting out the problem since it may display data of different types identically.

2.2 Expressions, Arithmetic Operators, and Precedence

An *expression* is code that returns data, or, said another way, evaluates to a value.[7] In Python, it is more correct to say that an expression returns an *object* instead of a *value* or *data*, but we will ignore this distinction for now (and return to it later). The literal values considered in Sec. 2.1 are the simplest form of an expression. When we write a literal, we are providing (fixed) data. We can also have expressions that evaluate to a new value. For example, we can add two numbers that may be entered as literals. The result of this expression is the new value given by the sum of the numbers. When an expression is entered in an interactive environment, Python will display the result of the expression, i.e., the value to which the expression evaluates.

Python provides the standard arithmetic operations of addition, subtraction, multiplication, and division which are invoked via the *operators* `+`, `-`, `*`, and `/`, respectively. The numbers to either side of these operators are known as the *operands*. Listing 2.4 demonstrates the use of these operators.

Listing 2.4 Demonstration of the basic arithmetic operators.

```
1 >>> 4 - 5         # Difference of two ints.
2 -1
3 >>> 5 + 4         # Sum of two ints.
4 9
5 >>> 5 + 4.        # Sum of int and float.
```

[7]Rather than adhering to traditional usage where the noun *data* is only plural (with a singular form of datum), we will adopt modern usage where data can be either singular or plural.

```
6  9.0
7  >>> 5 + 4 * 3     # * has higher precedence than +.
8  17
9  >>> (5 + 4) * 3   # Parentheses can change order of operations.
10 27
11 >>> 5 / 4         # float division of two ints.
12 1.25
13 >>> 10 / 2        # float division of two ints.
14 5.0
15 >>> 20 / 4 / 2    # Expressions evaluated left to right.
16 2.5
17 >>> (20 / 4) / 2 # Parentheses do not change order of operation here.
18 2.5
19 >>> 20 / (4 / 2) # Parentheses do change order of operation here.
20 10.0
```

Lines 1 through 4 show that the sum or difference of two `int`s is an `int`. Lines 5 and 6 show that the sum of an `int` and a `float` yields a `float` (as indicated by the decimal point in the result in line 6).[8] When an arithmetic operation involves an `int` and a `float`, Python has to choose a data type for the result. Regardless of whether or not the fractional part of the result is zero, the result is a `float`. Specifically, the `int` operand is converted to a `float`, the operation is performed, and the result is a `float`.

The expression in line 7 uses both addition and multiplication. We might wonder if the addition is done first, since it appears first in the expression, or if the multiplication is done first since in mathematics multiplication has higher precedence than addition. The answer is, as indicated by the result on line 8, that Python follows the usual mathematical rules of precedence: Addition and subtraction have lower precedence than multiplication and division. Thus, the expression `5 + 3 * 4` is equivalent to `5 + (3 * 4)` where operations in parentheses always have highest precedence. Line 9 shows that we can use parentheses to change the order of operation.

Lines 11 and 13 show the "`float` division" of two `int`s. In line 11 the numerator and denominator are such that the quotient has a non-zero fractional part and thus we might anticipate obtaining a `float` result. However, in line 13, the denominator divides evenly into the numerator so that the quotient has a fractional part of zero. Despite this fact, the result is a `float`. This is a consequence of the fact that we are using a single slash (`/`) to perform the division. Python also provide another type of division, floor division, which is discussed in Sec. 2.8.

In line 15 there are two successive divisions: `20 / 4 / 2`. Here the operations are evaluated left to right. First `20` is divided by `4` and then the result of that is divided by `2`, ultimately yielding `2.5`. If we are ever unsure of the order of operation (or if we merely think it will enhance the clarity of an expression), we can always add parentheses as line 17 illustrates. Lines 15 and 17 are completely equivalent and the parentheses in line 17 are merely "decorative."

The operators in Listing 2.4 are *binary operators* meaning that they take two operands—one operand is to the left of the operator and the other is to the right. However, plus and minus signs can also be used as *unary operators* where they take a single operand. This was illustrated in lines

[8]Note that the blank spaces that appear in these expressions are merely used for clarity. Thus `5 + 4` is equivalent to `5+4`.

5 and 7 of Listing 2.1 where the plus and minus signs are used to establish the sign of the numeric value to their right. Thus, in those two expressions, the unary operators have a single operand which is the number to their right.

Finally, keep in mind that in the interactive environment, if an expression is the only thing on a line, the value to which that expression evaluates is shown to the programmer upon hitting return. As was discussed in connection with Listing 1.9, this is not true if the same expression appears in a file. In that case, in order to see the value to which the expression evaluates, the programmer has to use a `print()` statement to explicitly generate output.

2.3 Statements and the Assignment Operator

A *statement* is a complete command (which may or may not produce data). We have already been using `print()` statements. `print()` statements produce output but they do not produce data. At this point this may be a confusing distinction, but we will return to it shortly.

In the previous section we discussed the arithmetic operators that you probably first encountered very early in your education (i.e., addition, subtraction, etc.). What you learned previously about these operators still pertains to how these operators behave in Python. Shortly after being introduced to addition and subtraction, you were probably introduced to the equal sign (=). Unfortunately, ***the equal sign in Python does not mean the same thing as in math!***

In Python (and in many other computer languages), the equal sign is the *assignment operator*. This is a binary operator. When we write the equal sign we are telling Python: "Evaluate the operand/expression on the right side of the equal sign and assign that to the operand on the left." The left operand must be something to which we can assign a value. Sometimes this is called an "lvalue" (pronounced ell-value). In Python we often call this left operand an *identifier* or a *name*. In many other computer languages (and in Python) a more familiar name for the left operand is simply *variable*.[9] You should be familiar with the notion of variables from your math classes. A variable provides a name for a value. The term variable can also imply that something is changing, but this isn't necessarily the case in programs—a programmer may define a variable that is associated with a value that doesn't change throughout a program.

In a math class you might see the following statements: $x = 7$ and $y = x$. In math, how is x related to y if x changes to, say, 9? Since we are told that y is equal to x, we would say that y must also be equal to 9. In general, ***this is not how things behave in computer languages!***

In Python we can write statements that *appear* to be identical to what we would write in a math class, but they are not the same. Listing 2.5 illustrates this. (For brevity, we will use single-character variable names here. The rules governing the choice of variable names are discussed in Sec. 2.6.)

Listing 2.5 Demonstration of the assignment operator.

```
1 >>> x = 7                           # Assign 7 to variable x.
2 >>> y = x                           # Assign current value of x to y.
3 >>> print("x =", x, ", y =", y)    # See what x and y are.
```

[9] In Sec. 2.7 we present more details about what we mean by an *identifier* or a *name* in Python. We further consider the distinction between an lvalue and a variable in Sec. 6.6.

```
x = 7 , y = 7
>>> x = 9                          # Assign new value to x.
>>> print("x =", x, ", y =", y)   # See what x and y are now.
x = 9 , y = 7
```

In line 1 we assign the value 7 to the variable x. In line 2, we do *not* equate x and y. Instead, in line 2, we tell Python to evaluate the expression to the right of the assignment operator. In this case, Python merely gets the current value of x, which is 7, and then it assigns this value to the variable on the left, which is the variable y. Line 3 shows that, in addition to literals, the print() function also accepts variables as arguments. The output on line 4 shows us that both x and y have a value of 7. However, although these values are equal, they are distinct. The details of what Python does with computer memory isn't important for us at the moment, but it helps to imagine that Python has a small amount of computer memory where it stores the value associated with the variable x and in a different portion of memory it stores the value associated with the variable y.

In line 5 the value of x is changed to 9, i.e., we have used the assignment operator to assign a new value to x. Then, in line 6, we again print the values of x and y. Here we see that y does not change! So, again, the statement in line 2 does not establish that x and y are equal. Rather, it assigns the current value of x to the variable y. A subsequent change to x or y does not affect the other variable.

Let us consider one other common idiom in computer languages which illustrates that the assignment operator is different from mathematical equality. In many applications we want to change the value of a variable from its current value but the change incorporates information about the current value. Perhaps the most common instance of this is incrementing or decrementing a variable by 1 (we might do this with a variable we are using as a counter). The code in Listing 2.6 demonstrates how one would typically increment a variable by 1. Note that in the interactive environment, if we enter a variable on a line and simply hit return, the value of the variable is echoed back to us. (Python evaluates the expression on a line and shows us the resulting value. When they appear to the right of the assignment operator, variables simply evaluate to their associated value.)

Listing 2.6 Demonstration of incrementing a variable.

```
>>> x = 10        # Assign a value to x.
>>> x             # Check the value of x.
10
>>> x = x + 1     # Increment x.
>>> x             # Check that x was incremented.
11
```

Line 4 of Listing 2.6 makes no sense mathematically: there is no value of x that satisfies the expression if we interpret the equal sign as establishing equality of the left and right sides. However, line 4 makes perfect sense in a computer language. Here we are telling Python to evaluate the expression on the right. Since x is initially 10, the right hand side evaluates to 11. This value is then assigned back to x—this becomes the new value associated with x. Lines 5 and 6 show us that x was successfully incremented.

In some computer languages, when a variable is created, its type is fixed for the duration of the variable's life. In Python, however, a variable's type is determined by the type of the value that has been most recently assigned to the variable. The code in Listing 2.7 illustrates this behavior.

Listing 2.7 A variable's type is determined by the value that is last assigned to the variable.

```
1  >>> x = 7 * 3 * 2
2  >>> y = "is the answer to the ultimate question of life"
3  >>> print(x, y)          # Check what x and y are.
4  42 is the answer to the ultimate question of life
5  >>> x, y                 # Quicker way to check x and y.
6  (42, 'is the answer to the ultimate question of life')
7  >>> type(x), type(y)     # Check types of x and y.
8  (<class 'int'>, <class 'str'>)
9  >>> # Set x and y to new values.
10 >>> x = x + 3.14159
11 >>> y = 1232121321312312312312 * 9873423789237438297
12 >>> print(x, y)          # Check what x and y are.
13 45.14159 12165255965071649871208683630735493412664
14 >>> type(x), type(y)     # Check types of x and y.
15 (<class 'float'>, <class 'int'>)
```

The first two lines of Listing 2.7 set variables `x` and `y` to an integer and a string, respectively. We can see this implicitly from the output of the `print()` statement in line 3 (but keep in mind that just because the output from a `print()` statement appears to be a numeric quantity doesn't mean the value associated with that output necessarily has a numeric type).

The expression in line 5 is something new. This shows that if we separate multiple variables (or expressions) with commas on a line and hit return, the interactive environment will show us the values of these variables (or expressions). This output is slightly different from the output produced by a `print()` statement in that the values are enclosed in parentheses and strings are shown enclosed in quotes.[10] When working with the interactive environment, you may want to keep this in mind for when you want to quickly check the values of variables (and, as we shall see, other things).

In line 7 of Listing 2.7 we use the `type()` function to explicitly show the types of `x` and `y`. In lines 10 and 11 we set `x` and `y` to a `float` and an `int`, respectively. Note that in Python an `int` can have an arbitrary number of digits, limited only by the memory of your computer.

Finally, before concluding this section, we want to mention that we will describe the relationship between a variable and its associated value in various ways. We may say that a variable points to a value, or a variable has a value, or a variable is equal to a value, or simply a variable is a value. As examples, we might say "`x` points to `7`" or "`x` has a value of `7`" or "`x` is `7`". We consider all these statements to be equivalent and discuss this further in Sec. 2.7.

[10] The output produced here is actually a collection of data known as a *tuple*. We will discuss tuples in Sec. 6.6.

2.4 Cascaded and Simultaneous Assignment

There are a couple of variations on the use of the assignment operator that are worth noting. First, it is possible to assign the same value to multiple variables using *cascaded* assignments. Listing 2.8 provides an example. In line 1 the expression on the right side is evaluated. The value of this expression, `11`, is assigned to `x`, then to `y`, and then to `z`. We can cascade any number of assignments but, other than the operand on the extreme right, all the other operands must be lvalues/variables. However, keep in mind that, as discussed in Sec. 2.3, even though `x`, `y`, and `z` are set to the same value, we can subsequently change the value of any one of these variables, and it will not affect the values of the others.

Listing 2.8 An example of cascading assignment in which the variables `x`, `y`, and `z` are set to the same value with a single statement.

```
1 >>> z = y = x = 2 + 7 + 2
2 >>> x, y, z
3 (11, 11, 11)
```

A cascading assignment is a construct that exists in many other computer languages. It is a useful shorthand (allowing one to collapse multiple statements into one), but typically doesn't change the way we think about implementing an algorithm. Cascaded assignments can make the code somewhat harder to read and thus should be used sparingly.

Python provides another type of assignment, known as simultaneous assignment, that is not common in other languages. Unlike cascaded assignments, the ability to do simultaneous assignment *can* change the way we implement an algorithm. With simultaneous assignment we have multiple expressions to the right of the assignment operator (equal sign) and multiple lvalues (variables) to the left of the assignment operator. The expressions and lvalues are separated by commas. For simultaneous assignment to succeed, there must be as many lvalues to the left as there are comma-separated expressions to the right.[11]

Listing 2.9 provides an example in which initially the variable `c` points to a string that is considered the "current" password and the variable `o` points to a string that is assumed to be the "old" password. The programmer wants to exchange the current and old passwords. This is accomplished using simultaneous assignment in line 5.

Listing 2.9 An example of simultaneous assignment where the values of two variables are exchanged with the single statement in line 5.

```
1 >>> c = "deepSecret"           # Set current password.
2 >>> o = "you'll never guess"  # Set old password.
3 >>> c, o                       # See what passwords are.
4 ('deepSecret', "you'll never guess")
5 >>> c, o = o, c                # Exchange the passwords.
```

[11]It is also possible to have a single lvalue to the left of the assignment operator and multiple expressions to the right. This, however, is not strictly simultaneous assignment. Rather, it is a regular assignment where a single tuple, which contains the multiple values, is assigned to the variable. We will return to this in Chap. 6.

```
6 >>> c, o                    # See what passwords are now.
7 ("you'll never guess", 'deepSecret')
```

The output generated by the statement in line 6 shows that the values of the current and old passwords have been swapped. There is something else in this code that is worth noting. The string in line 2 includes a single quote. This is acceptable provided the entire string is surrounded by something other than a single quote (e.g., double quotes or another quoting option considered later).

To accomplish a similar swap in a language without simultaneous assignment, we have to introduce a "temporary variable" to hold one of the values while we overwrite one of the variables. Listing 2.10 demonstrates how this swap is accomplished without the use of simultaneous assignment.

Listing 2.10 Demonstration of the swapping of two variables without simultaneous assignment. A temporary variable (here called `t`) must be used.

```
1 >>> c = "deepSecret"        # Set current password.
2 >>> o = "you'll never guess" # Set old password.
3 >>> c, o                    # See what passwords are.
4 ('deepSecret', "you'll never guess")
5 >>> t = c                   # Assign current to temporary variable.
6 >>> c = o                   # Assign old to current.
7 >>> o = t                   # Assign temporary to old.
8 >>> c, o                    # See what passwords are now.
9 ("you'll never guess", 'deepSecret')
```

Clearly the ability to do simultaneous assignment simplifies the swapping of variables and, as we will see, there are many other situations where simultaneous assignment proves useful. But, one should be careful not to overuse simultaneous assignment. For example, one could use simultaneous assignment to set the values of `x` and `y` as shown in line 1 of Listing 2.11. Although this is valid code, it is difficult to read and it would be far better to write two separate assignment statements as is done in lines 6 and 7.

Listing 2.11 Code that uses simultaneous assignment to set the value of two variables. Although this is valid code, there is no algorithmic reason to implement things this way. The assignment in line 1 should be broken into two separate assignment statements as is done in lines 6 and 7.

```
1 >>> # A bad use of simultaneous assignment.
2 >>> x, y = (45 + 34) / (21 - 4), 56 * 57 * 58 * 59
3 >>> x, y
4 (4.647058823529412, 10923024)
5 >>> # A better way to set the values of x and y.
6 >>> x = (45 + 34) / (21 - 4)
7 >>> y = 56 * 57 * 58 * 59
8 >>> x, y
9 (4.647058823529412, 10923024)
```

2.5 Multi-Line Statements and Multi-Line Strings

The end of a line usually indicates the end of a statement. However, statements can span multiple lines if we *escape* the newline character at the end of the line or if we use enclosing punctuation marks, such as parentheses, to span multiple lines. To escape a character is to change its usual meaning. This is accomplished by putting a forward slash (\) in front of the character. The usual meaning of a newline character is that a statement has ended.[12] If we put the forward slash at the end of the line, just before typing return on the keyboard, then we are saying the statement continues on the next line. This is demonstrated in Listing 2.12 together with the use of parentheses to construct multi-line statements.

Listing 2.12 Constructing multi-line statements by either escaping the newline character at the end of the line (as is done in line 1) or by using parentheses (as is done in lines 3 and 5).

```
>>> x = 3 + \
... 4
>>> y = (123213123123121212312 * 3242342423
... + 2343242332 + 67867687678 - 2
... + 6)
>>> x, y
(7, 399499136172418158920598441990)
```

Here we see, in lines 2, 4, and 5, that the prompt changes to triple dots when the interpreter is expecting more input.

The converse of having a statement span multiple lines is having multiple statements on a single line. The ability to do this was mentioned in Sec. 1.3 and demonstrated in Listing 1.5. One merely has to separate the statements with semicolons. However, for the sake of maximizing code readability, it is best to avoid putting multiple statements on a line.

Often we want to construct multi-line strings. This can be done in various ways but the simplest technique is to enclose the string in either a single or double quotation mark that is repeated three times.[13] This is illustrated in Listing 2.13.

Listing 2.13 Using triple quotation marks to construct multi-line strings.

```
>>> s = """This string
... spans
... multiple lines."""
>>> t = '''This
... sentence
... no
```

[12] Keep in mind that although we cannot see a newline character, there is a collection of bits at the end of a line that tells the computer to start subsequent output on a new line.

[13] When using triple quotation marks, the string doesn't have to span multiple lines, but since one can use a single set of quotation marks for a single-line string, there is typically no reason to use triple quotation marks for a single-line string.

```
7  ... verb.'''
8  >>> print(s)
9  This string
10 spans
11 multiple lines.
12 >>> print(t)
13 This
14 sentence
15 no
16 verb.
17 >>> s, t   # Quick check of what s and t are.
18 ('This string\nspans\nmultiple lines.', 'This\nsentence\nno\nverb.')
```

In lines 1 through 3 and lines 4 through 7 we create multi-line strings and assign them to variables `s` and `t`. The `print()` statements in lines 8 and 12 show us that `s` and `t` are indeed multi-line strings. In line 17 we ask Python to show us the values of `s` and `t`. The output on line 18 appears rather odd in that the strings do not span multiple lines. Instead, where a new line should start we see `\n`. `\n` is Python's way of indicating a newline character—the `n` is escaped so that it doesn't have its usual meaning. We will consider strings in much more detail in Chap. 9.[14]

2.6 Identifiers and Keywords

In addition to assigning names to values (or objects), there are other entities, such as functions and classes, for which programmers must provide a name. These names are known as *identifiers*. The rules that govern what is and isn't a valid identifier are the same whether we are providing a name for a function or variable or anything else. A valid identifier starts with a letter or an underscore and then is followed by any number of letters, underscores, and digits. No other characters are allowed. Identifiers are case sensitive so that, for example, `ii`, `iI`, `Ii`, and `II`, are all unique names.

Examples of some valid and invalid identifiers are shown in Listing 2.14.

Listing 2.14 Some valid and invalid identifiers.

`one2three`	Valid.
`one_2_three`	Valid.
`1_2_three`	Invalid. Cannot start with a digit.
`one-2-three`	Invalid. Cannot have hyphens (minus signs).
`n31`	Valid.
`n.31`	Invalid. Cannot have periods.
`trailing__`	Valid.
`__leading`	Valid.
`net worth`	Invalid. Cannot have blank spaces.
`vArIaBlE`	Valid.

[14] If you were paying close attention to the output presented in Listing 1.12, you would have noticed `\n` in it.

When choosing an identifier, it is best to try to provide a descriptive name that isn't too long; for example, `area` may be sufficiently descriptive while `area_of_the_circle` may prove cumbersome. There are some identifiers that are valid but, for the sake of clarity, are best avoided. Examples include `l` (lowercase "ell" which may look like one), `O` (uppercase "oh" which may look like zero), and `I` (uppercase "eye" which may also look like one).

It is also good practice to use short variable names consistently. For example, `s` is often used as a name for a string while `ch` is used for a string that consists of a single character (i.e., `ch` connotes character). The variables `i` and `j` are often used to represent integers while `x` and `y` are more likely to be used for `floats`.

In addition to the rules that govern identifiers, there are also *keywords* (sometimes called reserved words) that have special meaning and *cannot* be used as an identifier. All the keywords in Python 3 are listed in Listing 2.15.[15] Keywords will appear in a bold blue `Courier` font.

Listing 2.15 The 33 keywords in Python 3.

```
False    None     True     and      as       assert   break
class    continue def      del      elif     else     except
finally  for      from     global   if       import   in
is       lambda   nonlocal not      or       pass     raise
return   try      while    with     yield
```

At this point we don't know what these keywords represent. The important point, however, is that apart from these words, any identifier that obeys the rules mentioned above is valid.

Perhaps we should take a moment to further consider this list of keywords and think about them in the context of what we have learned so far. Thus far we have used two functions that Python provides: `print()` and `type()`. Notice that neither `print` nor `type` appears in the list of keywords. So, can we have a variable named `print`? The interactive interpreter provides an excellent means to answer this question. Listing 2.16 shows what happens when a value is assigned to the identifier `print`. The statement in line 1 assigns the integer `10` to `print` and we see, in lines 2 and 3, that this assignment works! There is no error. Now we have to wonder: What happened to the `print()` function? In line 4 an attempt is made to print `Hello!` The output following line 4 shows this is not successful—we have lost our ability to print since Python now thinks `print` is the integer `10`!

Listing 2.16 Demonstration of what happens when we assign a value to a built-in Python function.

```
1 >>> print = 10        # Assign 10 to print.
2 >>> print             # Check what print is.  Assignment worked!
3 10
4 >>> print("Hello!")   # Try to print something...
5 Traceback (most recent call last):
```

[15]You can produce this list from within Python by issuing these two commands: `import keyword; print(keyword.kwlist)`.

```
6    File "<stdin>", line 1, in <module>
7  TypeError: 'int' object is not callable
```

Variables spring into existence when we assign them a value. If an identifier had previously been used, the old association is forgotten when a new value is assigned to it. In practice this is rarely a problem. There are very few built-in Python functions that you would consider using as an identifier. There is one possible exception to this that we will discuss in more detail in Chap. 6.[16]

For now, the use of leading underscores is discouraged as Python uses leading underscores for some internal identifiers. Also identifiers with leading underscores are treated slightly differently than other identifiers. However, underscores within an identifier, such as net_worth, are quite common and are encouraged.

Finally, there is one final identifier that is worth noting. In the interactive environment (and only in the interactive environment), a single underscore (_) is equal to the value of the most recently evaluated expression. Although we will not use this functionality in this book, it does prove useful when using the interactive environment as a calculator. Listing 2.17 briefly illustrates this.

Listing 2.17 Using "_" to recall the value of previously evaluated expressions.

```
1  >>> 5 + 4        # Enter an expression.
2  9
3  >>> _            # See that _ is equal to value of last expression.
4  9
5  >>> _ + 27       # Use _ in new expression.
6  36
7  >>> print(_)     # See that _ has been reset.
8  36
9  >>> _ - 15
10 21
11 >>> result = _   # Store value of _ to a variable.
12 >>> result
13 21
```

2.7 Names and Namespaces

Let us now delve deeper into what happens when a value is assigned to a variable. A variable has a name, i.e., an identifier. The assignment serves to associate the value with the name. This value really exists in the form of an *object*. (We aren't concerned here with the details of objects,[17] so we

[16]If you want to learn more now, there is a built-in function called list(). Lists are commonly used in Python and it is tempting to use list as an identifier of your own. You can do this, but you will then lose easy access to the built-in function list(). However, one further aside: If you ever mask one of the built-in functions by assigning a value to the function's identifier, you can recover the built-in function by deleting, using the del() function, your version of the identifier. Returning to Listing 2.16 as an example, to recover the built-in print() function you would issue the statement del(print). Note that del is a keyword and hence you cannot accidently assign it a new value.

[17]Objects are discussed in Chap. 5.

can simply think of an object as a "thing" that has a value and we are only interested in this value. For example, although the integer 7 is really treated by Python as a particular kind of object, here we consider this to be simply the integer 7.)

Within your computer's memory, Python maintains what are known as *namespaces*. A namespace is used to keep track of all the currently defined names (identifiers) as well as the objects associated with these names. If, within an expression, you refer to a name that was *not* previously defined, an exception will be raised (i.e., an error will occur). However, if a previously undefined name is used on the *left* side of the assignment operator, then the name is created within the namespace and it is associated with the value produced by the expression to the right of the assignment operator.

The behavior of namespaces is illustrated in Fig. 2.1. The namespace is shown to the right of the figure and the code that is assumed to be executed is shown to the left. In the figure, the namespace is depicted as consisting of two parts: a list of names and a list of objects (where only the values of the objects are shown). The namespace serves to map names to objects. In Fig. 2.1(a), the statement `x = 7` is executed. The name to the left side of the assignment operator, `x`, is placed in the list of names and is associated with the value `7`. This association is indicated by the curved line. We can think of this as "`x` points to `7`." In Fig. 2.1(b), the statement `x = 19` is executed. (Here we are assuming this statement is executed in the same session in which the statement `x = 7` was previously executed.) Since the name `x` already exists in the list of names, no new name is created. Instead, an object with a value of `19` is created and `x` is associated with this new value. You may ask: What happens to the value `7`? The fate of that object is up to Python and not our concern. In the figure, rather than removing the object from the list of objects, we show that it persists but no name is associated with it (i.e., nothing points to it and hence it is shown in gray rather than black).

Finally, in Fig. 2.1(c), the statement `x = x / 2` is executed. Recall that in assignment statements the expression to the right of the assignment operator is evaluated and then this value is associated with the name to the left of the assignment operator. Here `x` appears in the expression to the right of the assignment operator. This does not cause a problem since `x` has previously been defined. When this expression is evaluated, the value associate with `x` is `19`. The result of `19 / 2` is the `float` value `9.5`, hence after this third statement is executed, `x` is associated with the `float` `9.5`.

As another example of the behavior of namespaces, consider Fig. 2.2 which involves two variables, `x` and `y`. Listing 2.2(a) is no different from Listing 2.1(a): the name `x` is assigned the value `7`. Figure 2.1(b) depicts what happens when the statement `y = x` is executed. From your math classes, it would be natural to think this statement would somehow associate `y` directly with `x`. But, this is *not* the case. Instead, as always, the right hand side is evaluated and the resulting value is associated with the name `y`. In this case, the right hand side simply evaluates to `7` and `y` is associated with that. Since `7` is already in the list of objects, Python does not create a new object but rather has `y` point to the same object as `x`.

Figure 2.1(c) depicts what happens when the statement `x = 19` is executed. Python does not change the value of an integer object, i.e., it will not change the value in memory from `7` to `19`. Instead, it creates a new object and associates `x` with it.

As a final example, Fig. 2.3 depicts the behavior of a namespace when code involving simultaneous assignment is executed. In Fig. 2.3(a), `x` and `y` are simultaneously assigned values: `x` is assigned the integer `2` while `y` is assigned the string `two`. In Fig. 2.3(b), `x` is assigned a new

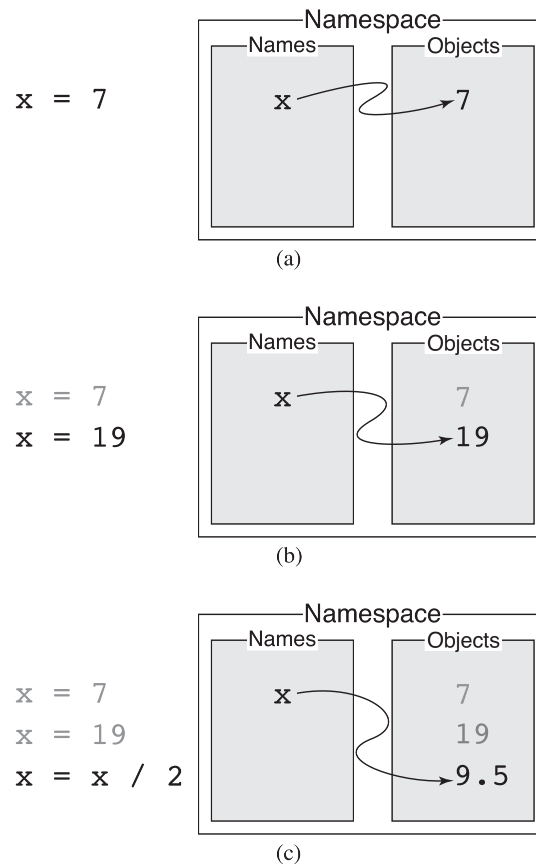


Figure 2.1: Depiction of a namespace which consists of names (i.e., identifiers or variables) and a collection of objects (where only the value of the object is shown). (a) Contents of the namespace after the statement `x = 7` is executed. (b) Contents of the namespace after executing `x = 19`. (c) Contents of the namespace after executing `x = x / 2`.

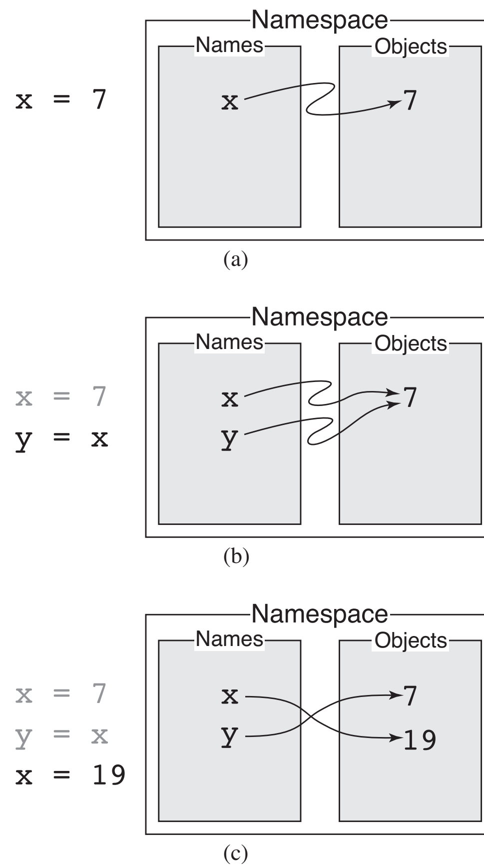


Figure 2.2: Depiction of another namespace (i.e., independent of the namespace shown in Fig. 2.1). (a) Contents of the namespace after the statement `x = 7` is executed. (b) Contents of the namespace after executing `y = x`. (c) Contents of the namespace after executing `x = 19`.

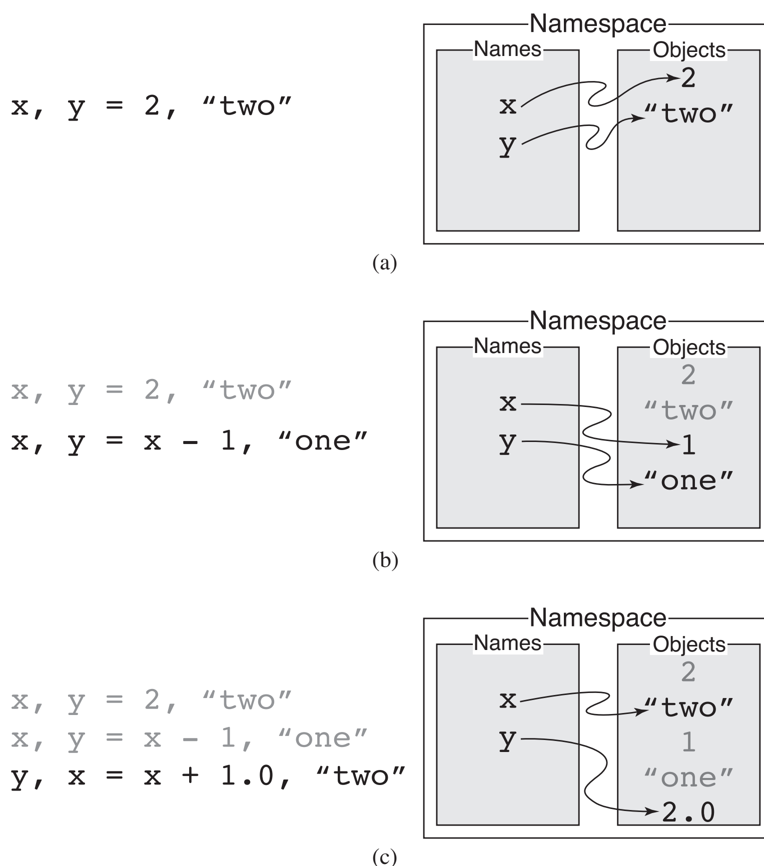


Figure 2.3: Depiction of another namespace. The code that is executed involves simultaneous assignment to the names `x` and `y`. (a) `x` and `y` are initialized to an integer and a string, respectively. (b) `x` and `y` are assigned different integer and string values. (c) `y` is set to a `float` (based on the current value of `x` and `x` is assigned a `string`.

integer value (that is obtained by decrementing its initial value by one) and `y` is assigned the string `one`. In Fig. 2.3(c), `y` is assigned the `float 2.0` (which is obtained by adding `1.0` to the current value of `x`) and `x` is assigned the string `two`. Since the `float 2.0` is distinct from the integer `2`, in Fig. 2.3(c) the value `2.0` is added to the list of objects.

Again we note that, as Figs. 2.1–2.3 illustrate, in Python a name can, at different points in a program, be associated with values of different type. Thus, a given name may be associated with an integer in one section of the code, with a `float` in a different section of the code, and a string in yet another section. This “freedom” to easily associate names with different types of values is *not* possible in many other computer languages. For example, in C a variable can typically only be associated with values of a fixed type (the value itself can change, but its type cannot). This ability in Python is generally a blessing but can be something of a curse. It allows the programmer to express things in a few lines of code that might otherwise take many more. However, it can be a source of bugs when the underlying type isn't what the programmer assumes.

2.8 Additional Arithmetic Operators

There are other arithmetic operators in Python that are somewhat less common than those discussed in Sec. 2.2, but are nevertheless quite important in the implementation of many algorithms.

2.8.1 Exponentiation

Exponentiation is obtained with the operator `**`. The expression `x ** y` says to calculate x^y. This operator is unlike the other arithmetic operators in that it associates (or evaluates) right to left. The expression `x ** y ** z` is equivalent to $x^{y^z} = x^{(y^z)}$ and *not* $(x^y)^z$. If both operands are `int`s, the result is an `int`. If either operand is a `float`, the result is a `float`. Listing 2.18 demonstrates the use of exponentiation.

Listing 2.18 Demonstration of the `**` (exponentiation) operator.

```
1  >>> 2 ** 3
2  8
3  >>> 2 ** 3 ** 4     # Operator is right associative.
4  2417851639229258349412352
5  >>> 3 ** 4
6  81
7  >>> (2 ** 3) ** 4  # Use parentheses to change order of operation.
8  4096
9  >>> 3 ** 400         # A big number.  Exact value as an int.
10 70550791086553325712464271575934796216507949612787315762871223209262085
11 55158293415657929852944713415815495233482535591186692979307182456669414
12 5084454535257027960285323760313192443283334088001
13 >>> 3.0 ** 400      # A big number.  Approximate value as a float.
14 7.055079108655333e+190
```

In line 1 we calculate 2^3 and in line 3 we calculate $2^{3^4} = 2^{81}$. In line 9 3^{400} is calculated using integers. The result shown on the next three lines is the exact result; this number is actually returned as one continuous number (i.e., without explicit line breaks) but here the number has been split into three lines for the sake of readability. Lines 13 and 14 show what happens when one of the arguments is a `float`. The result in line 14 is an approximation to the exact answer. The `float` result has about 16 digits of precision whereas the exact result has 191 digits.

2.8.2 Floor Division

Recall from Sec. 2.2 that `float` division is represented by the operator `/`. (If one simply says "division," `float` division is implied.) Python provides another type of division which is known as floor division. Many computer languages provide two types of division and Python's floor division is similar to what is known as integer division in most other languages. In most other languages, both integer and `float` division share the same symbol, i.e., `/`. The type of division that is done depends on the types of the two operands involved. This can be confusing and often

leads to bugs! Python is different in that the operator / *always* represents `float` division while `//` is *always* used for floor division. In floor division the quotient is a whole number. Specifically, the result is the greatest whole number that does not exceed the value that would be obtained with float division. For example, if the value obtained from `float` division was `3.2`, the result from floor division would be either the integer `3` or the `float 3.0` since three is the largest whole number that doesn't exceed `3.2`. The result is an `int` if both operands are integers but is a `float` if either or both operands are `floats`. As another example, if the value obtained from `float` division was `-3.2`, the result from floor division would be either the integer `-4` or the `float -4.0` since negative four is the largest whole number that doesn't exceed `-3.2`.

Floor division may not seem to be a useful tool in that it discards information (the fractional part). However, in many problems it only makes sense to talk about whole units of a quantity. Thus, discarding the fractional part is appropriate. For example, what if we want to know how many complete minutes there are in 257 seconds or how many quarters there are in 143 pennies? The first six lines of Listing 2.19 illustrate how floor division answers these questions.

Listing 2.19 Demonstration of floor division.

```
1  >>> time = 257              # Time in seconds.
2  >>> minutes = time // 60 # Number of complete minutes in time.
3  >>> print("There are", minutes, "complete minutes in", time, "seconds.")
4  There are 4 complete minutes in 257 seconds.
5  >>> 143 // 25
6  5
7  >>> 143.4 // 25
8  5.0
9  >>> 9 // 2.5
10 3.0
```

Lines 7 through 10 show that `floats` can be used with floor division. The result is a `float` but with a fractional part of zero (i.e., a whole number represented as a `float`).

Floor division has the same precedence as multiplication and `float` division.

2.8.3 Modulo and `divmod()`

Another arithmetic operator that you may not have encountered before is the *modulo*, or sometimes called *remainder*, operator. In some sense, modulo is the complement of floor division. It tells us what remains after the left operand is divided by the right operand an integer number of times. The symbol for modulo is `%`. An expression such as `17 % 5` is read as seventeen mod five (this expression evaluates to `2` since `5` goes into `17` three times with a remainder of `2`). Although modulo may be new to you, it is extremely useful in a wide range of algorithms. To give a hint of this, assume we want to express 257 seconds in terms of minutes and seconds. From Listing 2.19 we already know how to obtain the number of minutes using floor division. How do we obtain the number of seconds? If we have the number of minutes, we can calculate the remaining number of seconds as line 3 of Listing 2.20 shows.

Listing 2.20 Slightly cumbersome way to find the number of minutes and seconds in a given time (which is assumed to be in seconds).

```
1 >>> time = 257                     # Initialize total number of seconds.
2 >>> minutes = time // 60           # Calculate minutes.
3 >>> seconds = time - minutes * 60 # Calculate seconds.
4 >>> minutes, seconds               # Show minutes and seconds.
5 (4, 17)
```

In Listing 2.20 we first calculate the minutes and then, using that result, calculate the number of seconds. However, what if we just wanted the number of seconds? We can obtain that directly using the modulo operator, i.e., we can replace the statement in line 3 with `seconds = time % 60`.

Modulo has the same operator precedence as division and multiplication.

If we want to calculate both floor division and modulo with the same two operands, we can use the function `divmod()`. This function returns the results of both floor division and modulo. Thus,

```
divmod(x, y)
```

is equivalent to

```
x // y, x % y
```

We can use simultaneous assignment to assign the two results to two variables. Listing 2.21 demonstrates this.

Listing 2.21 Using the `divmod()` function to calculate both floor division and modulo simultaneously.

```
1 >>> time = 257         # Initialize time.
2 >>> SEC_PER_MIN = 60 # Use a "named constant" for 60.
3 >>> divmod(time, SEC_PER_MIN)   # See what divmod() returns.
4 (4, 17)
5 >>> # Use simulataneous assignment to obtain minutes and seconds.
6 >>> minutes, seconds = divmod(time, SEC_PER_MIN)
7 >>> # Attempt to display the minutes and seconds in "standard" form.
8 >>> print(minutes, ":", seconds)
9 4 : 17
10 >>> # Successful attempt to display time "standard" form.
11 >>> print(minutes, ":", seconds, sep="")
12 4:17
13 >>> # Obtain number of quarters and leftover change in 143 pennies.
14 >>> quarters, cents = divmod(143, 25)
15 >>> quarters, cents
16 (5, 18)
```

In addition to illustrating the use of the `divmod()` function, there are two other items to note in Listing 2.21. First, in Listings 2.19 and 2.20 the number `60` appears in multiple expressions. We know that in this context `60` represents the number of seconds in a minute. But, the number `60` by itself has very little meaning (other than the whole number that comes after 59). `60` could represent any number of things: the age of your grandmother, the number of degrees in a particular angle, the weight in ounces of your favorite squirrel, etc. When a number appears in a program without its meaning being fully specified, it is known as a "magic number." There are a few different definitions for magic numbers. The one relevant to this discussion is: "Unique values with unexplained meaning or multiple occurrences which could (preferably) be replaced with named constants."[18] In line 2 of Listing 2.21 we create a *named constant*, `SEC_PER_MIN`, as a substitute for the number `60` in the subsequent code. This use of named constants can greatly enhance the readability of a program. It is a common practice for a named constant identifier to use uppercase letters.

The other item to note in Listing 2.21 pertains to the `print()` statements in lines 8 and 11. In line 8 the `print()` statement has three arguments: the number of minutes, a string (corresponding to a colon), and the number of seconds. When displaying a time, we often separate the number of minutes and seconds with a colon, e.g., `4:17`. However, the output on line 9 isn't quite right. There are extraneous spaces surrounding the colon. The `print()` statement on line 11 fixes this by setting the optional parameter `sep` to an empty string. If you refer to the output shown in Listing 1.12, you will see that, by default, `sep`, which is the separator that appears between the values that are printed, has a value of one blank space. By setting `sep` to an empty string we get the desired output shown in line 12.

As with floor division, if both operands are integers, modulo returns an integer. If either or both operands are a `float`, the result is a `float`. This behavior holds true of the arguments of the `divmod()` function as well: both arguments must be integers for the return values to be integers.

2.8.4 Augmented Assignment

As mentioned in Sec. 2.3, it is quite common to assign a new value to a variable that is based on the old value of that variable, e.g., `x = x + 1`. In fact, this type of operation is so common that many languages, including Python, provide arithmetic operators that serve as a shorthand for this. These are known as *augmented assignment operators.* These operators use a compound symbol consisting of one of the "usual" arithmetic operators together with the assignment operator. For example, `+=` is the addition augmented assignment operator. Other examples include `-=` and `*=` (which are the subtraction augmented assignment operator and the multiplication augmented assignment operator, respectively).

To help explain how these operators behave, let's write a general augmented assignment operator as *<op>*= where *<op>* is a placeholder for one of the usual arithmetic operators such `+`, `-`, or `*`. A general statement using an augmented assignment operator can be written as

```
<lvalue> <op>= <expression>
```

[18] `http://en.wikipedia.org/wiki/Magic_number_(programming)`

where <lvalue> is a variable (i.e., an lvalue), <op>= is an augmented assignment operator, and <expression> is an expression. The following is a specific example that fits this general form

```
x += 1
```

Here x is the lvalue, += is the augmented assignment operator, and the literal 1 is the expression (albeit a very simple one—literals are the simplest form of an expression). This statement is completely equivalent to

```
x = x + 1
```

Now, returning to the general expression, an equivalent statement in non-augmented form is

```
<lvalue> = <current_value_of_lvalue> <op> (<expression>)
```

where <current_value_of_lvalue> is, naturally, the value of the lvalue prior to the assignment. The parentheses have been added to <expression> to emphasize the fact that this expression is completely evaluated before the operator associated with the augmented assignment comes into play.

Let's consider a couple of other examples. The following statement

```
y -= 2 * 7
```

is equivalent to

```
y = y - (2 * 7)
```

while

```
z *= 2 + 5
```

is equivalent to

```
z = z * (2 + 5)
```

Note that if we did not include the parentheses in this last statement, it would not be equivalent to the previous one.

There are augmented assignment versions of all the arithmetic operators we have considered so far. Augmented assignment is further illustrated in the code in Listing 2.22.

Listing 2.22 Demonstration of the use of augmented assignment operators.

```
1 >>> x = 22        # Initialize x to 22.
2 >>> x += 7        # Equivalent to: x = x + 7
3 >>> x
4 29
5 >>> x -= 2 * 7    # Equivalent to: x = x - (2 * 7)
6 >>> x
7 15
8 >>> x //= 5       # Equivalent to: x = x // 5
9 >>> x
```

```
10 3
11 >>> x *= 100 + 20 + 9 // 3 # Equivalent to: x = x * (100 + 20 + 9 // 3)
12 >>> x
13 369
14 >>> x /= 9         # Equivalent to: x = x / 9
15 >>> x
16 41.0
```

2.9 Chapter Summary

Literals are data that are entered directly into the code.

Data has *type*, for example, **`int`** (integer), **`float`** (real number with finite precision), and **`str`** (string, a collection of characters).

`type()` returns the type of its argument.

A literal `float` has a decimal point or contains the power-of-ten exponent indicator `e`, e.g., `1.1e2`, `11e1`, and `110.0` are equivalent `floats`.

A literal `int` does not contain a decimal point (nor the power-of-ten exponent indicator `e`).

A literal `str` is enclosed in either a matching pair of double or single quotes. Quotation marks can be repeated three times at the beginning and end of a string, in which case the string can span multiple lines.

`\n` is used in a string to indicate the newline character.

Characters in a string may be *escaped* by preceding the character with a backslash. This causes the character to have a different meaning than usual, e.g., a backslash can be placed before a quotation mark in a string to prevent it from indicating the termination of the string; the quotation mark is then treated as part of the string.

An *expression* produces data. The simplest expression is a literal.

There are numerous arithmetic operators. *Binary operators* (which require two operands) include:

`+`, `-`	⇔	addition and subtraction
`/`, `//`	⇔	`float` and floor division
`%`	⇔	modulo (or remainder)
`*`	⇔	multiplication
`**`	⇔	exponentiation

`float` division (`/`) yields a `float` regardless of the types of the operands. For all other arithmetic operators, only when *both* operands are integers does the operation yield an integer. Said another way, when either or both operands are `floats`, the arithmetic operation yields a `float`.

Floor division (`//`) yields a whole number (which may be either an `int` or a `float`, depending on the operands). The result is the largest whole number that does not exceed the value that would be obtained with `float` division.

Modulo (`%`) yields the remainder (which may be either an `int` or a `float`, depending on the operands) after floor division has been performed.

divmod(a, b): equivalent to ⇒ `a // b, a % b`.

Evaluation of expressions containing multiple operations follows the rules of *precedence* to determine the order of operation. Exponentiation has highest precedence; multiplication, integer and `float` division, and modulo have equal precedence which is below that of exponentiation; addition and subtraction have equal precedence which is below that of multiplication, division, and modulo.

Operations of equal precedence are evaluated left to right except for exponentiation operations which are evaluated right to left.

The negative sign (−) and positive sign (+) can be used as *unary operators*, e.g., `-x` changes the sign of `x`. The expression `+x` is valid but has no effect on the value of `x`.

Parentheses can be used to change the order or precedence.

Statements are complete commands.

The equal sign = is the *assignment operator*. The value of the expression on the right side of the equal sign is assigned to the *lvalue* on the left side of the equal sign.

An *lvalue* is a general name for something that can appear to the left side of the assignment operator. It is typically a variable that must be a *valid identifier*.

A variable can also be thought of as a *name* within a *namespace*. A namespace maps names to their corresponding values (or objects).

Valid identifiers start with a letter or underscore followed by any number of letters, digits, and underscores.

There are 33 *keywords* that cannot be used as identifiers.

Augmented operators can be used as shorthand for assignment statements in which an identifier appears in an arithmetic operation on the left side of the equal sign *and* on the the right side of the assignment operator. For example, `x += 1` is equivalent to `x = x + 1`.

Simultaneous assignment occurs when multiple comma-separated expressions appear to the right side of an equal sign and an equal number of comma-separated lvalues appear to the left side of the equal sign.

A statement can span *multiple lines* if it is enclosed in parentheses or if the newline character at the end of each line of the statement (other than the last) is escaped using a backslash.

Multiple statements can appear on one line if they are separated by semicolons.

A *magic number* is a numeric literal whose underlying meaning is difficult to understand from the code itself. *Named constants* should be used in the place of magic numbers.

2.10 Review Questions

1. What does Python print as a result of this statement:

```
print(7 + 23)
```

   (a) `7 + 23`

   (b) `7 + 23 = 30`

   (c) 30
   (d) This produces an error.

2. Which of the following *are* valid variable names?
   (a) _1_2_3_
   (b) ms.NET
   (c) WoW
   (d) green-day
   (e) big!fish
   (f) 500_days_of_summer

3. Which of the following *are* valid identifiers?
   (a) a1b2c
   (b) 1a2b3
   (c) a_b_c
   (d) _a_b_
   (e) a-b-c
   (f) -a-b-
   (g) aBcDe
   (h) a.b.c

4. Suppose the variable x has the value 5 and y has the value 10. After executing these statements:

```
x = y
y = x
```

   what will the values of x and y be, respectively?
   (a) 5 and 10
   (b) 10 and 5
   (c) 10 and 10
   (d) 5 and 5

5. True or False: "x ** 2" yields the identical result that is produced by "x * x" for all integer and float values of x.

6. True or False: "x ** 2.0" yields the identical result that is produced by "x * x" for all integer and float values of x.

7. What does Python print as a result of this statement?

```
print(5 + 6 % 7)
```

(a) This produces an error.

(b) `5 + 6 % 7`

(c) `11`

(d) `4`

8. What is the output from the `print()` statement in the following code?

```
x = 3 % 4 + 1
y = 4 % 3 + 1
x, y = x, y
print(x, y)
```

(a) `2 4`

(b) `4 2`

(c) `0 3`

(d) `3 0`

9. What is the output produced by the following code?

```
x = 3
y = 4
print("x", "y", x + y)
```

(a) `3 4 7`

(b) `x y 7`

(c) `x y x + y`

(d) `3 4 x + y`

(e) `x y 34`

For each of the following, determine the value to which the expression evaluates. (Your answer should distinguish between `floats` and `ints` by either the inclusion or exclusion of a decimal point.)

10. `5.5 - 11 / 2`

11. `5.5 - 11 // 2`

12. `10 % 7`

13. `7 % 10`

14. `3 + 2 * 2`

15. `16 / 4 / 2`

16. `16 / 4 * 2`

17. Given that the following Python statements are executed:

```
a = 2
b = a + 1 // 2
c = a + 1.0 // 2
d = (a + 1) // 2
e = (a + 1.0) // 2
f = a + 1 / 2
g = (a + 1) / 2
```

Determine the values to which the variables `b` through `g` are set.

18. What output is produced when the following code is executed?

```
hello = "yo"
world = "dude"
print(hello, world)
```

(a) `hello, world`

(b) `yo dude`

(c) `"yo" "dude"`

(d) `yodude`

(e) This produces an error.

19. The following code is executed. What is the output from the `print()` statement?

```
x = 15
y = x
x = 20
print(y)
```

(a) `15`

(b) `20`

(c) `y`

(d) `x`

(e) This produces an error.

20. The following code is executed. What is the output from the `print()` statement?

```
result = "10" / 2
print(result)
```

    (a) `5`
    (b) `5.0`
    (c) `'"10" / 2'`
    (d) This produces an error.

21. The following code is executed. What is the output from the `print()` statement?

```
x = 10
y = 20
a, b = x + 1, y + 2
print(a, b)
```

    (a) `10 20`
    (b) `11 22`
    (c) `'a, b'`
    (d) `'x + 1, y + 2'`
    (e) This produces an error.

22. True or False: In general, “`x / y * z`” is equal to “`x / (y * z)`”.

23. True or False: In general, “`x / y ** z`” is equal to “`x / (y ** z)`”.

24. True or False: In general, “`w + x * y + z`” is equal to “`(w + x) * (y + z)`”.

25. True or False: In general, “`w % x + y % z`” is equal to “`(w % x) + (y % z)`”.

26. True or False: If both `m` and `n` are `int`s, then “`m / n`” and “`m // n`” both evaluate to `int`s.

27. True or False: The following three statements are all equivalent:

```
x = (3 +
4)

x = 3 + \
4

x = """3 +
4"""
```

28. Given that the following Python statements are executed:

```
x = 3 % 4
y = 4 % 3
```

What are the values of x and y?

29. To what value is the variable z set by the following code?

```
z = 13 + 13 // 10
```

(a) 14.3
(b) 14.0
(c) 2
(d) 14
(e) 16

30. Assume the float variable ss represents a time in terms of seconds. What is an appropriate statement to calculate the number of complete minutes in this time (and store the result as an int in the variable mm)?

(a) mm = ss // 60
(b) mm = ss / 60
(c) mm = ss % 60
(d) mm = ss * 60

31. To what values does the following statement set the variables x and y?

```
    x, y = divmod(13, 7)
```

(a) 6 and 1
(b) 1 and 6
(c) 6.0 and 2.0
(d) This produces an error.

ANSWERS: 1) c; 2) a and c are valid; 3) a, c, d, and g are valid; 4) c; 5) True; 6) False, if x is an integer x * x yields an integer while x ** 2.0 yields a float; 7) c; 8) b; 9) b; 10) 0.0; 11) 0.5; 12) 3; 13) 7; 14) 7; 15) 2.0; 16) 8.0; 17) b = 2, c = 2.0, d = 1, e = 1.0, f = 2.5, g = 1.5; 18) b; 19) a; 20) d; 21) b; 22) False (the first expression is equivalent to "x * z / y"); 23) True; 24) False; 25) True; 26) False ("m / n" evaluates to a float); 27) False (the first two are equivalent arithmetic expressions, but the third statement assigns a string to x); 28) 3 and 1; 29) d; 30) a; 31) b.

2.11 Exercises

1. Write a small program that assigns an angle in degrees to a variable called `degrees`. The program converts this angle to radians and assigns it to a variable called `radians`. To convert from degrees to radians, use the formula $\text{radians} = \text{degrees} \times 3.14/180$ (where we are using 3.14 to approximate π). Print the angle in both degrees and radians.

   The following demonstrates the program output when the angle is 150 degrees:

```
Degrees:  150
Radians:  2.616666666666667
```

2. Write a program that calculates the average score on an exam. Assume we have a small class of only three students. Assign each student's score to variables called `student1`, `student2`, and `student3` and then use these variables to find the average score. Assign the average to a variable called `average`. Print the student scores and the average score.

   The following demonstrates the program output when the students have been assigned scores of 80.0, 90.0, and 66.5:

```
Student scores:
80.0
90.0
66.5
Average:  78.83333333333333
```

3. Imagine that you teach three classes. These classes have 32, 45, and 51 students. You want to divide the students in these classes into groups with the same number of students in each group but you recognize that there may be some "left over" students. Assume that you would like there to be 5 groups in the first class (of 32 students), 7 groups in the second class (of 45 students), and 6 groups in the third class (of 51 students). Write a program that uses the `divmod()` function to calculate the number of students in each group (where each group has the same number of students). Print this number for each class and also print the number of students that will be "leftover" (i.e., the number of students short of a full group). Use simultaneous assignment to assign the number in each group and the "leftover" to variables.

   The following demonstrates the program's output:

```
Number of students in each group:
Class 1:  6
Class 2:  6
Class 3:  8
Number of students leftover:
Class 1:  2
Class 2:  3
Class 3:  3
```

4. The Python statements below have several errors. Identify the errors and correct them so that the program properly calculates the circumference of Jimmy's pie ($\text{circumference} = 2\pi r$).

```
pi = '3.14'
pie.diameter = 55.4
pie_radius = pie.diameter // 2
circumference = 2 * pi ** pie_radius
circumference-msg = 'Jimmy's pie has a circumference: '
print(circumference-msg, circumference)
```

The following demonstrates the output from the corrected program:

```
Jimmy's pie has a circumference:  173.956
```

5. Write a program that calculates the wavelength of a wave traveling at a constant velocity given the speed and the frequency. Use the formula $\lambda = v/f$, where λ (lambda) is wavelength in meters, v is velocity in meters per second, and f is frequency in Hertz (cycles per second). Print the velocity, frequency, and wavelength. Assign each of these values to a variable and use the variables in your `print()` statements.

The following demonstrates what the program prints:

```
The speed (m/s):  343
The frequency (Hz):  256
The wavelngth (m):  1.33984375
```

Chapter 3

Input and Type Conversion

It is hard to imagine a useful program that doesn't produce some form of output. From the last two chapters we know how to generate text output in Python using `print()`,[1] but the output produced by a program doesn't have to be text. Instead, it can be an image sent to your screen or written to a file; it can be an audio signal sent to your speakers; and it can even be data generated solely to pass along to another program (without needing to be sent to your screen or a file).

In general, programs not only generate output, they also work with input. This input may come from the user via a keyboard or mouse, from a file, or from some other source such as another program or a network connection. In this chapter we implement programs that perform both *input and output* (often abbreviated as I/O) but restrict ourselves to obtaining input only from a keyboard.

3.1 Obtaining Input: `input()`

The built-in function `input()` takes a string argument. This string is used as a prompt. When the `input()` function is called, the prompt is printed and the program waits for the user to provide input via the keyboard. The user's input is sent back to the program once the user types return. `input()` *always returns the user's input as a string*. This is true even if we are interested in obtaining a numeric quantity (we will see how to convert strings to numeric values in the next section).

Before demonstrating the use of the `input()` function, let's expand on what we mean when we say a function *returns* a value. As we will see, functions can appear in expressions. When Python encounters a function in an expression, it evaluates (calls) that function. If a function appears in an expression, it will almost certainly *return* some form of data.[2] As it evaluates the expression, Python will replace the function with whatever the function returned. For the `input()` function, after it has been called, it is as though it disappears and is replaced by whatever string the function returns.

Listing 3.1 demonstrates the behavior of the `input()` function. Here the goal is to obtain

From the file: `input.tex`

[1]For now, our output only goes to a screen. In a later chapter we will learn how to write output to a file.

[2]Not all functions return data. The `print()` function is an example of a function that does not return data—although it generates output, it does not produce data that can be used in an expression. This is considered in greater detail in Chap. 4.

a user's name and age. Based on this information we want to print a personalized greeting and determine how many multiples of 12 are in the user's age (the Chinese zodiac uses a 12-year cycle).

Listing 3.1 Demonstration of the use of the `input()` function to obtain input from the keyboard. The `input()` function always returns a string.

```
1  >>> name = input("Enter your name: ") # Prompt for and obtain name.
2  Enter your name: Ishmael
3  >>> # Greet user.  However, the following produces an undesired space.
4  >>> print("Greetings", name, "!")
5  Greetings Ishmael !
6  >>> # Remove undesired spaces using the sep optional argument.
7  >>> print("Greetings ", name, "!", sep="")
8  Greetings Ishmael!
9  >>> age = input("Enter your age: ")   # Prompt for and obtain age.
10 Enter your age: 37
11 >>> # Attempt to calculate the number of 12-year cycles of the
12 >>> # Chinese zodiac the user has lived.
13 >>> chinese_zodiac_cycles = age // 12
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 TypeError: unsupported operand type(s) for //: 'str' and 'int'
17 >>> age       # Check age.  Looks like a number but actually a string.
18 '37'
19 >>> type(age) # Explicitly check age's type.
20 <class 'str'>
```

In line 1 the `input()` function appears to the right of the assignment operator. As part of Python's evaluation of the expression to the right of the assignment operator, it invokes the `input()` function. `input()`'s string argument is printed as shown on line 2. After this appears, the program waits for the user's response. We see, also in line 2, that the user responds with `Ishmael`. After the user types return, `input()` returns the user's response as a string. So, in this particular example the right side of line 1 ultimately evaluates to the string `Ishmael` which is assigned to the variable `name`.

In line 4 a `print()` statement is used to greet the user using a combination of two string literals and the user's name. As shown on line 5, the output from this statement is less than ideal in that it contains a space between the name and the exclamation point. We can remove this using the optional parameter `sep` as shown in lines 7 and 8.

In line 9 the user is prompted to enter his or her age. The response, shown in line 10, is `37` and this is assigned to the variable `age`. The goal is next to calculate the multiples of 12 in the user's age. In line 13 an attempt it made to use floor division to divide `age` by `12`. This produces a `TypeError` exception as shown in lines 14 through 16. Looking more closely at line 16 we see that Python is complaining about operands that are a `str` and an `int` when doing floor division. This shows us that, even though we wanted a numeric value for the age, at this point the variable `age` points to a string and thus `age` can only be used in operations that are valid for a string. This

leads us to the subject of the next section which shows a couple of the ways data of one type can be converted to another type.

Before turning to the next section, however, it is worth mentioning now that if a user simply types return when prompted for input, the `input()` function will return the empty string. Later we will exploit this to determine when a user has finished entering data.

3.2 Explicit Type Conversion: `int()`, `float()`, and `str()`

We have seen that there are instances in which data of one type are implicitly converted to another type. For example, when adding an integer and a `float`, the integer is implicitly converted to a `float` and the result is a `float`. This might lead us to ask: What happens if we try to add a string and an integer or perhaps multiply a string and a `float`? Our intuition probably leads us to guess that such operations will produce an error if the string doesn't look like a number, but the answer isn't necessarily obvious if we have a string such as `"1492"` or `"1.414"`. We'll try a few of these types of operations.

Listing 3.2 illustrates a couple of attempts to use strings in arithmetic operations.

Listing 3.2 Attempts to add a string and an integer and to multiply a string and a `float`. Both attempts result in errors.[3]

```
1 >>> "1492" + 520
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: cannot convert 'int' object to str implicitly
5 >>> "1.414" * 1.414
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 TypeError: cannot multiply sequence by non-int of type 'float'
```

Line 1 attempts to add the string `"1492"` and the integer `520`. This attempt fails and produces a `TypeError`. However, note carefully the error message on line 4. It says that it cannot convert an integer to a string *implicitly*. This suggests that perhaps *explicit* conversion is possible, i.e., we have to state more clearly the type of conversion we want to occur. (Also, we want the string to be converted to an integer, not the other way around.)

The attempt to multiply a string and a `float` in line 5 also fails and produces a `TypeError`. Here the error message is a bit more cryptic and, at this stage of our knowledge of Python, doesn't suggest a fix. But, the way to "fix" the statement in line 1 is the way to fix the statement in line 4: we need to explicitly convert one of the operands to a different type to obtain a valid statement.

The `int()` function converts its argument to an integer while the `float()` function converts its argument to a `float`. The arguments to these functions can be either a string or a number (or an expression that returns a string or a number). If the argument is a string, it must "appear" to be an appropriate numeric value. In the case of the `int()` function, the string must look like an

[3]The actual error messages produced using Python 3.2.2 have been change slightly in this example. For the sake of formatting and consistency, "`Can't`" and "`can't`" have been changed to `cannot`.

integer (i.e., it cannot look like a `float` with a decimal point). Listing 3.3 illustrates the behavior of these two functions.

Listing 3.3 Demonstration of the `int()` and `float()` functions.

```
>>> int("1492")   # Convert string to an int.
1492
>>> int("1.414") # String must look like an int to succeed.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.414'
>>> # Explicit conversion of string allows following to succeed.
>>> int("1492") + 520
2012
>>> int(1.414)      # Can convert a float to an int.
1
>>> int(2.999999)   # Fractional part discarded -- rounding not done.
2
>>> int(-2.999999) # Same behavior for negative numbers.
-2
>>> float("1.414") # Conversion of string that looks like a float.
1.414
>>> float("1.414") * 1.414   # Arithmetic operation now works.
1.9993959999999997
>>> 1.414 * 1.414   # Result is same if we enter floats directly.
1.9993959999999997
```

Line 1 demonstrates that a string that looks like an integer is converted to an integer by `int()`. Line 3 shows that a `ValueError` is produced if the argument of the `int()` function is a string that looks like a `float`. Line 8 shows that, by using explicit conversion, this arithmetic operation now succeeds: The string `"1492"` is converted to an integer and added to the integer `520` to produce the integer result `2012`.

As line 10 illustrates, the `int()` function can also take a numeric argument. Here the argument is a `float`. The integer result on line 11 shows the fractional part has been discarded. The `int()` function does not round to the nearest integer. Rather, as illustrated in lines 12 to 15, it merely discards the fractional part.

It might seem the `int()` function's behavior is inconsistent in that it will not accept a string that looks like a `float` but it will accept an actual `float`. However, if one wants to convert a string that looks like a `float` to an integer, there are really two steps involved. First, the string must be converted to a `float` and then the `float` must be converted to an integer. The `int()` function will not do both of these steps. If a programmer wants this type of conversion, some additional code has to be provided. We will return to this shortly.

The expression in line 16 of Listing 3.3 shows that by using explicit conversion we can now multiply the value represented by the string `"1.414"` and the `float 1.414`. If you spend a moment looking at the result of this multiplication given in line 17, you may notice this result is *not* what you would get if you multiplied these values using most calculators. On a calculator you

are likely to obtain `1.999396`. Is the error a consequence of our having converted a string to a `float`? Lines 18 and 19 answer this. In line 18 the `float` values are entered directly into the expression and the identical result is obtained. The "error" is a consequence of these values being `floats` and, as discussed in Sec. 2.1, the fact that `floats` have finite precision.

With the ability to convert strings to numeric quantities, we can now revisit the code in Listing 3.1. Listing 3.4 demonstrates, in lines 5 and 6, the successful conversion of the user's input, i.e., the user's age, to a numeric value, here an integer.

Listing 3.4 Demonstration of the use of the `input()` function to obtain input from the keyboard. The `input()` function always returns a string.

```
1  >>> name = input("Enter your name: ")
2  Enter your name: Captain Ahab
3  >>> print("Greetings ", name, "!", sep="")
4  Greetings Captain Ahab!
5  >>> age = int(input("Enter your age: ")) # Obtain user's age.
6  Enter your age: 57
7  >>> # Calculate the number of complete cycles of the 12-year
8  >>> # Chinese zodiac the user has lived.
9  >>> chinese_zodiac_cycles = age // 12
10 >>> print("You have lived", chinese_zodiac_cycles,
11 ...       "complete cycles of the Chinese zodiac.")
12 You have lived 4 complete cycles of the Chinese zodiac.
```

In line 5 the `int()` function is used to convert the string that is returned by the `input()` function to an integer. Note that the construct here is something new: the functions are *nested*, i.e., the `input()` function is inside the `int()` function. Nesting is perfectly acceptable and is done quite frequently. The innermost function is invoked first and whatever it returns serves as an argument for the surrounding function.

Note that the code in Listing 3.4 is not very *robust* in that reasonable input could produce an error. For example, what if `Captain Ahab` enters his age as `57.5` instead of `57`? In this case the `int()` function would fail. One can spend quite a bit of effort trying to ensure that a program is immune to "incorrect" input. However, at this point, writing robust code is not our primary concern, so we typically will not dwell on this issue.

Nevertheless, along these lines, let's return to the point raised above about the inability of the `int()` function to handle a string argument that looks like a `float`. Listing 3.5 shows that if an integer value is ultimately desired, one can first use `float()` to safely convert the string to a `float` and then use `int()` to convert this numeric value to an integer.

Listing 3.5 Intermediate use of the `float()` function to allow entry of strings that appear to be either `floats` or `ints`.

```
1  >>> # Convert a string to a float and then to an int.
2  >>> int(float("1.414"))
3  1
```

```
4  >>> # float() has no problem with strings that appear to be ints.
5  >>> float("14")
6  14.0
7  >>> int(float("14"))
8  14
9  >>> # Desire an integer age but user enters a float, causing an error.
10 >>> age = int(input("Enter age: "))
11 Enter age: 57.5
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 ValueError: invalid literal for int() with base 10: '57.5'
15 >>> # Can use float() to allow fractional ages.
16 >>> age = int(float(input("Enter your age: ")))
17 Enter your age: 57.5
18 >>> age
19 57
```

In line 1 the `float()` function, which is nested inside the `int()` function, converts the string `"1.414"` to a `float`. The `int()` function converts this to an integer, discarding the fractional part, and the resulting value of `1` is shown on line 2. As shown in lines 5 through 8, the `float()` function can handle string arguments that appear to be integers.

Line 10 of Listing 3.5 uses the same statement as was used in Listing 3.4 to obtain an age. In this example, the user enters `57.5`. Since the `int()` function cannot accept a string that doesn't appear as an integer, an error is produced as shown in lines 12 through 14. (Although line 10 is shown in red, it does not explicitly contain a bug. Rather, the input on line 11 cannot be handled by the statement on line 10. Thus both line 10 and the input on line 11 are shown in red.)

If one wants to allow the user to enter fractional ages, the statement shown in line 16 can be used. Here three nested functions are used. The innermost function, `input()`, returns a string. This string is passed as an argument to the middle function, `float()`, which returns a `float`. This `float` is subsequently the argument for the outermost function, `int()`, which returns an integer. Line 18 shows that, indeed, the variable `age` has been set to an integer.

We've seen that `int()` and `float()` can convert a string to a numeric value. In some sense, the `str()` function is the converse of these functions in that it can convert a numeric value to a string. In fact, all forms of data in Python have a string representation. At this point, the utility of the `str()` function isn't obvious, but in subsequent chapters we will see that it proves useful in a number of situations (e.g., in forming a suitable prompt for the `input()` function which consists of a combination of text and a numeric value as shown in Listing 6.21). For now, in Listing 3.6, we merely demonstrate the behavior of `str()`. Please read the comments in the listing.

Listing 3.6 Demonstration that the `str()` function converts its argument to a string.

```
1 >>> str(5)                # Convert int to a string.
2 '5'
3 >>> str(1 + 10 + 100)     # Convert int expression to a string.
4 '111'
5 >>> str(-12.34)           # Convert float to a string.
```

```
'-12.34'
>>> str("Hello World!")   # str() accepts string arguments.
'Hello World!'
>>> str(divmod(14, 9))    # Convert tuple to a string.
'(1, 5)'
>>> x = 42
>>> str(x)                # Convert int variable to a string.
'42'
```

3.3 Evaluating Strings: eval()

Python provides a powerful function called `eval()` that is sometimes used to facilitate obtaining input. However, because of this function's power, one should be cautious about using it in situations where users could potentially cause problems with "inappropriate" input. The `eval()` function takes a string argument and *evaluates* that string as a Python expression, i.e., just as if the programmer had directly entered the expression as code. The function returns the result of that expression. Listing 3.7 demonstrates `eval()`.

Listing 3.7 Basic demonstration of the `eval()` function.

```
>>> string = "5 + 12" # Create a string.
>>> print(string)     # Print the string.
5 + 12
>>> eval(string)      # Evaluate the string.
17
>>> print(string, "=", eval(string))
5 + 12 = 17
>>> eval("print('Hello World!')") # Can call functions from eval().
Hello World!
>>> # Using eval() we can accept all kinds of input...
>>> age = eval(input("Enter your age: "))
Enter your age: 57.5
>>> age
57.5
>>> age = eval(input("Enter your age: "))
Enter your age: 57
>>> age
57
>>> age = eval(input("Enter your age: "))
Enter your age: 40 + 17 + 0.5
>>> age
57.5
```

In line 1 a string is created that looks like an expression. In line 2 this string is printed and the output is the same as the string itself. In line 4, the string is the argument to the `eval()` function.

eval() evaluates the expression and returns the result as shown in line 5. The print() statement in line 6 shows both the string and the result of evaluating the string. Line 8 shows that one can call a function via the string that eval() evaluates—in this statement the string contains a print() statement.

In line 11 the input() function is nested inside the eval() function. In this case whatever input the user enters will be evaluated, i.e., the string returned by input() will be the argument of eval(), and the result will be assigned to the variable age. Because the user entered 57.5 in line 12, we see, in lines 13 and 14, that age is the float value 57.5. Lines 15 through 18 show what happens when the user enters 57. In this case age is the integer 57. Then, in lines 19 through 22, we see what happens when the user enters an expression involving arithmetic operations—eval() handles it without a problem.

Recall that, as shown in Sec. 2.4, simultaneous assignment can be used if multiple expressions appear to the right of the assignment operator and multiple variables appear to the left of the assignment operator. The expressions and variables must be separated by commas. This ability to do simultaneous assignment can be coupled with the eval() function to allow the entry of multiple values on a single line. Listing 3.8 demonstrates this.

Listing 3.8 Demonstration of how eval() can be used to obtain multiple values on a single line.

```
1  >>> eval("10, 32")          # String with comma-separated values.
2  (10, 32)
3  >>> x, y = eval("10, 20 + 12") # Use simultaneous assignment.
4  >>> x, y
5  (10, 32)
6  >>> # Prompt for multiple values.  Must separate values with a comma.
7  >>> x, y = eval(input("Enter x and y: "))
8  Enter x and y: 5 * 2, 32
9  >>> x, y
10 (10, 32)
```

When the string in the argument of the eval() function in line 1 is evaluated, it is treated as two integer literals separated by a comma. Thus, these two values appear in the output shown in line 2. We can use simultaneous assignment, as shown in line 3, to set the value of two variables when eval()'s string argument contains more than one expression.

Lines 7 through 10 show that the user can be prompted for multiple values. The user can respond with a wide range of expressions if these expressions are separated by a comma.

As an example of multiple input on a single line, let us calculate a user's body mass index (BMI). BMI is a function of height and weight. The formula is

$$\text{BMI} = 703\frac{W}{H^2}$$

where the weight W is in pounds and the height H is in inches. The code shown in Listing 3.9 shows how the height and weight can be obtained from the user and how the BMI can be calculated.

Listing 3.9 Obtaining multiple values on a single line and calculating the BMI.

```
>>> print("Enter weight and height separated by a comma.")
Enter weight and height separated by a comma.
>>> weight, height = eval(input("Weight [pounds], Height [inches]: "))
Weight [pounds], Height [inches]: 160, 72
>>> bmi = 703 * weight / (height * height)
>>> print("Your body mass index is", bmi)
Your body mass index is 21.6975308642
```

Line 3 is used to obtain both weight and height. The user's response on line 4 will set these both to integers but it wouldn't have mattered if `float` values were entered. The BMI is calculated in line 5. In the denominator `height` is multiplied by itself to obtain the square of the height. However, one could have instead used the power operator, i.e., `height ** 2`. Because `float` division is used, the result will be a `float`. The BMI is displayed using the `print()` statement in line 6. In all likelihood, the user wasn't interested in knowing the BMI to 10 decimal places. In Sec. 9.7 we will see how the output can be formatted more reasonably.

Again, a word of caution about the use of `eval()`: by allowing the user to enter an expression that may contain calls to other functions, the user's input could potentially have some undesired consequences. Thus, if you know you want integer input, it is best to use the `int()` function. If you know you want `float` input, it is best to use the `float()` function. If you really need to allow multiple values in a single line of input, there are better ways to handle it; these will be discussed in Chap. 9.

3.4 Chapter Summary

`input()`: Prompts the user for input with its string argument and returns the string the user enters.

`int()`: Returns the integer form of its argument.

`float()`: Returns the `float` form of its argument.

`eval()`: Returns the result of evaluating its string argument as any Python expression, including arithmetic and numerical expressions.

Functions such as the four listed above can be *nested*. Thus, for example, **`float(input())`** can be used to obtain input in string form which is then converted to a float value.

3.5 Review Questions

1. The following code is executed

```
x = input("Enter x: ")
```

In response to the prompt the user enters

```
-2 * 3 + 5
```

What is the resulting value of `x`?

(a) `-16`
(b) `-1`
(c) `'-2 * 3 + 5'`
(d) This produces an error.

2. The following code is executed

```
y = int(input("Enter x: ")) + 1
```

In response to the prompt the user enters

```
50
```

What is the resulting value of `y`?

(a) `'50 + 1'`
(b) `51`
(c) `50`
(d) This produces an error.

3. The following code is executed

```
y = int(input("Enter x: ") + 1)
```

In response to the prompt the user enters

```
50
```

What is the resulting value of `y`?

(a) `'50 + 1'`
(b) `51`
(c) `50`
(d) This produces an error.

4. The following code is executed

```
x = input("Enter x: ")
print("x =", x)
```

In response to the prompt the user enters

```
2 + 3 * -4
```

What is the output produced by the `print()` statement?

(a) `x = -10`
(b) `x = -20`
(c) `x = 2 + 3 * -4`
(d) This produces an error.
(e) None of the above.

5. The following code is executed

```
x = int(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
2 + 3 * -4
```

What is the output produced by the `print()` statement?

(a) `x = -10`
(b) `x = -20`
(c) `x = 2 + 3 * -4`
(d) This produces an error.
(e) None of the above.

6. The following code is executed

```
x = int(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
5.0
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`
(b) `x = 5`
(c) This produces an error.
(d) None of the above.

7. The following code is executed

```
x = float(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

8. The following code is executed

```
x = eval(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

9. The following code is executed

```
x = input("Enter x: ")
print("x =", x)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

10. The following code is executed

```
x = int(input("Enter x: "))
print("x =", x + 1.0)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`
(b) `x = 6`
(c) This produces an error.
(d) None of the above.

11. The following code is executed

```
x = float(input("Enter x: "))
print("x =", x + 1)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`
(b) `x = 6`
(c) This produces an error.
(d) None of the above.

12. The following code is executed

```
x = eval(input("Enter x: "))
print("x =", x + 1)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`
(b) `x = 6`
(c) This produces an error.
(d) None of the above.

13. The following code is executed

```
x = input("Enter x:  ")
print("x =", x + 1)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`

(b) `x = 6`

(c) This produces an error.

(d) None of the above.

14. The following code is executed

```
x = eval(input("Enter x:  "))
print("x =", x)
```

In response to the prompt the user enters

```
2 + 3 * -4
```

What is the output produced by the `print()` statement?

(a) `x = -10`

(b) `x = -20`

(c) `x = 2 + 3 * -4`

(d) This produces an error.

(e) None of the above.

15. What is produced by the `print()` statement in the following code?

```
s = "8 / 4 + 4"
print(s, eval(s), sep=" = ")
```

What is the resulting value of `y`?

(a) This produces an error.

(b) `8 / 4 + 4 = 6`

(c) `6.0 = 6.0`

(d) `8 / 4 + 4 = 6.0`

(e) None of the above.

16. True or False: All of the following are acceptable arguments for the `int()` function: `5`, `5.0`, `"5"`, and `"5.0"` (these arguments are an `int`, a `float`, and two `str`s, respectively).

17. True or False: All of the following are acceptable arguments for the `float()` function: `5`, `5.0`, `"5"`, and `"5.0"`.

18. True or False: All of the following are acceptable arguments for the `eval()` function: `5`, `5.0`, `"5"`, and `"5.0"`.

19. True or False: The string `"5.0, 6.0"` is an acceptable argument for the `eval()` function but not for the `float()` function.

ANSWERS: 1) c; 2) b; 3) d, the addition is attempted before conversion to an `int`; 4) c; 5) d; 6) c; 7) a; 8) b; 9) b; 10) a; 11) a; 12) b; 13) c, cannot add string and integer; 14) a; 15) d; 16) False; 17) True; 18) False, the argument must be a string; 19) True.

3.6 Exercises

1. A commonly used method to provide a rough estimate of the right length of snowboard for a rider is to calculate 88 percent of their height (the actual ideal length really depends on a large number of other factors). Write a program that will help people estimate the length of snowboard they should buy. Obtain the user's height in feet and inches (assume these values will be entered as integers) and display the length of snowboard in centimeters to the user. There are 2.54 centimeters in an inch.

   The following demonstrates the proper behavior of the program:

```
1 Enter your height.
2 Feet: 5
3 Inches: 4
4
5 Suggested board length: 143.0528 cm
```

2. Newton's Second Law of motion is expressed in the formula $F = m \times a$ where F is force, m is mass, and a is acceleration. Assume that the user knows the mass of an object and the force on that object but wants to obtain the object's acceleration a. Write a program that prompts the user to enter the mass in kilograms (kg) and the force in Newtons (N). The user should enter both values on the same line separated by a comma. Calculate the acceleration using the above formula and display the result to the user.

   The following demonstrates the proper behavior of the program:

```
1 Enter the mass in kg and the force in N: 55.4, 6.094
2
3 The acceleration is 0.11000000000000001
```

3. Write a program that calculates how much it costs to run an appliance per year and over a 10 year period. Have the user enter the cost per kilowatt-hour in cents and then the number of kilowatt-hours used per year. Assume the user will be entering floats. Display the cost to the user in dollars (where the fractional part indicates the fraction of a dollar and does not have to be rounded to the nearest penny).

   The following demonstrates the proper behavior of the program:

```
Enter the cost per kilowatt-hour in cents: 6.54
Enter the number of kilowatt-hours used per year: 789

The annual cost will be:  51.60060000000001
The cost over 10 years will be:  516.0060000000001
```

4. In the word game Mad Libs, people are asked to provide a part of speech, such as a noun, verb, adverb, or adjective. The supplied words are used to fill in the blanks of a preexisting template or replace the same parts of speech in a preexisting sentence. Although we don't yet have the tools to implement a full Mad Libs game, we can implement code that demonstrates how the game works for a single sentence. Consider this sentence from P. G. Wodehouse:

   > Jeeves lugged my purple socks out of the drawer as if he were a vegetarian fishing a caterpillar out of his salad.

   Write a program that will do the following:

   - Print the following template:

     ```
     Jeeves [verb] my [adjective] [noun] out of the [noun]
     as if he were a vegetarian fishing a [noun] out of his
     salad.
     ```

   - Prompt the user for a verb, an adjective, and three nouns.
   - Print the template with the terms in brackets replaced with the words the user provided.

   Use string concatenation (i.e., the combining of strings with the plus sign) as appropriate.

   The following demonstrates the proper behavior of this code

```
Jeeves [verb] my [adjective] [noun] out of the [noun] as if
he were a vegetarian fishing a [noun] out of his salad.

Enter a verb: bounced
Enter an adjective: invisible
Enter a noun: parka
Enter a noun: watermelon
Enter a noun: lion

Jeeves bounced my invisible parka out of the watermelon as if
he were a vegetarian fishing a lion out of his salad.
```

Chapter 4

Functions

In this chapter we consider another important component of useful programs: *functions*. A programmer often needs to solve a problem or accomplish a desired task but may not be given a precise specification of *how* the problem is to be solved. Determining the "how" and implementing the solution is the job of the programmer. For all but the simplest programs, it is best to divide the task into smaller tasks and associate these subtasks with *functions*.

Most programs consist of a number of functions.[1] When the program is run, the functions are *called* or *invoked* to perform their particular part of the overall solution. We have already used a few of Python's built-in functions (e.g., `print()` and `input()`), but most programming languages, including Python, allow programmers to create their own functions.[2]

The ability to organize programs in terms of functions provides several advantages over using a monolithic collection of statements, i.e., a single long list of statements:

- Even before writing any code, functions allow a hierarchical approach to *thinking about* and solving the problem. As mentioned, one divides the overall problem or task into smaller tasks. Naturally it is easier to think about the details involved in solving these subtasks than it is to keep in mind all the details required to solve the entire problem. Said another way, the ability to use functions allows us to adopt a *divide and conquer* approach to solve a problem.

- After determining the functions that are appropriate for a solution, these functions can be implemented, tested, and debugged, individually. Working with individual functions is often far simpler than trying to implement and debug monolithic code.

- Functions provide *modularization* (or compartmentalization). Suppose a programmer associates a particular task with a particular function, but after implementing this function, the programmer determines a better way to implement what this function does. The programmer can rewrite (or replace) this function. It the programmer does not change the way data are entered into the function nor the way data are returned by the function, the programmer can replace this function without fear of affecting any other aspect of the overall program. (With monolithic code, this is generally not the case.)

From the file: `functions.tex`

[1] The term *function* can mean many things. We will not attempt to completely nail down a meaning here. Instead, we will introduce various aspects of what constitutes a function and hope the definition eventually becomes self-evident.

[2] What we are calling functions might be called subroutines or methods in other languages.

- Functions facilitate code reuse. Often some of the smaller subtasks that go into solving one particular problem are common to other problems. Once a function has been properly implemented, it can be reused in any number of programs.

To further motivate the use of functions, consider the BMI calculation performed in Listing 3.9. In this code the calculation is entered as an expression that is evaluated once. Surrounding code provides the input and output (I/O). But what if one wants to calculate the BMI for several people? Ideally one would *not* have to enter all the statements each time the calculation is performed. This is a situation where functions are useful: these statements can be placed in a function and then the function can be called (or invoked) whenever the calculation needs to be performed.

Broadly speaking, in Python, and in several other languages, functions are used in one of three ways:

1. A function may be called for the value it returns. In some ways, functions like this are similar to the mathematical functions with which you are familiar. Usually some values are passed into the function as arguments (or, more formally, parameters). The function generates new data based on these values and then *returns* this new data to the point in the program where the function was called. A function that returns data is said to be a *non-void function*. (We will soon see how a function returns data.)

2. A function may be called solely for its *side effects*. A side effect is an action the function takes that does not produce a return value. For example, a function may be called to modify the values in a list of data. As another example, a function may be called to generate some output using one or more `print()` statements. It is important not to confuse output, such as a `print()` statement produces, with the return value of a function. (We say more related to this point below.) If the function does not explicitly return data, we will say it is a *void function*.

3. A function may be called both for its side effects and its return value. Such a function is also non-void (since it explicitly returns something), but the code that calls the function does not necessarily use the value returned by the function. In some cases, when the side effect is the only action of interest, the return value is ignored.

In the following we consider both void and non-void functions.[3]

4.1 Void Functions and **None**

Before turning to the creation of our own functions, let us further consider the `print()` function. As we've seen, the `print()` function produces output, but it is, in fact, a *void function*. If a function doesn't explicitly return data and yet it is used in an expression, the function evaluates to `None` (`None` is one of the keywords listed in Listing 2.15). `None` is, in some ways, Python's way of saying, "I'm nothing. I don't exist."

To illustrate how `None` can appear in code, consider Listing 4.1.

[3]In the strictest sense, all Python functions are non-void in that even the ones that do not explicitly return a value will return `None`. This point is discussed in the following pages. Nevertheless, we will identify functions that only return `None` as void functions.

Listing 4.1 Demonstration that `print()` is a void function that returns `None`. This code also demonstrates that `None` generally cannot be used in expressions.

```
1  >>> type(None) # Check None's type.
2  <class 'NoneType'>
3  >>> None + 2   # Try to use None in an arithmetic operation.
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
7  >>> # Assign return value of print() to x.  Since print() is a void
8  >>> # function, x is set to None.
9  >>> x = print("Salutations!")
10 Salutations!
11 >>> type(x)      # Check x's type.
12 <class 'NoneType'>
13 >>> x            # Nothing is echoed with entry of x.
14 >>> print(x)     # print() shows us x is None.
15 None
16 >>> x * 2        # Cannot use x (None) in arithmetic expression.
17 Traceback (most recent call last):
18   File "<stdin>", line 1, in <module>
19 TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

Line 1 uses `type()` to check the type of `None`. We see it is of type `NoneType`. Again, this essentially says "I don't exist." As line 3 indicates, `None` cannot be used in an arithmetic operation.

In line 9 the assignment operator is used to assign whatever the `print()` function returns to the variable `x`. However, *although the* `print()` *produces output, it does not explicitly return anything*. Since `print()` is a void function, the statement in line 9 sets `x` to `None`. This fact is demonstrated by checking `x`'s type as is done in line 11. In line 13, `x` is entered on a line by itself. In the interactive environment, this usually results in the value of the variable being echoed to the programmer. However, if the value of the variable is `None`, then no output is produced. If the variable is printed using a `print()` statement, as is done in line 14, then the output is the word `None`. Line 16 again illustrates that `None` (whether the literal or in the form of a variable) cannot be used in an arithmetic operation.

It may not be obvious at the moment, but the values returned by functions will be extremely important in much of what we will do later. Also, at some point in the future, you are likely to encounter a bug that involves `None` showing up in your code. So, although you do not need to have a deep understanding of `None`, you should have a feel for how it can appear in your code and what it represents.

4.2 Creating Void Functions

To create a function, one uses the template shown in Listing 4.2 where the terms in angle brackets are replaced with appropriate values as discussed below.

Listing 4.2 Template for defining a function.

```
def <function_name>(<parameter_list>):
    <body>
```

In this template the first line is known as the *header*. The first word of the header is the keyword def which you should think of as *define*, i.e., we are defining a new function. This keyword is followed by the function's name which is any identifier that follows the rules given in Sec. 2.6. The parentheses following the function's name are mandatory. They enclose a list of *formal* parameters that are identifiers separated by commas (i.e., a list of variable names). The actual values assigned to these parameters are established when the function is called; thus, expressions enclosed in parentheses when the function is called are known as the *actual* parameters. We will return to this point shortly. It should also be noted that a function is not required to have any parameters. If a function has no parameters, the parentheses enclose nothing. The parentheses are followed by a colon which signals the end of the header. The body of the function starts on the second line and is given in *indented code*.[4] The body can consist of any number of statements and it continues until the code is no longer indented. Indentation of statements must be at the same level. (In an integrated development environment such as IDLE, when one hits return, the proper indentation will generally be provided for you. To terminate the definition of the function in an interactive environment, you simply enter a blank line. When using IDLE to edit a file, you may have to use the delete key to remove the indentation that is automatically entered at the start of a line.)

The function can be terminated by a return statement, which ends execution of the function and returns execution to the location where the function was called.[5] If no return statement is present, execution continues until the end of the body, at which point execution returns to the point in the program where the function was called. (Thus, a return statement at the end of the body does not affect the execution in any way, but it is often included in the function definition to signal the end of the function.)

A function is not executed when it is defined. Rather, to execute (or call, or invoke) a function, one writes the function's name followed by parentheses with the requisite number of (actual) parameters.

Listing 4.3 provides a demonstration of the creation of two void functions. When a line ends with a colon, the Python interpreter knows more input is needed. Thus, in line 3, the prompt changes to indicate more input is expected. As you add lines of indented code, the interactive prompt will continue to be three dots. In the interactive environment you must use a blank line to terminate a function's body.[6]

[4]More formally, the body is known as the suite and the header and suite together are considered a *clause*.

[5]In Python, as with most computer languages, statements are executed sequentially, one after the other. However, when a program is run, in some sense the order in which statements are executed may not be the same as the order in which the statements were written. Later we will consider statements which explicitly alter the flow of execution (e.g., if statements or for loops which can tell Python to skip or repeat statements, respectively). However, when there is an expression in which a function is called, this effectively tells Python to not proceed further until the statements in the body of that particular function have been executed.

[6]In the command-line interactive environment, "semi-blank" lines can be included in the body of a function. In this case the indentation must be present, but the rest of the line is left blank. This can sometimes enhance the readability

Listing 4.3 Demonstration of the creation of two functions that do not explicitly return anything (hence they evaluate to `None` if used in an expression). Note that the function `say_hi()` is used in the body of the function `greet_user()`.

```
>>> # Create Hello-World function that takes no parameters.
>>> def say_hi():
...     print("Hello World!")
...
>>> say_hi()  # Check that function works properly.
Hello World!
>>> # Create function to greet a user.
>>> def greet_user(name):
...     say_hi()
...     print("Oh!  And hello ", name, "!", sep="")
...     return
...
>>> greet_user("Starbuck")  # Provide string literal as argument.
Hello World!
Oh!  And hello Starbuck!
>>> the_whale = "Moby Dick"
>>> greet_user(the_whale)   # Provide string variable as argument.
Hello World!
Oh!  And hello Moby Dick!
```

In lines 2 and 3 we define the function `say_hi()` which takes no parameters. The body of this function consists of a single `print()` statement. In line 5 this function is called to ensure that it works properly. The output on line 6 shows that it does.

In lines 8 through 11 the function `greet_user()` is defined. This function has a single parameter called `name`. The first line in the body of this function (line 9 in the overall listing) is a call to the function `say_hi()`. This is followed by a `print()` statement that has as one of its arguments the `name` parameter. The third line of the body is a `return` statement. When this function is called, the statements in the body are executed sequentially. Thus, first the `say_hi()` function is executed, then the `print()` statement is executed, then the `return` statement instructs the interpreter to return to the point in the code where this function was called. Although it is not an error to have statements in the body of the function after a `return` statement, such statements would never be executed.

The order in which the `say_hi()` and `greet_user()` functions are defined is not important. Either could have been defined first. However, it is important that every function is defined before it is called. Thus, both the `say_hi()` and `greet_user()` functions must be defined before the `greet_user()` function is called. (If one defines the `greet_user()` first and then called it before the `say_hi()` function was defined, an error would occur because the interpreter would not know what to do with the call to `say_hi()` that appears in the body of `greet_user()`.)

In line 13 the `greet_user()` function is called with the string literal `"Starbuck"` as the argument. This is the *actual* parameter for this particular invocation of the function. Whenever

of a multi-line function.

a function is called, the parameters that are given are the actual parameters. Recall that when the function was defined, the list of parameters in parentheses were the *formal* parameters, and these consisted of identifiers (i.e., variable names). Before the body of the function is executed, *the actual parameters are assigned to the formal parameters*. So, just before the body of the greet_user() function is executed, it is as if this statement were executed:

```
name = "Starbuck"
```

The output in lines 14 and 15 shows that the function works as intended.

In line 17 greet_user() is called with a variable that has been defined in line 16. So, for this particular invocation, before the body of the function executes, it is as if this statement had been executed:

```
name = the_whale
```

Since the_whale has been set to Moby Dick, we see this whale's name in the subsequent output in lines 18 and 19.

Now let's return to the calculation of the BMI which was previously considered in Listing 3.9. Listing 4.4 shows one way in which to implement a function that can calculate a person's BMI.

Listing 4.4 Calculation of BMI using a void function. Here all input and output are handled within the function bmi().

```
1  >>> def bmi():          # Define function.
2  ...     weight = float(input("Enter weight [pounds]: "))
3  ...     height = float(input("Enter height [inches]: "))
4  ...     bmi = 703 * weight / (height * height)
5  ...     print("Your body mass index is:", bmi)
6  ...
7  >>> bmi()               # Invoke bmi() function.
8  Enter weight [pounds]: 160
9  Enter height [inches]: 72
10 Your body mass index is: 21.6975308642
11 >>> bmi()               # Invoke bmi() function again.
12 Enter weight [pounds]: 123
13 Enter height [inches]: 66
14 Your body mass index is: 19.8505509642
```

In lines 1 through 5 the function bmi() is defined. It takes no parameters and handles all of its own input and output. This function is called in lines 7 and 11. Note the ease with which one can now calculate a BMI.

4.3 Non-Void Functions

Non-void functions are functions that return something other than None. The template for creating a non-void function is identical to the template given in Listing 4.2 for a void function. What

distinguishes the two types of functions is how `return` statements that appear in the body of the function are implemented. In the previous section we mentioned that a `return` statement can be used to explicitly terminate the execution of a function. When the `return` statement appears by itself, the function returns `None`, i.e., it behaves as a void function. However, to return some other value from a function, one simply puts the value (or an expression that evaluates to the desired value) following the keyword `return`.

Listing 4.5 demonstrates the creation and use of a non-void function. Here, as mentioned in the comments in the first two lines, the function calculates a (baseball) batting average given the number of hits and at-bats.

Listing 4.5 Demonstration of the creation of a non-void function.

```
>>> # Function to calculate a batting average given the number of
>>> # hits and at-bats.
>>> def batting_average(hits, at_bats):
...     return int(1000 * hits / at_bats)
...
>>> batting_average(85, 410)
207
>>> # See what sort of help we get on this function.  Not much...
>>> help(batting_average)
Help on function batting_average in module __main__:

batting_average(hits, at_bats)
```

Note that the `return` statement in line 4 constitutes the complete body of the function. In this case the function returns the value of the expression immediately following the keyword `return`. Lines 6 and 7 show what happens when the function is invoked with 85 hits and 410 at-bats. As mentioned, before the body of the function is executed, the actual parameters are assigned to the formal parameters. So, for the statement in line 6, before the body of `batting_average()` is executed, it is as if these two statements had been issued:

```
hits = 85
at_bats = 410
```

The expression following the keyword `return` in the body of the function evaluates to `207` for this input (a batting average of "207" means if this ratio of hits to at-bats continues, one can anticipate 207 hits out of 1,000 at-bats). Because the `batting_average()` function was invoked in the interactive environment and its return value was not assigned to a variable, the interactive environment displays the return value (as given on line 7).

In line 9 the `help()` function is called with an argument of `batting_average` (i.e., the function's name without any parentheses). The subsequent output in lines 10 through 12 isn't especially helpful although we are told the parameter names that were used in defining the function. If these are sufficiently descriptive, this may be useful.

Listing 4.6 demonstrates the creation of a slightly more complicated non-void function. The goal of this function is to determine the number of dollars, quarters, dimes, nickels, and pennies in

a given "total" (where the total is assumed to be in pennies). Immediately following the function header, a multi-line string literal appears that describes what the function does. This string isn't assigned to anything and thus it is seemingly discarded by the interpreter. However, when a string literal is given immediately following a function header, it becomes what is known as a "docstring" and will be displayed when help is requested for the function. This is demonstrated in lines 16 through 21. So, rather than describing what a function does in comments given just prior to the definition of a function, we will often use a docstring to describe what a function does. When more than one sentence is needed to describe what a function does, it is recommended that the docstring be started by a single summary sentence. Next there should be a blank line which is then followed by the rest of the text.

Listing 4.6 A function to calculate the change in terms of dollars, quarters, dimes, nickels, and pennies for a given "total" number of pennies

```
1  >>> # Function to calculate change.
2  >>> def make_change(total):
3  ...     """
4  ...     Calculate the change in terms of dollars, quarters,
5  ...     dimes, nickels, and pennies in 'total' pennies.
6  ...     """
7  ...     dollars, remainder  = divmod(total, 100)
8  ...     quarters, remainder = divmod(remainder, 25)
9  ...     dimes, remainder    = divmod(remainder, 10)
10 ...     nickels, pennies    = divmod(remainder, 5)
11 ...     return dollars, quarters, dimes, nickels, pennies
12 ...
13 >>> make_change(769)
14 (7, 2, 1, 1, 4)
15 >>> # Try to get help on this function.  "Docstring" is given!
16 >>> help(make_change)
17 Help on function make_change in module __main__:
18
19 make_change(total)
20     Calculate the change in terms of dollars, quarters,
21     dimes, nickels, and pennies in 'total' pennies.
```

Technically `return` statements can only return a single value. It might appear that the code in Listing 4.6 violates this since there are multiple values (separated by commas) in the `return` statement in line 11. However, in fact, these values are collected together and returned in a single collection known as a *tuple*. Later we will study tuples and other collections of data. For now it suffices to know that it is possible to get multiple values from a function simply by putting them in a comma-separated list in the `return` statement. The output in line 14 shows that `769` pennies is the equivalent of seven dollars, two quarters, one dime, one nickel, and four pennies. (One could use simultaneous assignment to store these values to individual variables.)

Now let us again revisit the calculation of a BMI. In the `bmi()` function of Listing 4.4, the user was prompted for a weight and height for each invocation of the function. However, there may

be applications for which weight and height data are stored in a file and thus it doesn't make sense to prompt for these values. In anticipation of situations such as this, it may be better to separate the calculation of the BMI from the code that handles the input (before the calculation) and the output (after the calculation).

Listing 4.7 demonstrates one way this might be done. Here three functions are implemented. `get_wh()` gets the weight and height; `calc_bmi()` calculates the BMI (given the weight and height as parameters); and `show_bmi()` displays the BMI (which it is given as a parameter).

Listing 4.7 Reimplementation of the BMI calculation where the calculation itself has been separated from the handling of the input and output.

```
1  >>> def get_wh():
2  ...     """Obtain weight and height from user."""
3  ...     weight = float(input("Enter weight [pounds]: "))
4  ...     height = float(input("Enter height [inches]: "))
5  ...     return weight, height
6  ...
7  >>> def calc_bmi(weight, height):
8  ...     """
9  ...     Calculate body mass index (BMI) for weight in
10 ...     pounds and height in inches.
11 ...     """
12 ...     return 703 * weight / (height * height)
13 ...
14 >>> def show_bmi(bmi):
15 ...     print("Your body mass index is:", bmi)
16 ...
17 >>> w, h = get_wh()
18 Enter weight [pounds]: 280
19 Enter height [inches]: 72
20 >>> bmi = calc_bmi(w, h)
21 >>> show_bmi(bmi)
22 Your body mass index is: 37.9706790123
```

Lines 1 through 16 define the three functions `get_wh()`, `calc_bmi()`, and `show_bmi()`. In line 17, `get_wh()` is called to obtain the weight and height. This leads to the user being prompted for input as shown in lines 18 and 19. The values the user enters are returned by `get_wh()` and assigned, simultaneously, to the variables `w` and `h`. (As will be discussed in the next section, the `weight` and `height` variables defined in the function `get_wh()` are not defined outside of that function. Thus, we need to store the values returned by the function. By "store" we mean assign the values to variables for later use.)

In line 20, `calc_bmi()` is called to calculate the BMI. If we have the weight and height (whether read from a file or obtained via the keyboard), this function can be used to calculate the BMI. The value this function returns is stored in the variable `bmi` which is subsequently passed as a parameter to `show_bmi()` in line 21 to obtain the desired output.

4.4 Scope of Variables

For now, you should always use parameters to pass data into a function and always use the `return` statement to obtain data from a function. (There are other ways to exchange data with a function using what are known as *global variables*, but the use of global variables is generally considered to be a dangerous programming practice.) Something that you need to keep in mind is that all the parameters and variables that are defined in a function are *local* to a function, meaning that these variables cannot be "seen" by code outside of the function.

The code over which a variable is accessible or visible is known as the variable's scope. The scope of a variable within a function is from the point it is created (either in the parameter list or in the body via an assignment operation) to the end of the function.

Although scope is an important and, at times, complicated issue, it is not something we want to belabor here. Nevertheless, there are a few issues related to scope about which you should be aware as this awareness may help prevent bugs from appearing in your code.

Listing 4.8 again uses the `get_wh()` function that was used in Listing 4.7.

Listing 4.8 Demonstration that variables are local to a function.

```
1  >>> def get_wh():
2  ...     """Obtain weight and height from user."""
3  ...     weight = float(input("Enter weight [pounds]: "))
4  ...     height = float(input("Enter height [inches]: "))
5  ...     return weight, height
6  ...
7  >>> get_wh()
8  Enter weight [pounds]: 120
9  Enter height [inches]: 60
10 (120.0, 60.0)
11 >>> print(weight)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 NameError: name 'weight' is not defined
15 >>> print(height)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 NameError: name 'height' is not defined
```

Lines 1 through 5 define `get_wh()` as before. This function is invoked in line 7. The user enters `120` for the weight and `60` for the `height`. Since the return value of this function is not assigned to anything, the interactive environment displays the return value in line 10 (were this code run from within a file, the return value would simply be ignored—it would not appear as output). In lines 11 and 15 an attempt is made to print the `weight` and `height`, respectively, that exist in the `get_wh()` function. Both these attempts produce `NameError`s with a statement that the variables are not defined. Again, these variables exist in the `get_wh()` function and only in `get_wh()`.

In Sec. 2.7 we discussed namespaces and how Python uses them to associate identifiers with values. Python actually maintains multiple namespaces. When a function is defined, there is a namespace created for that particular function. When a identifier is referenced within a function, Python will look in that function's namespace for the associated value. If it cannot find that identifier, it will look in the surrounding scope, i.e., in the namespace associated with place from which the function was called. (For example, if we call a function from the interactive environment, the *global* namespace is the surrounding scope for that function call.) However, if an identifier is referenced outside of a function, Python will never look inside the namespaces of functions to try to determine the value of that identifier. Similarly, if an identifier is referenced in one function, Python will not look in another function's namespace to try to determine the value of that identifier.

Now consider the code shown in Listing 4.9 which further demonstrates the local nature of variables. This code is quite confusing in that the variable name `weight` is used throughout the code. Nevertheless, the variable `weight` in the function `change_weight()` is different from the variable `weight` defined outside the function!

Listing 4.9 Another demonstration of the local scope of variables within a function.

```
1 >>> weight = 160                    # Set initial weight to 160.
2 >>> def change_weight(weight):      # Define function to change weight.
3 ...     weight = weight - 10
4 ...     print("weight is now", weight)
5 ...
6 >>> change_weight(weight)           # Claim is weight is now 150.
7 weight is now 150
8 >>> print(weight)                   # Check on weight.  Still 160!
9 160
```

In line 1 `weight` is set to `160`. In lines 2 through 4 a function is defined whose name implies it will change the `weight`. The function is invoked in line 6. The output in line 7 shows that `weight` is now 150, i.e., 10 less than the value that was passed to the function. *However*, one needs to realize that the variable `weight` within the `change_weight()` function is different from the variable `weight` that lives outside the function. The proof of this is in lines 8 and 9 where we see the value of `weight` outside the function is unchanged from its initial value, i.e., it is still `160`.

So, how can one employ a function to change the value of a variable? To change the value of a `float`, `int`, or `str` variable, you must assign a new value to the variable. This assignment must be done within the desired scope. To use a function to change a variable, the function should return the desired value and then this should be assigned to the variable. The code in Listing 4.10 illustrates this (note that the `change_weight()` function defined in Listing 4.10 is different from the one defined in Listing 4.9).

Listing 4.10 Demonstration of a way in which a function can be used to change the value of an `int` variable. Here one must assign the return value of the function back to the variable.

```
>>> weight = 160                  # Initialize weight to 160.
>>> def change_weight(weight): # Define function to return new weight.
...     return weight - 10
...
>>> weight = change_weight(weight) # Assign return value to weight.
>>> print(weight)                  # Confirm that weight has changed.
150
```

4.5 Scope of Functions

As described in Secs. 4.2 and 4.3, a `def` statement is used to create a function. So far, all the functions we have created have been defined outside the body of any other function. Defining a function in this way gives the function *global scope* meaning it is visible everywhere—we can call a function with global scope from within another function or call it while outside of any function (i.e., directly from the interactive prompt).

However, it is also possible to define a function within the body of another. When this is done, the inner function can only be invoked from within the function in which it was defined.[7] Thus the scope of the inner function is local to the outer function. This is demonstrated in Listing 4.11 where the function `g0()` is defined in the body of `f0()`. `g0()` returns the cube of its argument. `f0()` passes its argument to `g0()`, raises that to the fourth power, adds `4`, and returns the resulting value.

Listing 4.11 Demonstration that one function can be defined within another. However, when this is done, the scope of the inner function is restricted to the outer function in which it was defined.

```
>>> def f0(x):
...     def g0(y):
...         return y ** 3 + 3
...     return g0(x) ** 4 + 4
...
>>> f0(4)
20151125
>>> g0(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g0' is not defined
>>> def f1(x):
...     return g1(x) ** 4 + 4
...
>>> def g1(x):
...     return x ** 3 + 3
...
```

[7] Also, the function can only be called *after* it is defined.

```
18 >>> f1(4)
19 20151125
20 >>> g1(4)
21 67
```

The f0 and g0() functions are defined in lines 1 through 4. The body of g0() consists of a single return statement (line 3). The body of f0() consists of the definition of g0() and then its return statement in line 4. Lines 6 and 7 show the result produced by calling f0() with an argument of 4.

In line 8 an attempt is made to call g0(). This produces a NameError with the statement that g0() is not defined. Again, the scope of g0() is restricted to f0() and thus there is no other way to access this function. There are situations in which this sort of *encapsulation* is considered advantageous. Encapsulation is the practice of restricting the scope of programming elements, such as variables and functions, so that these elements are accessible only where they are needed. If a function, such as g0(), is only needed within another function, such as f0(), then this type of nesting can be considered a good programming practice.

However, many languages do not allow the nesting of function definitions as Python does. Thus, we will generally restrict our function definitions to be in the *global scope*, external to any function. Lines 12 through 21 demonstrate an alternative way of implementing f0(). This alternative approach creates the functions f1() and g1() (which perform calculations equivalent to f0() and g0(), respectively). Since g1() is defined in the global scope, it can be called from within f1() but also external to f1() as line 20 demonstrates.

Ultimately the scoping rules for functions are no different than for variables: anything defined inside a function is local to that function. Variables and functions defined external to any function have global scope and are visible "everywhere."

4.6 print() vs. return

In some situations print() statements and return statements appear to behave the same way. *However,* print() *and* return *are fundamentally different and a failure to understand how they differ can lead to errors.* To illustrate this, consider the code in Listing 4.12. Two functions are defined, p() and r(). Both take a single argument and both multiply this argument by 2. p() *prints* the result of this multiplication whereas r() *returns* this result.

Listing 4.12 Demonstration of the difference between a function that prints a value and one that returns the same value.

```
1 >>> def p(x):           # p() prints a result
2 ...     print(2 * x)
3 ...
4 >>> def r(x):           # r() returns a result
5 ...     return 2 * x
6 ...
7 >>> p(10)
```

```
8  20
9  >>> r(10)
10 20
11 >>> p_result = p(10)   # Use p() in assignment operation.  Note output.
12 20
13 >>> r_result = r(10)   # Use r() in assignment operation.  No output!
14 >>> p_result, r_result
15 (None, 20)
16 >>> p(10) + 7
17 20
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20 TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
21 >>> r(10) + 7
22 27
```

After defining the two function, `p()` is called in line 7 with an argument of `10`. On line 8 we see the value that is *printed* by this function, i.e., 20. Since `p()` doesn't explicitly return a value, it is a void function.

In line 9 `r()` is called, also with an argument of `10`. In this case `r()` *returns* the result, which is also `20`. We see the result on line 10 *only* because the interactive environment displays the value of an expression if the expression is the only thing on the line. To further illustrate this point, consider the next few lines.

An assignment statement appears in line 11 where `p()` is to the right of the assignment operator and `p_result` is the variable to the left. Note that `20` appears on line 12. This is because when the `p()` function is called, it prints two times its argument. So, what is `p_result`? Knowing that `p()` is a void function provides the answer (but the answer also appears later in Listing 4.12 itself).

Another assignment statement is given in line 13. In this case `r()` appears to the right of the operator and `r_result` is to the left. Note that, unlike when `p()` was used in a similar statement, this statement produces no output! This is a consequence of the fact that `r()` does not print anything. The value this function returns is simply assigned to `r_result` and the interactive environment doesn't display anything for such statements. Line 14 displays the values of `p_result` and `r_result`, which are `None` and `20`, respectively.

Look closely at the statement in line 16 and the subsequent result. This statement is the sum of `p(10)` and `7`. To calculate this sum, Python must first evaluate `p(10)`. Thus the `p()` function is called with an argument of `10`. Since the body of `p()` contains a `print()` statement, we see the output of that `print()` statement in line 17. Since `p()` is a void function, line 16 is ultimately equivalent to: `None + 7`. An integer cannot be added to `None` and hence this produces an error as the text in lines 18 through 20 shows. Finally, line 21 shows that `r(10)` can be used in a sum since it *returns* an integer.

The code in Listing 4.12 might leave you with the impression that `print()` statements cannot appear together with a `return` statement in a function, but this is *not* the case. For debugging purposes `print()` statements are often the quickest and easiest way to ascertain where a problem lies. Thus, when developing your code, it can often be useful to "sprinkle" your code with

`print()` statements that enable you to check that values are what they should be. There are also many other circumstances where a function should have one or more `print()` statements *and* a `return` statement.

As an example of mixing `print()` and `return` statements, consider the code shown in Listing 4.13. A function `f()` is defined that calculates a value, prints that value, and then returns it.

Listing 4.13 Demonstration that `print()` and `return` statements can appear in the same function.

```
1  >>> def f(x):
2  ...     y = 2 * (x - 3) // 14
3  ...     print("f(", x ,") = ", y, sep="")
4  ...     return y
5  ...
6  >>> z = (7 + f(94)) * 2
7  f(94) = 13
8  >>> z
9  40
```

In line 6 the `f()` function is used in an expression. When this function is evaluated, as part of that evaluation, the `print()` statement in its body is executed. Thus, in line 7, the output we see is the output produced by the `print()` statement in `f()` (line 3 of the listing). The value that `f(94)` *returns* (which is `13`) is used in the arithmetic expression in line 6 and `z` is assigned the value `40`.

4.7 Using a `main()` Function

In some popular computer languages it is required that every program has a function called `main()`. This function tells the computer where to start execution. Since Python is a *scripted* language in which execution starts at the beginning of a file and subsequent statements are executed sequentially, there is no requirement for a `main()` function. Nevertheless, many programmers carry over this convention to Python—they write programs in which, other than function definitions themselves, nearly all statements in the program are contained within one function or another. The programmers create a `main()` function and often call this in the final line of code in the program. Thus, after all the functions have been defined, including `main()` itself, the `main()` function is called.

Listing 4.14 provides an example of a program that employs this type of construction. Here it is assumed this code is stored in a file (hence the interactive prompt is not shown). When this file is executed, for example, in an IDLE session, the last statement is a call to the `main()` function. Everything prior to that final invocation of `main()` is a function definition. `main()` serves to call the other three functions that are used to prompt for input, calculate a BMI, and display that BMI.

Listing 4.14 A BMI program that employs a main() function which indicates where computation starts. main() is invoked as the final statement of the program. (Since the definitions of the first three functions are unchanged from before, the comments and docstrings have been removed.)

```
1  def get_wh():
2      weight = float(input("Enter weight [pounds]: "))
3      height = float(input("Enter height [inches]: "))
4      return weight, height
5
6  def calc_bmi(weight, height):
7      return 703 * weight / (height * height)
8
9  def show_bmi(bmi):
10     print("Your body mass index is:", bmi)
11
12 def main():
13     w, h = get_wh()
14     bmi = calc_bmi(w, h)
15     show_bmi(bmi)
16
17 main()  # Start the calculation.
```

In an IDLE session, after this file has been run and one BMI calculation has been performed, you can perform any number of subsequent BMI calculations simply by typing main() at the interactive prompt.

4.8 Optional Parameters

Python provides many built-in functions. The first function we introduced in this book was the built-in function print(). The print() function possesses a couple of interesting features that we don't yet know how to incorporate into our own functions: print() can take a variable number of parameters, and it also accepts optional parameters. The optional parameters are sep and end which specify the string used to separate arguments and what should appear at the end of the output, respectively. To demonstrate this, consider the code shown in Listing 4.15.

Listing 4.15 Demonstration of the use of the sep and end optional parameters for print().

```
1 >>> # Separator defaults to a blank space.
2 >>> print("Hello", "World")
3 Hello World
4 >>> # Explicitly set separator to string "-*-".
5 >>> print("Hello", "World", sep="-*-")
6 Hello-*-World
7 >>> # Issue two separate print() statements.  (Multiple statements can
8 >>> # appear on a line if they are separated by semicolons.)
```

```
 9 >>> # By default, print() terminates its output with a newline.
10 >>> print("Why,", "Hello"); print("World!")
11 Why, Hello
12 World!
13 >>> # Override the default separator and line terminator with the
14 >>> # optional arguments of sep and end.
15 >>> print("Why,", "Hello", sep="-*-", end="^v^"); print("World!")
16 Why,-*-Hello^v^World!
```

In line 2 `print()` is called with two arguments. Since no optional arguments are given, the subsequent output has a blank space separating the arguments and the output is terminated with a newline. In line 5 the optional argument is set to `-*-` which then appears between the arguments in the output, as shown in line 6. In line 10 two `print()` statements are given (recall that multiple statements can appear on a single line if they are separated by semicolons). The output from each of these statements is terminated with the default newline characters as shown implicitly in the subsequent output in lines 11 and 12. Line 15 again contains two `print()` statements but the first `print()` statement uses the optional parameters to set the separator and line terminator to the strings `-*-` and `^v^`, respectively.

As explained earlier in this chapter, we create functions in Python using a `def` statement. In the header, enclosed in parentheses, we include the list of formal parameters for the function. Python provides several different constructs for specifying how the parameters of a function are handled. We can, in fact, define functions of our own which accept an arbitrary number of arguments and employ optional arguments. Exploring all the different constructs can be a lengthy endeavor and the use of multiple arguments isn't currently of interest. However, creating functions with optional arguments is both simple and useful. Thus, let's consider how one defines a function with optional parameters.

First, as shown in Listing 4.16, let's define a function without optional parameters which squares its argument. The function is defined in lines 1 and 2 and then, in line 3, invoked with an argument of `10`.

Listing 4.16 Simple function to square its argument. Here the function is invoked by passing it the actual parameter which is the literal `10`.

```
1 >>> def square(x):
2 ...     return x * x
3 ...
4 >>> square(10)
5 100
```

Recall that when we write `square(10)`, the *actual* parameter is `10`. The actual parameter is assigned to the formal parameter and then the body of the function is executed. So, in this example, where we have `square(10)`, it is as if we had issued the statement `x=10` and then executed the body of the `square()` function.

If we so choose, we can explicitly establish the connection between the formal and actual parameters when we call a function. Consider the code in Listing 4.17 where the `square()`

function is defined exactly as above. Now, however, in line 4, when the function is invoked, the formal parameter `x` is explicitly set equal to the actual parameter `10`.

Listing 4.17 The same square function is defined as in Listing 4.16. However, in line 4, when the function is invoked, the formal parameter `x` is explicitly assigned the value of the actual parameter `10`.

```
1 >>> def square(x):
2 ...     return x * x
3 ...
4 >>> square(x=10)
5 100
```

Note carefully what appears in the argument when the `square()` function is called in line 4. We say that `x` is assigned `10`. This assignment is performed, the body of the function is then executed, and, finally, the returned value is `100` which is shown on line 5. In practice, for this simple function, there is really no reason to explicitly assign `10` to `x` in the invocation of the function. However, some functions have many arguments, some of which are optional and some of which are not. For these functions, explicitly assigning values to the parameters can aid readability (even when the parameters are required).

What happens if we use a different variable name (other than `x`) when we invoke the `square()` function? Listing 4.18 provides the answer.

Listing 4.18 The `square()` function is unchanged from the previous two listings. When the function is invoked in line 4, an attempt is made to assign a value to something that is not a formal parameter of the function. This produces an error.

```
1 >>> def square(x):
2 ...     return x * x
3 ...
4 >>> square(y=10)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: square() got an unexpected keyword argument 'y'
```

In line 4 we tried to assign the value `10` to the variable `y`. But, the `square()` function doesn't have any formal parameter named `y`, so this produces an error (as shown in lines 5 through 7). We have to use the formal parameter name that was used when the function was defined (in this case `x`).

Now, with this background out of the way: To create a function with one or more optional parameters, we simply assign a default value to the corresponding formal parameter(s) in the header of the function definition. An example helps to illustrate this.

Let's create a function called `power()` that raises a given number to some exponent. The user can call this function with either one or two arguments. The second argument is optional and corresponds to the exponent. If the exponent is not given explicitly, it is assumed to be `2`, i.e.,

the function will square the given value. The first argument is required (i.e., not optional) and represents the number that should be raised to the given exponent. The code in Listing 4.19 shows how to implement the `power()` function. Notice that in the header of the function definition (line 1), we simply assign a default value of `2` to the formal parameter `exponent`.

Listing 4.19 Function to raise a given number to an exponent. The exponent is an optional parameter. If the exponent is not explicitly given when the function is invoked, it defaults to `2` (i.e., the function returns the square of the number).

```
>>> def power(x, exponent=2):
...     return x ** exponent
...
>>> power(10)      # 10 squared, i.e., 10 ** 2.
100
>>> power(3)       # 3 squared, i.e., 3 ** 2.
9
>>> power(3, 0.5) # Square root of 3, i.e., 3 ** 0.5.
1.7320508075688772
>>> power(3, 4)    # 3 ** 4
81
>>> power(2, exponent=3)    # 2 ** 3
8
>>> power(x=3, exponent=3) # 3 ** 3
27
>>> power(exponent=3, x=5) # 5 ** 3
125
>>> power(exponent=3, 5)    # Error!
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Lines 4 through 7 show the argument is squared when `power()` is called with a single argument, i.e., the value of the argument is assigned to the formal parameter `x` and the default value of `2` is used for `exponent`. Lines 8 through 11 show what happens when `power()` is called with two arguments. The first argument is assigned to `x` and the second is assigned to `exponent`, thus overriding `exponent`'s default value of `2`. In lines 8 and 10, the assignment of actual parameters to formal parameters is based on position—the first actual parameter is assigned to the first formal parameter and the second actual parameter is assigned to the second formal parameter (this is no different than what you have previously observed with multi-parameter functions, such as `calc_bmi()` in Listing 4.7). So, for example, based on the call in line 8, `3` is assigned to `x` and `0.5` is assigned to `exponent` (which yields $\sqrt{3}$).

In line 12 we see that the function can be called with the optional parameter explicitly "named" in an assignment statement (thus optional parameters are sometimes called *named parameters*). Line 14 shows that, in fact, both parameters can be named when the function is called. Keep in mind, however, that `x` is *not* an optional parameter—a value must be provided for `x`. If one names all the parameters, then the order in which the parameters appear is not important. This is illustrated in line 16 where the optional parameter appears first and the required parameter appears

second (here the function calculates 5^3). Finally, line 18 shows that we cannot put an optional parameter before an unnamed required parameter.

Let us consider one more example in which a function calculates the y value of a straight line. Recall that the general equation for a line is

$$y = mx + b$$

where x is the independent variable, y is the dependent variable, m is the slope, and b is the intercept (i.e., the value at which the line crosses the y axis). Let's write a function called `line()` that, in general, has three arguments corresponding to x, m, and b. The slope and intercept will be optional parameters, and the slope will have a default value of 1 and the intercept a default value of 0. Listing 4.20 illustrates both the construction and use of this function.

Listing 4.20 Function to calculate the y value for a straight line given by $y = mx + b$. The value of x must be given. However, m and b are optional with default values of 1 and 0, respectively.

```
1  >>> def line(x, m=1, b=0):
2  ...     return m * x + b
3  ...
4  >>> line(10)         # x=10 and defaults of m=1 and b=0.
5  10
6  >>> line(10, 3)      # x=10, m=3, and default of b=0.
7  30
8  >>> line(10, 3, 4)   # x=10, m=3, b=4.
9  34
10 >>> line(10, b=4)    # x=10, b=4, and default of m=1.
11 14
12 >>> line(10, m=7)    # x=10, m=7, and default of b=0.
13 70
```

In lines 4, 6, and 8, the function is called with one, two, and three arguments, respectively. In these three calls the arguments are not named; hence the assignment to the formal parameters is based solely on position. When there is a single (actual) argument, this is assigned to the formal argument `x` while `m` and `b` take on the defaults values of `1` and `0`, respectively. When there are two unnamed (actual) arguments, as in line 6, they are assigned to the first two formal parameters, i.e., `x` and `m`. However, if one of the arguments is named, as in lines 10 and 12, the assignment of actual parameters to formal parameters is no longer dictated by order.

In line 10, the first argument (`10`) is assigned to `x`. The second argument dictates that the formal parameter `b` is assigned a value of `4`. Since nothing has been specified for the value of `m`, it is assigned the default value.

4.9 Chapter Summary

The template for defining a function is:

```
def <function_name>(<params>):
    <body>
```

where the function name is a valid identifier, the *formal parameters* are a comma-separated list of variables, and the body consists of an arbitrary number of statements that are indented to the same level.

A function is called/invoked by writing the function name followed by parentheses that enclose the *actual parameters* which are also known as the *arguments*.

A function that does not explicitly return a value is said to be a *void function*. Void functions return `None`.

A variable defined as a formal parameter or defined in the body of the function is not defined outside the function, i.e., the variables only have *local scope*. Variables accessible throughout a program are said to have *global scope*.

Generally, a function should obtain data via its parameters and return data via a **`return`** statement.

`print()` does not return anything (it generates output) and the `return` statement does not print anything (it serves to return a value).

If comma-separated expressions are given as part of the `return` statement, the values of these expressions are returned as a collection of values that can be used with simultaneous assignment. The values are in a `tuple` as described in Chap. 6.

Function definitions may be nested inside other functions. When this is done, the inner function is only usable within the body of the function in which it is defined. Typically such nesting is not used.

The scoping rules for functions are the same as for variables. Anything defined inside a function, including other functions, is local to that function. Variables and functions defined external to functions have global scope and are visible "everywhere."

Often programs are organized completely in terms of functions. A function named **`main()`** is, by convention, often the first function called at the start of a program (but after defining all the functions). The statements in `main()` provide the other function calls that are necessary to complete the program. Thus, the program consists of a number of function definitions and the last line of the program file is a call to `main()`.

An optional parameter it created by assigning a default value to the formal parameter in the header of the function definition.

4.10 Review Questions

1. The following code is executed

```
def f(x):
    return x + 2, x * 2

x, y = f(5)
print(x + y)
```

What is the output produced by the `print()` statement?

(a) `7 10`
(b) `17`
(c) `x + y`
(d) This produces an error.
(e) None of the above.

2. True or False: Names that are valid for variables are also valid for functions.

3. What output is produced by the `print()` statement when the following code is executed?

```
def calc_q1(x):
    q = 4 * x + 1
    return q

calc_q1(5)
print(q)
```

(a) `24`
(b) `21`
(c) `q`
(d) This produces an error.
(e) None of the above.

4. What is the value of `q` after the following code has been executed?

```
def calc_q2(x):
    q = 4 * x + 1
    print(q)

q = calc_q2(5)
```

(a) `24`
(b) `21`
(c) This produces an error.
(d) None of the above.

5. What is the value of `q` after the following code has been executed?

```
q = 20
def calc_q3(x):
    q = 4 * x + 1
    return q

q = calc_q3(5)
```

(a) `24`
(b) `21`
(c) This produces an error.
(d) None of the above.

6. What is the output produced by the `print()` statement in the following code?

```
def calc_q4(x):
    q = 4 * x + 1

print(calc_q4(5))
```

(a) `24`
(b) `21`
(c) `q`
(d) This produces an error.
(e) None of the above.

7. What is the output of the `print()` statement in the following code?

```
abc = 5 + 6 // 12
print(abc)
```

(a) This produces an error.
(b) `5 + 6 // 12`
(c) `5`
(d) `5.5`
(e) `6`

8. What is the output of the `print()` statement in the following code?

```
def = 5 + 6 % 7
print(def)
```

(a) This produces an error.
(b) `5 + 6 % 7`
(c) `11`
(d) `4`

9. The following code is executed:

```
def get_input():
    x = float(input("Enter a number: "))
    return x

def main():
    get_input()
    print(x ** 2)

main()
```

At the prompt the user enters `2`. What is the output of this program?

(a) `x ** 2`
(b) `4`
(c) `4.0`
(d) This produces an error.
(e) None of the above.

10. The following code is executed:

```
def get_input():
    x = float(input("Enter a number: "))
    return x

def main():
    print(get_input() ** 2)

main()
```

At the prompt the user enters `2`. What is the output of this program?

(a) `get_input() ** 2`
(b) `4`
(c) `4.0`
(d) This produces an error.
(e) None of the above.

11. What is the value of `z` after the following code is executed?

```
def f1(x, y):
    print((x + 1) / (y - 1))

z = f1(3, 3) + 1
```

(a) 3
(b) 3.0
(c) 2
(d) This produces an error.

12. What is the value of z after the following code is executed?

```
def f2(x, y):
    return (x + 1) / (y - 1)

z = f2(3, 3) + 1
```

(a) 3
(b) 3.0
(c) 2
(d) This produces an error.
(e) None of the above.

13. What is the value of z after the following code is executed?

```
def f3(x, y = 2):
    return (x + 1) / (y - 1)

z = f3(3, 3) + 1
```

(a) 3
(b) 3.0
(c) 2
(d) This produces an error.
(e) None of the above.

14. What is the value of z after the following code is executed?

```
def f3(x, y = 2):
    return (x + 1) / (y - 1)

z = f3(3) + 1
```

(a) 3
(b) 3.0
(c) 2

(d) This produces an error.

(e) None of the above.

15. The following code is executed.

```
def inc_by_two(x):
    x = x + 2
    return x

x = 10
inc_by_two(x)
print("x = ", x)
```

What is the output produced by the `print()` statement?

ANSWERS: 1) b; 2) True; 3) d, `q` is not defined outside of the function; 4) d, the function is a void function and hence `q` is `None`; 5) b; 6) e, the output is `None` since this is a void function; 7) c; 8) a, `def` is a keyword and we cannot assign a value to it; 9) d, the variable `x` is not defined in `main()`; 10) c; 11) d, `f1()` is a void function; 12) b; 13) b; 14) e, `z` would be assigned `5.0`; 15) `x = 10`.

4.11 Exercises

1. Write a function called `convert_to_days()` that takes no parameters. Have your function prompt the user to input numbers of hours, minutes, and seconds. Write a helper function called `get_days()` that uses these values and converts them to days in `float` form (fractions of a day are allowed). `get_days()` should return the number of days. Use this helper function within the `convert_to_days()` function to display the numbers of days to the user. The built-in function `round()` takes two arguments: a number and an integer indicating the desired precision (i.e., the desired number of digits beyond the decimal point). Use this function to round the number of days four digits after the decimal point.

   The following demonstrates the proper behavior of `convert_to_days()`:

```
1 >>> convert_to_days()
2 Enter number of hours: 97
3 Enter number of minutes: 54
4 Enter number of seconds: 45
5
6 The number of days is: 4.0797
```

2. Write a function called `calc_weight_on_planet()` that calculates your equivalent weight on another planet. This function takes two arguments: your weight on Earth in pounds and the surface gravity of the planet of interest with units m/s^2. Make the second argument optional and supply a default value of 23.1 m/s^2 which is the approximate surface gravity of Jupiter (Earth's suface gravity is approximately 9.8 m/s^2). To perform the conversion, use

the equation: weight is equal to mass times suface gravity. Since your weight on Earth is given and you know the Earth's surface gravity, have your function use this information to calculate your mass (it is fine if, at this point, the units of mass are a mix of Imperial and the MKS system). Then, use your mass and the given surface gravity to calculate your effective weight on the other planet.

The following demonstrates the proper behavior of this function:

```
>>> calc_weight_on_planet(120, 9.8)
120.0
>>> calc_weight_on_planet(120)
282.85714285714283
>>> calc_weight_on_planet(120, 23.1)
282.85714285714283
```

3. Write a function called `num_atoms()` that calculates how many atoms are in n grams of an element given its atomic weight. This function should take two parameters: the amount of the element in grams and an optional argument representing the atomic weight of the element. The atomic weight of any particular element can be found on a periodic table but make the default value for the optional argument the atomic weight of gold (Au) 196.97 with units in grams/mole. A mole is a unit of measurement that is commonly used in chemistry to express an amount of a substance. Avogadro's number is a constant, 6.022×10^{23} atoms/mole, that can be used to find the number of atoms in a given sample. Use Avogadro's number, the atomic weight, and the amount of the element in grams to find the number of atoms present in the sample. Your function should return this value.

   The following demonstrates the proper behavior of this function using 10 grams and the atomic weight of gold (default), carbon, and hydrogen:

```
>>> num_atoms(10)
3.0573183733563486e+22
>>> num_atoms(10, 12.001)
5.017915173735522e+23
>>> num_atoms(10, 1.008)
5.97420634920635e+24
```

4. The aspect ratio of an image describes the relationship between the width and height. Aspect ratios are usually expressed as two numbers separated by a colon that represent width and height respectively. Common aspect ratios in photography are 4:3, 3:2, and 16:9. An image that has an aspect ratio of $x : y$ means that for every x inches of width you will have y inches of height no matter the size of the image (and, of course, you can use any unit of length, not just inches, and even abstract units such as pixels). Suppose you are writing a blog and you have an image that is m units wide and n units high but your blog only has space for an image that is z units wide (where z is less than m). Write a function called `calc_new_height()` that returns the height the image must be in order to preserve the aspect ratio (i.e., a height that will not distort the image). This function takes no arguments and prompts the user for the current width, the current height, and the desired new width. In addition to *returning* the

new height, this function also prints the value. The new height is a `float` regardless of the types of the values the user entered.

The following demonstrates the proper behavior of this function:

```
>>> calc_new_height()
Enter the current width: 800
Enter the current height: 560
Enter the desired width: 370

The corresponding height is: 259.0
259.0
```

5. Write a function called `convert_temp()` that takes no arguments. This function obtains a temperature in Fahrenheit from the user and uses two helper functions to convert this temperature to Celsius and Kelvin. Write a helper function called `convert_to_celsius()` that takes a single argument in Fahrenheit and returns the temperature in Celsius using the formula

$$T_c = \frac{5}{9}(T_f - 32).$$

where T_c is the temperature in Celsius and T_f is the temperature in Fahrenheit. Write another helper function called `convert_to_kelvin()` that takes a single argument in degrees Celsius and returns degrees Kelvin using the formula

$$T_k = T_c + 273.15.$$

where T_k is the temperature in Kelvin. Use these two functions within your `convert_temp()` function to display (i.e., print) the temperatures for the user. The `convert_temp()` does not return anything.

The following demonstrates the proper behavior of this function:

```
>>> convert_temp()
Enter a temperature in Fahrenheit: 32

The temperature in Fahrenheit is: 32
The temperature in Celsius is: 0.0
The temperature in Kelvin is: 273.15
>>> convert_temp()
Enter a temperature in Fahrenheit: 80

The temperature in Fahrenheit is: 80
The temperature in Celsius is: 26.666666666666668
The temperature in Kelvin is: 299.81666666666666
```

Chapter 5

Introduction to Objects

In the coming chapters, we want to unleash some of the power of Python. To access this power we have to introduce something that is syntactically different from what we have seen so far. In order to understand the reason for this change in syntax, it is best to have a simple understanding of *objects* and *object oriented programming*.

Computer languages, considered as a whole, differ widely in how they allow programmers to construct a program. There are some languages that permit object oriented programming, often abbreviated OOP. Python is one such language (C++, Java, and C# are other popular OOP languages). As will be described, OOP provides a way of organizing programs that is more similar to the way people think (i.e., more so than if one uses a non-OOP approach). Thus, in many instances an OOP approach provides the clearest and simplest route to a solution. However, as you might imagine, when something attempts to reflect the way humans think, there can be many subtleties and several layers of complexity. We do *not* want to explore any of these subtleties and complexities here.

At the conclusion of this chapter you should have a sufficiently clear understanding of a couple of aspects of OOP—to the point where you should be comfortable with the material in chapters that follow. But, there is no expectation that this chapter will enable you to write your own programs that fully exploit the power of OOP. (So, if some questions about OOP linger at the end of this chapter, it is to be expected and should not be a concern. At the end of the chapter we will revisit the "take away" message of this material.)

5.1 Overview of Objects and Classes

In previous chapters we considered data such as strings, integers, and `float`s. We also considered functions that can accept data via a parameter (or parameters) and can return data via a `return` statement. Functions are created to accomplish particular tasks and are invariably restricted in the type of data they can accept. For example, if we write a function to capitalize the characters in a string, it doesn't make sense to pass numeric data to this function. As another example, if a function calculates the absolute value, it doesn't make sense to pass this function a string.

When you pause to think about data and functions, you quickly realize they are related in some way. The fact that only certain forms of data make sense with certain functions ties data and functions together conceptually. In OOP there is still the ability to work with traditional data and

From the file: `object-basics.tex`

functions, but OOP introduces an additional construct known as an *object*. An object is a collection of data *and* functions (we'll see how to create the collection in a moment). Using objects we can more closely associate functions with the type of data on which the functions can legitimately operate. To distinguish traditional data from the data contained in an object, we call the data in an object the object's *attributes*. And, to distinguish traditional functions from the functions contained in an object, we call the functions in an object the object's *methods*. So, instead of saying an object is a collection of data and functions, it is more proper to say an object is a collection of attributes and methods.[1]

In Python, as we shall see, essentially everything is an object! The data we have seen so far are actually objects of a given type (such as `str`, `int`, or `float`). Functions that we create are objects of a different type. Built-in functions are objects of yet another type. And so on. Replacing the word "type" with the word "class" in these sentences leads us to the slightly more technical statement: All objects are members of a *class*.

What does it mean to say all objects are members of a class? This is actually closely aligned with the way we naturally think of the world and the objects in it. In the real world, we automatically think in terms of "groups" of "things." Yes, you are unique, but you are still a member of the group of humans. We may say you are a unique *instance* of a human, but you still belong to the group of humans. Your clothing may be one-of-a-kind; perhaps you made your favorite dress yourself. Nevertheless, that dress is still a member of, or instance of, the group of dresses.

The ability to group things allows us to make sense of the world more easily than could be done without these associations. Knowing a thing is a member of a group allows us to make various assumptions about that thing. For example, if we know a "thing" is a human, we know he or she shares attributes with all other members of the group of humans, such as blood type, hair color, gender, and visual acuity. (Although all members of the group share the same attributes, an individual has specific values for these attributes that are his or her own, such as a blood type of AB positive, black hair, female, and 20/15 vision.) Knowing something is a car, we know it has a different set of attributes, such as number of seats, headroom, and miles per gallon. Dresses have attributes such as type of fabric, size, sleeve length, etc. This grouping also gives us information about the way in which something functions or interacts with the world. We wouldn't expect a human to behave in the same way as a car or a dress. Things in the human, car, and dress groups all have their own ways of interacting with other things.

If we go back to the previous two paragraphs and replace the word "thing" with "object" and "group" with "class," we have the start of viewing the world in an OOP way. Instead of "all things are members of a group" we have "all objects are members of a class." A class defines what the attributes are that members of the class share. However, when it comes to an individual object (i.e., an instance of a class), the values of these attributes are those of that particular object. Again, even though people share common attributes, the values of attributes such as blood type, hair color, gender, visual acuity, etc., can vary significantly from one individual to the next.

With that said, in the next section we show how to define a class (a "group") and create an instance of the class (a "thing") in Python.

[1]One thing that can be somewhat confusing about OOP is that different languages use different terms for the same underlying concept. For example, what we are calling a method in Python, would be called a *member function* in C++. Also, different authors may use different terms to describe the same things even in the same language!

5.2 Creating a Class: Attributes

In OOP, the programmer uses a *class statement* to provide something of a blueprint for the attributes an object has (we'll turn our attention to an object's methods in the next section). Let's use a slightly contrived scenario to help illustrate the creation of a class. Assume a programmer works for a hospital and needs to develop a program that deals with patients. These patients have only three attributes: name, age, and malady. To tell Python what constitutes a patient, the programmer creates a class with a statement that follows the template shown in Listing 5.1.[2]

Listing 5.1 Basic template for creating a class.

```
1 class <ClassName>:
2     <body>
```

Line 1 provides the class header which starts with the keyword `class`. This is followed by *<ClassName>* which is the name of the class and can be any valid identifier.[3] The header is terminated with a colon. The header is followed by the *<body>* which provides statements that specify the attributes and methods in the class (we defer consideration of methods to the next section). As was the case for a function definition, the *<body>* must be indented. The body can consist of any number of statements and is terminated by halting indentation of the code (or by entering a blank line in the interactive environment or in IDLE).

Although there are various ways to implement a `Patient` class, a programmer might use the `class` statement shown in lines 1 through 4 of Listing 5.2.[4] This statement creates the attributes `name`, `age`, and `malady` simply by using the assignment operator to assign default values to these identifiers. Keep in mind that this `class` statement itself does *not* actually create a `Patient`. Rather, it tells us what attributes a `Patient` has. In some ways this is similar to what we do with functions. We define a function with a `def` statement, but this doesn't invoke the function. After the function is defined, we are free to call it as needed. The creation of a class is somewhat similar. After we create a class with a `class` statement, we are free to call the class to create as many objects (of that class) as needed. (Creating an object is often called *instantiation* and we will use the terms object and *instance* synonymously.) The rest of the code in Listing 5.2 is discussed in more detail following the listing.

Listing 5.2 `class` statement that defines the `Patient` class where a `Patient` has a `name`, `age`, and `malady`. Each of these attributes is given a default value.

```
1 >>> class Patient:
2 ...     name = "Jane Doe"
3 ...     age = 0
```

[2] A more complicated `class` statement is available that allows one class to inherit properties from another, but we have no need for it here.

[3] The class identifier is usually written using the "CapWords" convention where the first letter of each word is capitalized and, of course, there are no spaces between the words (since there can be no spaces in an identifier).

[4] A construct like this would almost certainly *not* be used in practice, but this implementation is useful for the sake of illustration. In subsequent examples in this chapter we provide improved `class` statements.

```
4  ...     malady = "healthy"
5  ...
6  >>> # Create an instance of a Patient with identifier of sally.
7  >>> sally = Patient()
8  >>> # Show sally's name attribute.
9  >>> sally.name
10 'Jane Doe'
11 >>> # Set the name, age, and malady attributes for sally.
12 >>> sally.name = "Sally Smith"
13 >>> sally.age = 21
14 >>> sally.malady = "bruised ego"
15 >>> # Show sally's name, age, and malady attributes.
16 >>> print(sally.name, sally.age, sally.malady, sep="; ")
17 Sally Smith; 21; bruised ego
18 >>> def show(patient):
19 ...     print(patient.name, patient.age, patient.malady, sep="; ")
20 ...
21 >>> show(sally)
22 Sally Smith; 21; bruised ego
```

Following the `class` statement, a `Patient` is created in line 7 and assigned to the identifier `sally`. `sally` is simply a variable, but of a new data type, i.e., a `Patient`. We say that `sally` *is* a `Patient` or `sally` is an instance of the `Patient` class. In general, to create an instance of a class, we provide the class name followed by parentheses (similar to how we call a function—later we will see what one might put in the parentheses). Thus, the right side of line 7 instructs Python to create a `Patient` and the assignment operator associates this `patient` with the identifier `sally`.

In order to access attributes of an object, we use what is sometimes called *dot notation*: we write the variable name then a "dot" and then the attribute.[5] This is illustrated in line 9 where we access `sally`'s `name` attribute. More technically, in this context the period (or dot) is actually serving as the *access attribute operator* (which we may refer to as either the access operator or the attribute operator).

We can access an attribute either to see what it is, as is done in line 9, or to assign a value to it, as is done in lines 12 through 14 where new values are assigned to `name`, `age`, and `malady`. The output from the `print()` statement in line 16 shows that all of `sally`'s attributes have been changed from the default values.

If we want to display several `Patients`, it could be cumbersome to write several `print()` statements such as in line 16. To help streamline displaying `Patients`, we could write a function as shown in lines 18 and 19. The `show()` function takes a `Patient` as a parameter and then displays the `Patient`'s attributes. Note that the variable name used for the formal parameter, `patient`, can be any valid identifier, but `patient` is nicely descriptive of the fact that this function should be passed a `Patient` object as the actual parameter. Lines 21 and 22 show what happens when `sally` is passed to this function.

[5]This dot notation actually works with literals too as will be seen later.

The show() function defined in lines 18 and 19 is quite specialized—it is only designed to work with Patients. Thus, it would be more logical if this function were bundled with the Patient class. Again, when a function is incorporated into a class, it is known as a method. We consider how to create a method in the next section.

Referring to Listing 5.2, with the introduction of the access attribute operator, it is tempting to think that something like sally.name is a valid identifier. However, *it is not.* The rules that dictate what constitutes a valid identifier have not changed from before (see Sec. 2.6). Thus, sally is a valid identifier: it is a Patient object. And, name is a valid identifier: it is an attribute of the Patient object. However, sally.name is not a valid identifier. This does not imply that sally.name is not a legitimate lvalue. In fact, it is a valid lvalue—we can assign values to it. But, we cannot, for example, use sally.name as the name of a function or the name of an integer variable.

5.3 Creating a Class: Methods

In lines 18 and 19 of Listing 5.2 a function is defined to show a Patient's attributes. Although this function only works for Patients, it is not truly affiliated with the Patient class—the function stands on its own. We could, perhaps, accidentally call the show() function with a string as the parameter. If we did, an error would occur (since strings don't have attributes of name, age, and malady).

Instead of defining a function such that it exists outside the class, we can include a function as a method within the class if we move the associated def statement into the class statement. By doing this, we can ensure the method only operates on the objects for which it is designed to operate. A method is almost exactly like a function but there are a couple of notable differences: one affects how we define a method and the other affects how we invoke it. Let us first consider the difference in invocation.

Similar to the show() function in Listing 5.2, assume we have written a *method* that shows the attributes of a Patient. To distinguish this from the show() function, let's call this method display(). In Listing 5.2, the *function* to show the object's attributes is invoked with an argument of sally using this statement:

```
show(sally)
```

Since display() is a method, where it is defined in the class statement, we invoke it using this statement:

```
sally.display()
```

We'll discuss this in more detail in a moment, but we want to state up front that this invocation *does* pass sally as a parameter to the display() method.[6] To invoke a method, we provide an instance of the class (which is sally here) and then, using dot notation, specify the method. The parentheses are necessary. sally "knows" her attributes but she also "knows" her methods; hence the dot notation allows us to access not only the attributes, but also the methods within sally's class.

[6]The fact that sally is passed as a parameter to the display() method is done implicitly when the method is called, but, as will be shown soon, is indicated explicitly in the method definition.

Notice that in the invocation of the method it appears that the method has no parameters. However, this is *not* the case. When working with methods, Python always passes the instance on which the method is invoked (`sally` in this example) as the first parameter of the method. When defining the method (with a `def` statement), the convention is to call this first (formal) parameter `self`. An example helps to clarify this.

In Listing 5.3 the `class` statement starts with the same attributes in Listing 5.2. For the sake of readability, line 5 is semi-blank (in that there is an indentation but nothing else on the line). This is followed by the definition of the method `display()` (which is slightly fancier than the `show()` function in Listing 5.2 but ultimately displays the same data).

The template for a method is the same as the template for a function as given in Listing 4.2. The one thing that must be kept in mind, though, is that Python always provides an instance of the class as the first parameter to the method (in this case the method only has a single parameter—later we will consider a method with multiple parameters). In some sense, although we might write `sally.display()`, Python effectively translates this to `display(sally)`. Thus, when we write `sally.display()`, line 6 instructs Python that the formal parameter `self` is assigned the actual parameter `sally` and then the body of the method is executed. There is nothing unique about the identifier `self`. Any valid identifier will work. Although it seems `patient` would be a more descriptive identifier in this particular case than `self`, the convention is to use `self` and we will follow this convention. Discussion of Listing 5.3 continues below the listing.

Listing 5.3 A class statement that includes a method definition. The first parameter of a method is always the object on which the method is invoked. Although any valid identifier can be used for this parameter, it is convention to call this first parameter `self`.

```
1  >>> class Patient:
2  ...     name = "Jane Doe"
3  ...     age = 0
4  ...     malady = "healthy"
5  ...
6  ...     def display(self):
7  ...         print("Name    = ", self.name)
8  ...         print("Age     = ", self.age)
9  ...         print("Malady  = ", self.malady)
10 ...
11 >>> samuel = Patient()
12 >>> samuel.name = "Samuel Sneed"
13 >>> samuel.age = 18
14 >>> samuel.malady = "broken heart"
15 >>> samuel.display()
16 Name    =  Samuel Sneed
17 Age     =  18
18 Malady  =  broken heart
```

After the close of the class statement, a `Patient` is created in line 11 and assigned to `samuel` (i.e., the variable `samuel` is an instance of the `Patient` class). In lines 12 and 13, new values

are assigned to samuel's name, age, and malady. Then, in line 15, the display() method is invoked.

5.4 The dir() Function

When we create a class, we create a new type of data. (We can see this if we use the type() function to check on the type of one of the Patient's we created above.) Given that all data in Python are actually objects of classes means that all data are collections of attributes and methods. This is indeed the case and the built-in function dir() provides a way to see this collection, albeit not necessarily in the prettiest form. You can think of dir() as providing a ***dir***ectory of sorts.

Listing 5.4 is a continuation of Listing 5.3 in which the Patient class is defined and samuel is an instance of this class. In line 19 we use the type() function to ask what the type is of the Patient class. In line 20 we see that a Patient's type is type! This tells us that a Patient is its own (new) form of data.

In line 21 we ask for samuel's type. We see that samuel is a Patient but we are also given some additional information. This additional information relates to where the class is defined, but this doesn't concern us now.[7] In lines 23 and 24, type() shows that samuel's age attribute is an integer. Lines 25 and 26 show that samuel.display is a method. The rest of this code is discussed following the listing.

Listing 5.4 Illustration that a class is a type of data (and hence a new class is a new type of data). Also, this code demonstrates the use of the dir() function. This is a continuation of Listing 5.3.

```
19 >>> type(Patient)
20 <class 'type'>
21 >>> type(samuel)
22 <class '__main__.Patient'>
23 >>> type(samuel.age)
24 <class 'int'>
25 >>> type(samuel.display)
26 <class 'method'>
27 >>> dir(samuel)
28 ['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
29   '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
30   '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
31   '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
32   '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age',
33   'malady', 'name', 'display']
```

In line 27 dir() is used to show all the attributes and methods associated with Patient. We are shown a collection of strings.[8] Notice the last four strings correspond to the attributes and the

[7]Since we used the interactive environment to define the Patient class, this class is said to have been created in __main__ which is Python's name for the "top-level script environment."

[8]This particular collection is known as a list and will be described in a later chapter.

method we put in the `Patient class` statement (these are shown in slanted bold type to help differentiate them from the other strings in the list).

All the other strings in this list are attributes or methods associated with the class, but we did not establish this connection. This is something Python did for us. We will mostly ignore these other entries, but there is one notable exception which is also shown in slanted bold type: `__init__`. This is a method that allows us to initialize an object when it is created. We will illustrate the use of this method in the next section. Before doing so, we consider another example of the use of the `dir()` function.

We now know that integers, `float`s, and strings are instances of classes. As a preview of what is discussed in detail in the chapter on strings, let's create a string and then use `dir()` to see its list of attributes and methods, as is done in Listing 5.5. In line 1 the string `s` is created. In line 2 the `dir()` function is applied to this string which reveals a rather lengthy list!

Listing 5.5 Using `dir()` to list the attributes and methods of a string.

```
1  >>> s = "hello!"
2  >>> dir(s)
3  ['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
4    '__eq__', '__format__', '__ge__', '__getattribute__',
5    '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
6    '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
7    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
8    '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
9    '__subclasshook__', 'capitalize', 'center', 'count', 'encode',
10   'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
11   'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
12   'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
13   'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
14   'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
15   'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
16   'swapcase', 'title', 'translate', 'upper', 'zfill']
17 >>> help(s.upper)
18 Help on built-in function upper:
19
20 upper(...)
21     S.upper() -> str
22
23     Return a copy of S converted to uppercase.
24
25 >>> s.upper()
26 'HELLO!'
```

One of the strings in the list returned by `dir()` is highlighted. This is the `upper()` method. We already know that `help()` can be used to provide information on various things, including built-in functions. Similarly, we can often use `help()` to obtain information about a class's methods. In line 17, we see how we can obtain help on the `upper()` method: We provide an instance, a

dot, and then the method of interest (without parentheses—if we included parentheses we would invoke the method). The output in lines 18 through 24 indicates that this method returns a copy of the string, but all in uppercase. Lines 25 and 26 show this works as advertised!

We will not provide further examples of the `dir()` function here, but it is worth mentioning that one doesn't have to create a variable first in order to see the list of attributes and methods associated with a class. One can obtain the same output shown in Listing 5.5 by writing, for instance, `dir(str)` (where the class name is the parameter) or `dir("hello!")` or even just `dir("")` (where the latter provides an empty string as the parameter). If you are curious about the attributes and methods for an integer or `float`, you can, for example, type `dir(42)` or `dir(1.0)`, respectively.

Before moving on to the next section of this chapter, we want to mention that methods can be applied to literals. This is illustrated in Listing 5.6. If you enter `dir(int)` you will see that one of the methods in the integer class is `bit_length()`. This returns the minimum number of binary digits (bits) needed to represent an integer. However, if we want to know the "bit length" of the number `42`, we cannot write `42.bit_length()` because "`42.`" looks like a `float`. Instead, we have to enclose the numeric literal in parentheses as shown below.

Listing 5.6 Demonstration that methods can be applied to literals.

```
1 >>> "hello!".upper()  # Convert "hello!" to uppercase.
2 'HELLO!'
3 >>> # Determine minimum number of bits to represent 42.
4 >>> (42).bit_length()
5 6
```

5.5 The `__init__()` Method

We now return to the `__init__()` method mentioned in connection with Listing 5.4.[9] (Depending on the display device you use to read this material, it may not be obvious, but this method starts and ends with *two* underscores.) If a programmer defines this method, it is called automatically when an object is created. Hence `__init__()` serves as the ***init***ialization method. Using this method we can set the values of an object's attributes when the object is created (this is in contrast to the examples in Figs. 5.2 and 5.3 where the objects are created first and then their attributes are set using additional statements). Any parameters given to the `class` are passed along to `__init__()`. This is best illustrated with an example.

In Listing 5.7 we have a new `Patient class` statement. It may appear we have eliminated the `name`, `age`, and `malady` attributes. However, we have just moved their creation to statements inside the `__init__()` method that starts on line 2. In this example the `__init__()` method takes four parameters. In general `__init__()` must have at least one parameter but could have arbitrarily more. We have no choice about the first parameter: Python dictates that it is the object under creation, i.e., it is the object itself. The other three parameters are used to specify the desired

[9] Technically `__init__()` is a "method wrapper" but as far as we are concerned it is indistinguishable from a method.

name, age, and malady. The values that are passed as parameters are assigned to the attributes of the object itself in lines 4 through 6. When we have a statement such as

```
    self.<attribute> = <value>
```

this automatically creates the attribute (if it didn't exist before) and assigns "<value>" to it. If the attribute previously existed, its old value is replaced with the new value to the right of the assignment operator.

Listing 5.7 A new implementation of the Patient class (previous implementations were given in Figs. 5.2 and 5.3). Here the __init__() method is used to initialize attributes when an object is created. The display() method is unchanged from Listing 5.3. A cure() method has also been added to the class.

```
1  >>> class Patient:
2  ...     def __init__(self, name, age, malady):
3  ...         """Initialize the Patient's name, age, and malady."""
4  ...         self.name   = name
5  ...         self.age    = age
6  ...         self.malady = malady
7  ...
8  ...     def display(self):
9  ...         """Display the Patient's attributes."""
10 ...         print("Name   = ", self.name)
11 ...         print("Age    = ", self.age)
12 ...         print("Malady = ", self.malady)
13 ...
14 ...     def cure(self):
15 ...         """Cure the Patient."""
16 ...         self.malady = "healthy"
17 ...
18 >>> sally = Patient("Sally Smith", 21, "bruised ego")
19 >>> sally.display()
20 Name   =  Sally Smith
21 Age    =  21
22 Malady =  bruised ego
23 >>> sally.cure()
24 >>> sally.display()
25 Name   =  Sally Smith
26 Age    =  21
27 Malady =  healthy
```

After defining the Patient class, we create the Patient sally in line 18. In this case, we set her name, age, and malady when she is created. Although we are not explicitly invoking the __init__() method, Python calls it for us. All the parameters given to Patient are passed along to __init__(). So, although it isn't strictly true, when we write

```
   sally = Patient("Sally Smith", 21, "bruised ego")
```

in many respects we can think of it as being equivalent to a function call that looks like

```
__init__(sally, "Sally Smith", 21, "bruised ego")
```

Line 19 uses the `display()` method to show `sally`'s attributes. In line 23 we use the new `cure()` method to bring `sally` back to a healthy state. Line 24 again uses `display()` to show that `sally` has now recovered.

5.6 Operator Overloading

If you are shown a plus sign and asked what it does or what it represents, the first answer that springs to mind will probably pertain to the addition of two numbers. But, when you think about it a bit more, you realize the plus sign can mean different things in different contexts. For example, if you see `+5.234`, the plus sign is not indicating a sum: rather, it is indicating `5.234` is a positive number. Graffiti that says "*pat + chris*" indicates the existence of a couple where there is a lot of room for further interpretation. So, we should recognize that not only is the plus sign a convenient and familiar symbol, it can mean different things in different contexts.

Although we haven't dwelt on it, we've already seen that symbols in computer languages can mean different things in different "contexts." We've seen that a plus sign can mean addition (with the need for two operands) but it can also indicate sign (with a single operand). We've seen that a minus sign can mean subtraction, but it can also indicate sign (as in `-5.234`) or produce a change in sign (as in `-x`).

We've seen that a plus sign can be put between two `int`s or two `float`s or between an `int` and a `float`. We typically think of these operations as simply the sum of two numbers, but because of the way numbers are represented in a computer and the implementation of the underlying hardware, summing two `int`s requires actions on the part of the computer that are different from the actions involved in summing two `float`s. Also, when we sum an `int` and `float`, the computer must first *promote* the `int` to a `float` and then perform the sum. In Python, when we place the plus sign between two operands, the interpreter is aware of the class to which the operands belong (or, said another way, the types of the operands). Thus, Python will do "the right thing" whether we are summing `int`s or `float`s or anything else. In some cases, "the right thing" is to report an error if, for example, we try to sum an `int` and a string.

When we say a symbol can mean different things in different contexts, we take it as understood that the "context" is determined by the associated operand or operands for the symbol. For example, when a plus sign appears between two integers, the context is one involving integers and the plus sign needs to be interpreted in a way that is appropriate for integers. When a plus sign appears between an integer and a `float`, the context is now one of mixed types and the plus sign must be interpreted in a way that is appropriate for these specific types. With OOP, it is typically possible to specify what a symbol means or does in a given context, e.g., the programmer can dictate what should be done when a plus sign is used in a particular context. Defining a new interpretation of a symbol and how it should behave in a given context is known as *operator overloading*. We won't consider how one implements operator overload, but we do want to be aware of its existence.[10]

[10] Although the details aren't import to us and hence won't be covered, we will say that when objects appear to the left and right of a plus sign, Python will call the `__add__()` method for the class of the object to the left of the operand and provide both objects as parameters to this method. So, for example, an expression such as "`obj1 + obj2`" is

We have seen (line 1 of Listing 3.2) that we cannot "add" a string and a numeric value. The fact that such an operation produces an error intuitively makes sense. But, can we "add" two strings? Can we "multiply" a string by an integer or by a `float`? At this point we can only guess the answers, but we can certainly try these operations and see what happens. The results, shown in Listing 5.8, may seem odd at first, but with a bit of explanation (which is provided below the listing) they make sense and demonstrate some of the power and flexibility of Python.

Listing 5.8 Demonstration of operator overloading.

```
1  >>> s1 = "Hello" + " World!" # Concatenation.
2  >>> print(s1)
3  Hello World!
4  >>> s2 = "&" * 70            # Repetition.
5  >>> print(s1)
6  &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
7  >>> 10 * "EI" + "-OH!"       # Repetition and concatenation.
8  'EIEIEIEIEIEIEIEIEIEI-OH!'
9  >>> s3 = "#" * 70.0
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 TypeError: can't multiply sequence by non-int of type 'float'
```

The expression on the right side of line 1 has a plus sign with two string operands. When Python sees a "sum" of two strings, the plus sign is interpreted as the *concatenation* operator, i.e., it produces a new string that is obtained by *adding* the second string to the end of the first string. (To concatenate things is to link them together.) The output of the `print()` statement in line 2 shows that `s1` is indeed the concatenation of the two strings.

The expression on the right side of line 4 has the multiplication symbol with operands of a string and an integer. In this case Python interprets the multiplication as implying repetition. So, in this case the variable `s2` is assigned the string value consisting of `&` repeated 70 times. Lines 7 and 8 show another example of the repetition and concatenation operators.

Line 9 shows Python does not allow multiplication of a string and `float`. It makes sense to repeat a string an integer number of times but it generally doesn't make sense to repeat a string a fractional amount. Thus, Python requires that repetition be expressed as the product of a string and an `int`.

In the coming chapters we exploit the operator overloading already built into Python. This overloading is part of the power of OOP—the language is aware of the classes of the operands and will do what is appropriate for the given combination of operands and symbols.

5.7 Take-Away Message

This chapter seemingly provides quite a bit of material and the initial exposure to OOP nomenclature can be, shall we say, challenging. *However*, there are really just three messages you need to

evaluated by invoking `__add__(obj1, obj2)` where `__add__()` is defined in `obj1`'s class statement. There are similar *special methods* associated with the other symbols, such as `__mul__()` for the `*` symbol.

take away from this chapter. First, you should remember that instead of always invoking functions with an expression such as

```
<func>(<params>)
```

we will sometimes work with methods for which we write expressions of the form

```
<object>.<method>(<params>)
```

where `<object>` is a variable, literal data, or, in fact, any expression that evaluates to an object. Second, remember that the `dir()` function can be useful for reminding you of the attributes and methods in a particular class. And, third, remember that the meaning of a symbol can change depending on the class (or type) of the operands.

5.8 Chapter Summary

An *object* is a collection of *attributes* (data) and *methods* (functions).

Objects are said to be instances of a *class*.

A **`class`** statement provides a blueprint for creating objects.

In Python all data are objects. An object's type corresponds to its class.

To access an object's attributes or methods, one writes the object followed by the *access attribute operator*, i.e., a dot (`.`), followed by the desired attribute or method. (Keep in mind that the dot is an operator and cannot be part of an identifier.)

`dir()`: provides a listing of the attributes and methods of an object.

Operators, such as the plus sign, can be *overloaded* so that the operation performed depends on the types of the operands. For example, strings can be *concatenated* if they are "added" together.

Chapter 6

Lists and `for`-Loops

To facilitate solving various problems, we often organize data in some form of *data structure*. There are several different kinds of data structures used in computer science. The basic concepts behind some of these structures are already quite familiar to you. For example, there is a structure known as a *stack* that is similar to a stack of plates or trays, but instead of plates we work with data that is only added or removed from the top of the stack. Another data structure is a *queue* which is similar to a line at a checkout counter, but instead of customers, we work with data that is added to the back of the queue but removed from the front of the queue. In contrast to stacks and queues, other ways of organizing data, such as in a binary *tree*, probably aren't familiar to you. The reason there are different types of data structures is that there is no single "best" data structure: one structure may be ideal for solving one type of problem but poorly suited for solving problems of a different type.

In this chapter we introduce what is perhaps the simplest data structure but also arguably the most important, a *list*.[1] You are already well acquainted with lists and have certainly been using them for most of your life. Perhaps you've written "to do" lists, grocery lists, invitation lists, or wish lists. You've seen lists of capitals, countries, colleges, and courses. Lists may be organized in a particular order, such as a top-ten list, or in no particular order, such as the list of nominees for the Academy Awards. Clearly, lists are a part of our daily lives.

When we create a list, it serves to collect the associated data into one convenient "place." There is the list as a whole and then there are the individual items in the list which we typically refer to as the *elements* of the list. In Python, a list is an object with its own class, the `list` class. Thus, we will typically use `Courier` font when we refer to a `list` in the Python sense of the word. Since a `list` is an object, there are various methods that come with it. We will explore a few of these in this chapter.

In the implementation of algorithms, it is quite common that certain statements need to be repeated multiple times. Thus, computer languages invariably provide ways to construct *loops*. In this chapter we will also introduce one such construct: a `for`-loop. In Python a `for`-loop is a form of *definite loop*, meaning that the number of passes through the loop can be determined in advance (or, said another way, the number of times the loop will execute is known *a priori*). So, for example, we might write a `for`-loop to do something with each element in a `list` of five items. We thus know the loop will execute five times. On the other hand, as we will see in Sec. 11.6, there

From the file: `lists-n-loops.tex`

[1] In some languages, what we call a list is called an *array*.

is a different kind of construct known as an *indefinite loop*. The number of times an indefinite loop will execute is typically *not* known in advance. Indefinite loops can be used, for example, to allow a user to enter values until he or she indicates there are no more values to enter.

In many situations, lists and for-loops go together. A for-loop provides a convenient way to process the data contained in a list; hence lists and for-loops are presented together in this chapter. We start by introducing lists and follow with a discussion of for-loops.

6.1 lists

In Python a list is created when comma-separated expressions are placed between square brackets. For example, [1, "two", 6 / 2] is a list. This list has three elements: the first is an integer (1); the second is a string ("two"); and the third is a float since 6 / 2 evaluates to 3.0. Note that Python will evaluate each expression to determine the value of each element of the list. A list can be either homogeneous, containing only one type of data, or inhomogeneous, containing different types of data. The example above is inhomogeneous since it contains an integer, a string, and a float. (Since a list is a form of data, a list can, in fact, contain another list as one of its elements!)

lists can be assigned to a variable or returned by a function. This is demonstrated in Listing 6.1 where a list is assigned to x in line 1. The print() statement in line 2 shows the result of this assignment. The function f(), defined in lines 4 and 5, returns a list that contains the first four multiples of the parameter passed to the function (actually, as described in more detail below, this is something of an understatement of what f() can do).

Listing 6.1 lists can be assigned to a variable and returned by a function.

```
1  >>> x = [1, "two", 6 / 2]
2  >>> print(x)
3  [1, 'two', 3.0]
4  >>> def f(n):
5  ...     return [n, 2 * n, 3 * n, 4 * n]
6  ...
7  >>> f(2)
8  [2, 4, 6, 8]
9  >>> y = f(49)
10 >>> type(y)
11 <class 'list'>
12 >>> print(y)
13 [49, 98, 147, 196]
14 >>> f('yo')
15 ['yo', 'yoyo', 'yoyoyo', 'yoyoyoyo']
```

In line 8 we see the list produced by the function call f(2) in line 7. In line 9 the list returned by f(49) is assigned to the variable y. Lines 10 and 11 show that y has a type of list. Then, in line 12, a print() statement is used to display y.

In line 14 f() is called with a string argument. Although when f() was written we may have had in mind the generation of multiples of a number, we see that f() can also be used to produce different numbers of repetition of a string because of operator overloading.

Listing 6.2 provides another example of the creation of a list. In lines 2 and 3 the floats a and b are created. In line 4 the list x is created for which each element is an expression involving a and/or b. Line 5 is used to display the contents of x. The discussion continues below the listing.

Listing 6.2 Another demonstration of the creation of a list. Here we see that after a list has been created, subsequent changes to the variables used in the expressions for the elements do not affect the values of the list.

```
1  >>> # Create a list x that depends on the current values of a and b.
2  >>> a = -233.1789
3  >>> b = 4.268e-5
4  >>> x = [a, b, a + b, a - b]
5  >>> x
6  [-233.1789, 4.268e-05, -233.17885732, -233.17894268]
7  >>> # Modify a and b and then confirm that this does not affect x.
8  >>> a = 1
9  >>> b = 2
10 >>> x
11 [-233.1789, 4.268e-05, -233.17885732, -233.17894268]
```

In lines 8 and 9 the values of a and b are changed. The output in line 11 shows these changes to a and b do not affect the elements of x. Once an expression for the element of a list has been evaluated, Python will not go back and recalculate this expression (i.e., after the list has been created, the list is oblivious to subsequent changes to any of the variables that were used in calculating its elements).

6.2 list Methods

Because list is a class, the objects in this class (or, said another way, the instances of this class) will have various methods and attributes which can be listed using the dir() function. Additionally, operator overloading can be used with lists. As with strings, we can use the plus sign to concatenate lists and the multiplication sign for repetition. The length of a list (or tuple) can be determined using the built-in function len(). Listing 6.3 illustrates these features and also demonstrates the creation of a list with no element, i.e., a so-called *empty list* (which we will have occasion to use later).

Listing 6.3 Demonstration of list concatenation, the len() function, and some of the methods available for lists.

```
1  >>> # Concatenate two lists.
2  >>> w = ['a', 'bee', 'sea'] + ["solo", "duo", "trio"]
```

```
3  >>> w
4  ['a', 'bee', 'sea', 'solo', 'duo', 'trio']
5  >>> len(w)               # Determine length.
6  6
7  >>> dir(w)               # Show methods and attributes.
8  ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
9   '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
10  '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
11  '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
12  '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
13  '__reversed__', '__rmul__', '__setattr__', '__setitem__',
14  '__sizeof__', '__str__', '__subclasshook__', 'append', 'count',
15  'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
16 >>> type(w.sort)
17 <class 'builtin_function_or_method'>
18 >>> w.sort()             # Sort list.
19 >>> print(w)
20 ['a', 'bee', 'duo', 'sea', 'solo', 'trio']
21 >>> w.append('quartet') # Append value to list.
22 >>> w
23 ['a', 'bee', 'duo', 'sea', 'solo', 'trio', 'quartet']
24 >>> w.sort()             # Sort list again.
25 >>> w
26 ['a', 'bee', 'duo', 'quartet', 'sea', 'solo', 'trio']
27 >>> z = []               # Create an empty list.
28 >>> print(z)
29 []
30 >>> len(z)
31 0
32 >>> z.append("first")    # Append value to empty list.
33 >>> z
34 ['first']
35 >>> # Extend z with the values from another list.
36 >>> z.extend(["second", "third", "fourth"])
37 >>> z
38 ['first', 'second', 'third', 'fourth']
```

In line 2, two `lists` are concatenated and assigned to `w`. The output on line 4 shows that `w` contains all the elements from the two `lists` that were concatenated. The `len()` function, called on line 5, confirms that `w` is a six-element `list`.

The `dir()` function, called on line 7, shows the attributes and methods for this `list`. Three of the methods (`append()`, `extend()`, and `sort()`) have been highlighted since they are used later in this listing.

In line 16 the `type()` function is used to ascertain the type of `w.sort`. Line 17 reports this is a built-in method. This method is called in line 18 but no output is produced—`sort()` is a void method that returns `None`. However, as you might guess, this method sorts the contents of the `list` on which it is invoked. This is confirmed using the `print()` statement in line 19. In

line 20 the elements are in alphabetical order.

A single object can be appended to the end of a list using the append() method. This is demonstrated in lines 21 through 23. Note that although w was sorted alphabetically once, the sorting is not automatically maintained—values appended to the end will simply stay at the end. If a list needs to be resorted, this is easily accomplished with another call to the sort() method as shown in lines 24 through 26.

In line 27 of Listing 6.3 an empty list is created and assigned to z. When z is printed, we see it has no elements—merely the square brackets are shown. The len() function reports the length of z is 0. In line 32 the append() method is used to append the string first to the list. Since there were no other elements in the list, this string becomes the first and only element. We will often start with an empty list and then add elements to it (perhaps the elements are obtained from the user as he or she enters values).

In addition to using the plus sign to concatenate lists, the extend() method can be used to append the elements of one list to another. This is demonstrated in lines 36 through 38 of Listing 6.3.

The way in which append() and extend() differ is further demonstrated in Listing 6.4. In line 1, the list w is created and consists of three strings. In line 2 a list of two strings is appended to w. The list is treated as a single object and is made the fourth element of w. This is made evident in lines 3 and 4. In line 5 the list u is created that again consists of three strings. In line 6 u is extend()'ed by a list containing two strings. Lines 7 and 8 show that u now contains these two additional strings.

Listing 6.4 Demonstration of the way in which append() differs from extend(). append() appends a single object to a list whereas extend() appends all the objects from the given iterable to the list.

```
1 >>> w = ['a', 'b', 'c']
2 >>> w.append(['d', 'e'])
3 >>> w
4 ['a', 'b', 'c', ['d', 'e']]
5 >>> u = ['a', 'b', 'c']
6 >>> u.extend(['d', 'e'])
7 >>> u
8 ['a', 'b', 'c', 'd', 'e']
```

6.3 for-Loops

The previous section showed some examples of creating a list. When we displayed the lists, we simply displayed the entire list. However, we are often interested in working with the individual elements of a list. A for-loop provides a convenient way to do this. for-loops can be used with more than just lists. In Python lists are considered a type of *iterable*. An iterable is a data type that can return its elements separately, i.e., one at a time. for-loops are, in general, useful when working with any iterable, but here we are mainly concerned with using them with lists.

The template for a for-loop is shown in Listing 6.5. The header, in line 1, contains the keyword for followed by an appropriate lvalue (or identifier) which we identify as *<item>*. For now, you should simply think of *<item>* as a variable which is assigned, with each pass through the loop, the value of each element of the list. We will use the term *loop variable* as a synonym for the lvalue *<item>* that appears in the header. The next component of the header is the keyword in. This is followed by *<iterable>* which, for now, is taken to mean a list or any expression that returns a list. As usual, the header is terminated by a colon. The header is followed by indented code that constitutes the body of the loop.[2] The body is executed once for each element of the list (or iterable).

Listing 6.5 Template for a for-loop.

```
for <item> in <iterable>:
    <body>
```

Another way to interpret a for-loop statement is along the lines of: for each element of the list, do whatever the body of the loop says to do. You are familiar with this type of instruction since you've heard requests such as, "For each day of the week, tell me what you do." In this case the loop variable is the day. It will take on the values Monday, Tuesday, etc., which come from the list of days contained in "week." For each day you report what you do on that particular day. It may help to think of for as meaning "for each."

As something of an aside, we previously said that the body of a function defines a namespace or scope that is unique from the surrounding scope: one can use identifiers defined in the surrounding scope, but any variables defined within the function are strictly local to that function. The body of a function consisted of indented code. The bodies of for-loops also consist of indented code. *However*, this is not associated with a separate scope. Any variables defined within the body of a for-loop persist in the scope in which the for-loop was written.

Let's consider some for-loop examples to illustrate their use. In line 1 of Listing 6.6 a list of names is created and assigned to the variable names. Lines 2 and 3 construct a for-loop that iterates through each element of the names list. It is a common idiom to have the loop variable be a singular noun, such as name, and the list variable be a plural noun, such as names. However, this is not required and the identifiers used for the list and loop variable can be any valid identifier. This is demonstrated in the next loop where, in the header in line 11, the loop variable is foo.

Listing 6.6 Demonstrations showing how a for-loop can be used to access each of the elements of a list.

```
>>> names = ["Uma", "Utta", "Ursula", "Eunice", "Unix"]
>>> for name in names:
...     print("Hi " + name + "!")
```

[2]When the body of the loop consists of a single line, it may be written on the same line as the header, e.g.,
`for s in ["Go", "Fight", "Win"]: print(s)`
However, we will use the template shown in Listing 6.5 even when the body is a single line.

```
4  ...
5  Hi Uma!
6  Hi Utta!
7  Hi Ursula!
8  Hi Eunice!
9  Hi Unix!
10 >>> count = 0              # Initialize a counter.
11 >>> for foo in names:
12 ...     count = count + 1 # Increment counter.
13 ...     print(count, foo) # Display counter and loop variable.
14 ...
15 1 Uma
16 2 Utta
17 3 Ursula
18 4 Eunice
19 5 Unix
```

The body of the first for-loop consists of a single print() statement as shown in line 3. The argument of this print() statement uses string concatenation to connect "Hi " and an exclamation point with the name that comes from names. We see the output this produces in lines 5 through 9.

In line 10 a counter is initialized to zero. In the following for-loop, this counter is incremented as the first statement in the body, line 12. In line 13 print() is used to display the counter and the loop variable. The loop variable here is foo (again, any valid identifier could have been used). However, in the spirit of readability, it would have been better to have used a more descriptive identifier such as name.

Although the code in Listing 6.6 does not illustrate this, after completion of the loop, the loop variable is still defined and has the value of the last element of the list. So, in this example both name and foo end up with the string value Unix.

6.4 Indexing

As shown in the previous section, for-loops provide a convenient way to sequentially access all the elements of a list. However, we often want to access an individual element directly. This is accomplished by enclosing the *index* of the element in square brackets immediately following the list itself (or a list identifier). The index must be an integer (or an expression that returns an integer). You are undoubtedly used to associating an index of one with the first element of a list. However, this is *not* what is done in many computer languages. Instead, the first element of a list has an index of zero. Although this may seem strange at first, there are compelling reasons for this. In Python (and in C, C++, Java, etc.), you should think of the index as representing the offset from the start of the list. Thus, an index of zero represents the first element of a list, one is the index of the second element, and so on. Pausing for a moment, you may realize that we already have a general way to obtain the index of the last element in the list. This is demonstrated in Listing 6.7.

Listing 6.7 Demonstration of the use of indexing to access individual elements of a list.

```
>>> xlist = ["Do", "Re", "Mi", "Fa", "So", "La", "Ti"]
>>> xlist[0]              # First element.
'Do'
>>> xlist[1]              # Second element.
'Re'
>>> xlist[1 + 1]          # Third element.
'Mi'
>>> len(xlist)            # Length of list.
7
>>> xlist[len(xlist) - 1] # Last element.
'Ti'
>>> xlist[len(xlist)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> ["Fee", "Fi", "Fo", "Fum"][1]
'Fi'
>>> ["Fee", "Fi", "Fo", "Fum"][3]
'Fum'
>>> ([1, 2, 3] + ['a', 'b', 'c'])[4]
'b'
```

In line 1 of Listing 6.7 a list of seven strings is created and assigned to the variable xlist. Lines 2, 4, and 6 access the first, second, and third elements, respectively. Note that in line 6 an expression is used to obtain the index. If the expression evaluates to an integer, this is allowed.

In line 8 the length of the list is obtained using the len() function. There are seven elements in xlist, but the last valid index is one less than this length. In line 10 the index of the last element is obtained by subtracting one from the length. Calculating the index this way for the last element is valid for a list of any size.[3][4]

The statement in line 12 shows the error that is produced when the index is outside the range of valid indices, i.e., one obtains an IndexError. Keep in mind that the largest valid index is one less than the length of the list.

Lines 16 through 19 demonstrate that one can provide an index to directly obtain an index from a list literal. Typically this wouldn't be done in practice (why bother to enter the entire list into your code if only one element was of interest?). But, this serves to illustrate that one can index any expression that returns a list. To truly demonstrate this fact, in line 20 two lists are concatenated. This concatenation operation is enclosed in parentheses and produces the new list [1, 2, 3, 'a', 'b', 'c']. To the right of the parentheses is the integer 4 enclosed in

[3] However, an empty list has no valid indices since it has no elements. Thus this approach does not work for calculating the index of the last element of an empty list. However, since an empty list has no elements, this point is somewhat moot.

[4] As we will see in Sec. 7.5, negative indexing provides a more convenient way to access elements at the end of a list.

square brackets. This accesses the fifth element of the list, i.e., this index selects the string `'b'` as indicated by the result shown on line 21.

Let us return to the `for`-loop but now use the loop variable to store an integer instead of an element from a `list`. Listing 6.8 demonstrates that this approach can be used to access the elements of the list `fruits` either in order or in reverse order. The code is further discussed below the listing.

Listing 6.8 Demonstration that the elements of a `list` can be accessed using direct indexing and a `for`-loop. The loop variable is set to an integer in the range of valid indices for the `list` of interest.

```
1  >>> fruits = ["apple", "banana", "grape", "kiwi", "pear"]
2  >>> indices = [0, 1, 2, 3, 4]
3  >>> for i in indices:
4  ...     print(i, fruits[i])
5  ...
6  0 apple
7  1 banana
8  2 grape
9  3 kiwi
10 4 pear
11 >>> for i in indices:
12 ...     index = len(fruits) - 1 - i
13 ...     print(i, index, fruits[index])
14 ...
15 0 4 pear
16 1 3 kiwi
17 2 2 grape
18 3 1 banana
19 4 0 apple
```

In line 1 the list `fruits` is created with five elements. In line 2 the list `indices` is created with elements corresponding to each of the valid indices of `fruits` (i.e., the integers `0` through `4`).

The header of the `for`-loop in line 3 sets the loop variable `i` equal to the elements of `indices`, i.e., `i` will be assigned the values `0` through `4` for passes through the loop. The loop variable `i` is then used in the `print()` statement in line 4 to show the elements of `fruits`. The output in lines 6 through 10 shows both the index `i` and the corresponding element of `fruits`.

The `for`-loop in line 11 again sets `i` to the valid indices of the `fruits` list. However, rather than directly using `i` to access an element of `fruits`, we instead calculate an index that is given by `len(fruits) - 1 - i`. When `i` is zero, this expression yields the index of the last element in `fruits`. When `i` is `1`, this expression yields `3` which is the second to the last element, and so on. Each line of output in lines 15 through 19 shows, in order, the value of `i`, the calculated index, and the element of `fruits` for this index.

6.5 range()

In line 2 of Listing 6.8 a list called indices was created that contains the valid indices for the list fruits. We are indeed often interested in accessing each element of a list via an index. However, it would be quite inconvenient if we always had to create an indices list, as was done in Listing 6.8, that explicitly listed all the indices of interest. Fortunately there is a much better way to obtained the desired indices.

Python provides a function, called range(), that generates a sequence of integers. We can use this function to generate the integers corresponding to all the valid indices for a given list. Or, as you will see, we can use it to generate some subset of indices. The range() function does not actually produce a list of integers.[5] However, we can force range() to show the entire sequence of values that it ultimately produces if we enclose range() in the function list().[6]

The range() function is used as indicated in Listing 6.9.

Listing 6.9 The range() function and its parameters. Parameters in brackets are optional. See the text for further details.

```
range([start,] stop [, increment])
```

In its most general form, the range() function takes three parameters that we identify as start, stop, and increment. The start and increment parameters are optional and hence are shown in square brackets.[7] When they are not explicitly provided, start defaults to zero and increment defaults to positive one. There is no default value for stop as this value must always be provided explicitly. When two parameters are given, they are taken to be start and stop.

The first integer produced by range() is start. The next integer is given by start + increment, the next is start + 2 * increment, and so on. increment may be positive or negative, so the values may be increasing or decreasing. Integers continue to be produced until reaching the last value "before" stop. If increment is positive, the sequence of integers ends at the greatest value that is still strictly less than stop. On the other hand, if increment is negative, then the sequence ends with the smallest value that is still strictly greater than stop.

Admittedly, it can be difficult to understand what a function does simply by reading a description of it (such as given in the previous paragraph). However, a few examples usually help clarify the description. Listing 6.10 provides several examples illustrating the behavior of the range() function. The list() function is used so that we can see, all at once, the values produced by range().

[5] range() returns an *iterable* that can be used in a for-loop header. With each pass of the loop, the range() function provides the next integer in the sequence.

[6] The built-in function list() attempts to convert its argument to a list. If the conversion is not possible, an exception is raised (i.e., an error is generated). Thus, as with int(), float(), and the str() function, the list() function performs a form of data conversion. The reason we avoid using "list" as a variable name is so that we don't mask this function in a manner similar to the way print() was masked in Listing 2.16.

[7] This way of indicating optional arguments is fairly common and the square brackets here have nothing to do with lists.

Listing 6.10 Examples of the behavior of the `range()` function. The `list()` function is merely used to produce all of `range()`'s output in a single `list`.

```
1  >>> # The first three commands are all identical in that the arguments
2  >>> # provided for the start and increment are the same as the default
3  >>> # values of 0 and 1, respectively.
4  >>> list(range(0, 5, 1))   # Provide all three parameters.
5  [0, 1, 2, 3, 4]
6  >>> list(range(0, 5))      # Provide start and stop.
7  [0, 1, 2, 3, 4]
8  >>> list(range(5))         # Provide only stop.
9  [0, 1, 2, 3, 4]
10 >>> list(range(1, 10, 2)) # Odd numbers starting at 1.
11 [1, 3, 5, 7, 9]
12 >>> list(range(2, 10, 2)) # Even numbers starting at 2.
13 [2, 4, 6, 8]
14 >>> list(range(5, 5))      # No output since start equals stop.
15 []
16 >>> list(range(5, 0, -1)) # Count down from 5.
17 [5, 4, 3, 2, 1]
```

The statements in lines 4, 6, and 8 use three arguments, two arguments, and one argument, respectively. However, these all produce identical results since the `start` and `increment` values are set to the default values of `0` and `1`, respectively. The statement in line 10 produces a `list` that starts at `1` and increases by `2`, i.e., it generates odd numbers but stops at `9` (since this is the largest value in the sequence that is less than the `stop` value of `10`). The statement in line 12 uses the same `stop` and `increment` as in line 10 but now the `start` value is `2`. Thus, this statement produces even numbers but stops at `8`.

As lines 14 and 15 show, `range()` does not produce any values if `start` and `stop` are equal. Finally, the statement in line 16 has a `start` value greater than the `stop` value. No integers would be produced if `increment` were positive; however, in this case the `increment` is negative. Hence the resulting sequence shown in line 17 is descending. As always, the result in line 17 does not include the `stop` value.

Assume an arbitrary `list` is named `xlist`: What integers are produced by `range(len(xlist))`? Do pause for a moment to think about this. We know that `len(xlist)` returns the length of its argument. This value, in turn, serves as the sole argument to the `range()` function. As such, `range()` will start by producing `0` and, incrementing by one, go up to one less than the length of the `list`. These are precisely the valid indices for `xlist`! Since we made no assumptions about the number of elements in `xlist`, we can use this construct to obtain the valid indices for any `list`.

Listing 6.11 demonstrates the use of the `range()` function in the context of `for`-loops. The `for`-loop in lines 1 and 2 uses the `range()` function to generate the integers `0` through `4`. The subsequent loop, in lines 9 and 10, shows how the three-parameter form of the `range()` function can generate these values in reverse order.

Listing 6.11 Demonstration of the use of the `range()` function in the context of `for`-loops. The last four loops show how the elements of a `list` can be conveniently accessed with the aid of the `range()` function.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(4, -1, -1):
...     print(i)
...
4
3
2
1
0
>>> # Create a list of toppings and display in order.
>>> toppings = ["cheese", "pepperoni", "pineapple", "anchovies"]
>>> for i in range(len(toppings)):
...     print(i, toppings[i])
...
0 cheese
1 pepperoni
2 pineapple
3 anchovies
>>> # Have index go in descending order to show list in reverse.
>>> for i in range(len(toppings) - 1, -1, -1):
...     print(i, toppings[i])
...
3 anchovies
2 pineapple
1 pepperoni
0 cheese
>>> # Have loop variable take on values in ascending order, but
>>> # use this to calculate index which yields list in reverse order.
>>> for i in range(len(toppings)):
...     print(i, toppings[len(toppings) - 1 - i])
...
0 anchovies
1 pineapple
2 pepperoni
3 cheese
>>> # Obtain first and third topping (indices 0 and 2).
>>> for i in range(0, len(toppings), 2):
```

```
45 ...     print(i, toppings[i])
46 ...
47 0 cheese
48 2 pineapple
```

After creating the toppings list in line 18, the for-loop in lines 19 and 20 is used to show the elements of toppings in order. Note how the range() function is used in the header.

The for-loop in lines 27 and 28 used the three-parameter form of the range() function to cycle the loop variable i from the last index to the first. The for-loop in lines 36 and 37 also displays toppings in reverse order, but here the loop variable is ascending. Thus, in line 37, the loop variable is subtracted from len(toppings) - 1 in order to obtain the desired index.

The range() function in the header of the for-loop in line 44 has an increment of 2. Thus, only the first and third values are displayed in the subsequent output.

6.6 Mutability, Immutability, and Tuples

A list serves as a way to collect and organize data. As shown above, we can append or extend a list. But, we can also change the values of individual elements of a list. We say that a list is mutable which merely means we can change it. The mutability of lists is demonstrated in Listing 6.12.

Listing 6.12 Demonstration of the mutability of a list.

```
1 >>> x = [1, 2, 3, 4]              # Create list of integers.
2 >>> x[1] = 10 + 2                 # Change second element.
3 >>> x                             # See what x is now.
4 [1, 12, 3, 4]
5 >>> x[len(x) - 1] = "the end!"    # Change last element to a string.
6 >>> x
7 [1, 12, 3, 'the end!']
```

A list x is created in line 1. In line 2 the second element of the list is assigned a new value (that is obtained from the expression on the right). Lines 3 and 4 display the change. In line 5 the last element of x is set equal to a string. Note that it does not matter what the previous type of an element was—we are free to set an element to anything we wish.

It is worthwhile to spend a bit more time considering line 2. Line 2 employs the assignment operator. Previously we said that, in statements such as this, the expression to the right of the equal sign is evaluated and then the resulting value is assigned to the *lvalue* to the left of the equal sign. You may have wondered why we didn't just say "assigned to the *variable* to the left of the equal sign" or "to the *identifier* to the left of the equal sign." Line 2 shows us the reason for using lvalue instead of variable. If, in line 2, we had written x = 10 + 2, then we would indeed have assigned the value on the right to the variable on the left (and, in this case, x would now point to the int 12 rather than to the list it was originally assigned). But, instead, in line 2 we have x[1] = 10 + 2. In this case the variable x is associated with the entire list while x[1] is a single

element of this list. We can assign a value to this element (or use the element in expressions). Since we can assign a value to it, it can appear to the left of the assignment operator and is thus considered an lvalue. An element from a list is not typically considered a variable.

In Sec. 4.3, in connection with returning multiple values from a function, it was mentioned that the data was returned in a *tuple*. A `tuple` is another data type. Its behavior is quite similar to that of a `list`. To access the elements of a `tuple`, one still uses an index enclosed in square brackets. The first element has an index of zero. The length of a `tuple` is given by the `len()` function. One difference between a `tuple` and a `list` is that a `tuple` is created by enclosing the comma-separated data in parentheses (instead of square brackets as is done for a `list`). So, for example, `(1, "two", 2 + 1)` produces a `tuple` with elements of `1`, `"two"`, and `3`. However, if the comma-separated values do not span more than one line, the parentheses are optional. Listing 6.13 shows some examples pertaining to creating and working with `tuple`s.

Listing 6.13 Demonstration of the use of `tuple`s.

```
1  >>> t = 1, "two", 2 + 1
2  >>> t
3  (1, 'two', 3)
4  >>> type(t)
5  <class 'tuple'>
6  >>> for i in range(len(t)):
7  ...     print("t[", i, "] = ", t[i], sep="")
8  ...
9  t[0] = 1
10 t[1] = two
11 t[2] = 3
12 >>> z = ("one",
13 ...      "two",
14 ...      3)
15 >>> print(z)
16 ('one', 'two', 3)
```

In line 1 a `tuple` is created and assigned to the variable `t`. Note that no parentheses were used (even though enclosing the values to the right of the assignment operator in parentheses arguably would make the code easier to read). Line 2 is used to echo the `tuple`. We see the values are now enclosed in parentheses. Python will use parentheses to represent a `tuple` whether or not they were present when the `tuple` was created. Line 4 is used to show that `t`'s type is indeed `tuple`. The `for`-loop in lines 6 and 7 shows that a `tuple` can be indexed in the same way that we indexed a `list`. The statement in lines 12 through 14 creates a `tuple` called `z`. Here, since the statement spans multiple lines, parentheses are necessary.

The one major difference between `list`s and `tuple`s is that `tuple`s are *immutable*, meaning their values *cannot* be changed. This might sound like it could cause problems in certain situations, but, in fact, there is an easy fix if we ever need to change the value of an element in a `tuple`: we can simply convert the `tuple` to a `list` using the `list()` function. The immutability of a `tuple` and the conversion of a `tuple` to a `list` are illustrated in Listing 6.14.

Listing 6.14 Demonstration of the immutability of a `tuple` and how a `tuple` can be converted to a `list`.

```
1  >>> t = 'won', 'to', 1 + 1 + 1  # Create three-element tuple.
2  >>> t
3  ('won', 'to', 3)
4  >>> t[1] = 2           # Cannot change a tuple.
5  Traceback (most recent call last):
6    File "<stdin>", line 1, in <module>
7  TypeError: 'tuple' object does not support item assignment
8  >>> t = list(t)        # Convert tuple to a list.
9  >>> type(t)
10 <class 'list'>
11 >>> t[1] = 2           # Can change a list.
12 >>> t
13 ['won', 2, 3]
```

The tuple `t` is created in line 1. In line 4 an attempt is made to change the value of the second element in the `tuple`. Since `tuple`s are immutable, this produces the `TypeError` shown in lines 5 through 7.

In line 8 the `list()` function is used to convert the tuple `t` to a `list`. This `list` is reassigned back to the variable `t`. Lines 9 and 10 show that the type of `t` has changed and the remaining lines show that we can now change the elements of this `list`. (When we used `t` as the lvalue in line 8, we lost the original tuple. We could have used a different lvalue, say `tList`, in which case the tuple `t` would still have been available to us.)

The detailed reasons for the existence of both `tuple`s and `list`s don't concern us so we won't bother getting into them. We will just say that this relates to the way data is managed in memory and how values can be protected from being overwritten. There are times when a programmer does not want the values in a collection of data to be changed. A `tuple` provides a means of protection, although, as we've seen, with some effort, the tuple can be converted (and copied) to another form and then changed.[8]

6.7 Nesting Loops in Functions

It is possible to have a `for`-loop contained within the body of a function: The `for`-loop is said to be *nested* inside the function. Since a `for`-loop has a body of its own, its body must be indented farther than the statements of the body in which it is nested. Listing 6.15 provides a template for a `for`-loop nested inside a function. As indicated in lines 2 and 5, there can be code both before and after the loop, though neither is required.

[8] But, in fact, the original `tuple` wasn't changed—if another variable references this data, the `tuple` will persist in memory in its original form. The `list` that is created is a *copy* of the `tuple` that will occupy different memory than the `tuple`. As mentioned, if this new `list` is assigned to the same identifier as was used for the `tuple`, then this identifier references this new `list`/new memory. However, the assignment, in itself, does not destroy the original `tuple`.

Listing 6.15 Template for nesting a loop inside a function.

```
def <function_name>(<parameter_list>):
    <function_body_before_loop>
    for <item> in <iterable>:
        <for_loop_body>
    <function_body_after_loop>
```

To help illustrate the use of a `for`-loop inside a function assume a programmer wants to write a function that will prompt the user for an integer N and a `float`. The function should display the first N multiples of the `float` and then return the last multiple. We'll start by considering a couple of ways things can go wrong before showing a correct implementation.

Listing 6.16 shows a broken implementation in which the programmer places the prompt for input inside the body of the `for`-loop. In this case the user is prompted for the `float` in each pass of the loop.

Listing 6.16 Flawed implementation of a function where the goal is to show a specified number of multiples of a number that the user enters.

```
>>> def multiples():
...     num_mult = int(input("Enter number of multiples: "))
...     for i in range(1, num_mult + 1):
...         x = float(input("Enter a number: "))
...         print(i, i * x)
...     return i * x
...
>>> multiples()
Enter number of multiples: 4
Enter a number: 7
1 7.0
Enter a number: 7
2 14.0
Enter a number: 7
3 21.0
Enter a number: 7
4 28.0
28.0
```

The problem with this code is that the `input()` statement is in the body of the `for`-loop (see line 4). The user should be prompted for this value *before* entering the loop. As things stand now, when the user requests N multiples, the user also has to enter the `float` value N separate times.

Listing 6.17 shows another broken implementation. This time the user is properly prompted for the number of multiples and the `float` value prior to the loop. However, the `return` statement in line 6 is indented to the same level as the body of the `for`-loop. Because of this, the `return`

statement will terminate the function in the first pass of the loop; once a `return` statement is encountered, a function is terminated. This is evident from the output of the function which is shown in line 11, i.e., there is only one line of output despite the fact that the user requested `4` multiples as shown in line 9. When `multiples()` is invoked again, in line 13, the user enters that `4000` multiples are desired. However, again, only one line of output is produced as shown in line 16. (Note that the values displayed in lines 12 and 17 are the return values of the function that are echoed by the interactive environment—these values are not printed by the function itself.) So, keep in mind that the amount of indentation *is* important!

Listing 6.17 Another flawed implementation of a function where the goal is to show a specified number of multiples of a given value.

```
1  >>> def multiples():
2  ...     num_mult = int(input("Enter number of multiples: "))
3  ...     x = float(input("Enter a number: "))
4  ...     for i in range(1, num_mult + 1):
5  ...         print(i, i * x)
6  ...         return i * x
7  ...
8  >>> multiples()
9  Enter number of multiples: 4
10 Enter a number: 7
11 1 7.0
12 7.0
13 >>> multiples()
14 Enter number of multiples: 4000
15 Enter a number: 7
16 1 7.0
17 7.0
```

Finally, Listing 6.18 shows a proper implementation of the `multiples()` function.

Listing 6.18 Function with statements before and after the nested `for`-loop. This implementation properly obtains input from the user prior to the `for`-loop and returns the desired value after the `for`-loop has ended.

```
1  >>> def multiples():
2  ...     num_mult = int(input("Enter number of multiples: "))
3  ...     x = float(input("Enter a number: "))
4  ...     for i in range(1, num_mult + 1):
5  ...         print(i, i * x)
6  ...     return i * x  # Code after and outside the loop.
7  ...
8  >>> multiples()
9  Enter number of multiples: 4
10 Enter a number: 7
```

```
11 1 7.0
12 2 14.0
13 3 21.0
14 4 28.0
15 28.0
```

Later we will learn about other constructs, such as while-loops and conditional statements, that also have bodies of their own. All these constructs can be nested inside other constructs. In fact, we can nest one for-loop within another as will be discussed in Sec. 7.1. The important syntactic consideration is that the body of each of these is indented relative to its header.

6.8 Simultaneous Assignment with Lists

Simultaneous assignment is discussed in Sec. 2.4. In simultaneous assignment there are two or more comma-separated expressions to the right of the equal sign and an equal number of comma-separated lvalues to the left of the equal sign. As discussed in Sec. 6.6, a comma-separated collection of expressions automatically forms a tuple (whether or not it is surrounded by parentheses). Thus, another way to think of the simultaneous assignment operations we have seen so far is as follows: The number of lvalues to the left of the equal sign must be equal to the number of elements in the tuple to the right of the equal sign.

Given the claim in Sec. 6.6 that lists and tuples are nearly identical (with the exception that lists are mutable and tuples are not), one might guess that lists can be used in simultaneous assignment statements too. This is indeed the case. Listing 6.19 illustrate this.

Listing 6.19 Demonstration that lists, like tuples, can be used in simultaneous assignment statements. The assignments in lines 1 and 4 involve tuples while the ones in lines 7 and 11 involve lists. All behave the same way in terms of the actual assignment to the lvalues on the left side of the equal sign.

```
1  >>> x, y, z = 1, 2, 3   # Tuple to right, without parentheses.
2  >>> print(x, y, z)
3  1 2 3
4  >>> r, s, t = (1, 2, 3) # Tuple to right, with parentheses.
5  >>> print(r, s, t)
6  1 2 3
7  >>> a, b, c = [1, 2, 3] # List to right.
8  >>> print(a, b, c)
9  1 2 3
10 >>> xlist = [1, 2, 3]
11 >>> i, j, k = xlist
12 >>> print(i, j, k)
13 1 2 3
```

In line 1 simultaneous assignment is used in which a tuple appears to the right of the equal sign. This tuple is given without parentheses. Lines 2 and 3 show the result of this assignment.

In line 4 simultaneous assignment is again used with a tuple appearing to the right of the equal sign. However, parentheses enclose the elements of the tuple. These parentheses have no effect, and the statement in line 4 is functionally identical to the statement in line 1.

In lines 7 and 11 simultaneous assignment is used where now a *list* appears to the right of the equal sign. The output in lines 9 and 13 shows these assignments behave in the same way as the assignments in lines 1 and 4. Thus, for simultaneous assignment it does *not* matter if one is dealing with a list or a tuple.

We previously saw, in Listing 2.9, that simultaneous assignment can be used to swap the values of two variables. This sort of swapping can be done regardless of the type of data to which the variables point. This is illustrated by the code in Listing 6.20.

Listing 6.20 Demonstration that simultaneous assignment can be used with any data type including entire lists and tuples.

```
1  >>> p = ['a', 'b', 'c', 'd']   # Assign a list of strings to p.
2  >>> n = (12, 24)               # Assign a tuple of integers to n.
3  >>> print(p, n)                # Print p and n.
4  ['a', 'b', 'c', 'd'] (12, 24)
5  >>> p, n = n, p                # Simultaneous assignment to swap p and n.
6  >>> print(p, n)                # See what p and n are now.
7  (12, 24) ['a', 'b', 'c', 'd']
8  >>> w = "WoW"                  # Define string.
9  >>> print(p, n, w)             # Print variables.
10 (12, 24) ['a', 'b', 'c', 'd'] WoW
11 >>> p, n, w = w, p, n          # Swap variables.
12 >>> print(p, n, w)             # Print variables again.
13 WoW (12, 24) ['a', 'b', 'c', 'd']
```

In lines 1 and 2 a list and a tuple are assigned to variables. In line 5 the data assigned to these variables are swapped using simultaneous assignment. In line 8 a string is assigned to a variable. In line 11 the list, tuple, and string data are swapped among the various variables using simultaneous assignment. The output in line 13 shows the result of the swap.

6.9 Examples

In this section we present three examples that demonstrate the utility of lists, for-loops, and the range() function. We also introduce the concept of an accumulator.

6.9.1 Storing Entries in a list

Assume we want to allow the user to enter a list of data. For now we will require that the number of entries be specified in advance and that the entries will be simply stored as strings (if we want to store integers or floats, we can use, as appropriate, int(), float(), or eval() to perform the desired conversion of the input).

Listing 6.21 shows code that is suitable for this purpose. In lines 1 through 6 the function `get_names()` is defined. This function takes a single parameter, `num_names`, which is the number of names to be read. The body of the function starts, in line 2, by initializing the `list` `names` to the empty `list`. This is followed by a `for`-loop where the header is merely used to ensure the body of the loop is executed the proper number of times, i.e., it is executed `num_names` times. In the body of the loop, in line 4, the `input()` function is used to prompt the user for a name. Note how the prompt is constructed by adding one to the loop variable, converting this sum to a string (with the `str()` function), and then concatenating this with other strings. (Lines 9 through 11 show the resulting prompt.) The string entered by the user is stored in the variable `name` and this is appended to the `names` `list` in line 5. Please note the indentation that is used. The `for`-loop is *nested* inside the body of the function and hence the body of the `for`-loop must be indented even farther. Finally, in line 6, the `names` `list` is returned by the function. Importantly, note that the `return` statement is outside the `for`-loop since it is not indented at the same level as the body of the loop. Were the `return` statement in the body of the loop, the loop could not execute more than once—the function would be terminated as soon as the `return` statement was encountered. Nesting of a loop in a function is discussed in Sec. 6.7 while the remainder of the code in Listing 6.21 is discussed following the listing.

Listing 6.21 Function to create a `list` of strings where each string is entered by the user.

```
1  >>> def get_names(num_names):
2  ...     names = []
3  ...     for i in range(num_names):
4  ...         name = input("Enter name " + str(i + 1) + ": ")
5  ...         names.append(name)
6  ...     return names
7  ...
8  >>> furry_friends = get_names(3)
9  Enter name 1: Bambi
10 Enter name 2: Winnie the Pooh
11 Enter name 3: Thumper
12 >>> print(furry_friends)
13 ['Bambi', 'Winnie the Pooh', 'Thumper']
14 >>> n = int(input("Enter number of names: "))
15 Enter number of names: 4
16 >>> princesses = get_names(n)
17 Enter name 1: Cinderella
18 Enter name 2: Snow White
19 Enter name 3: Princess Tiana
20 Enter name 4: Princess Jasmine
21 >>> princesses
22 ['Cinderella', 'Snow White', 'Princess Tiana', 'Princess Jasmine']
```

In line 8 the `get_names()` function is called with an argument of `3`. Thus, the user is prompted to enter three names as shown in lines 9 through 11. The `list` that contains these names is returned

by `get_names()` and, also in line 11, assigned to the variable `furry_friends`. Line 12 is used to show the contents of `furry_friends`.

Instead of "hardwiring" the number of names, in line 14 the user is prompted to enter the desired number of names. The user enters `4` in line 15 and this value is stored in `n`. The `get_names()` function is called in line 16 with an argument of `n`. Hence, the user is prompted for four names as shown in lines 17 through 20. The output in line 22 shows that the user's input is now contained in the `list` `princesses`.

6.9.2 Accumulators

Often we want to perform a calculation that requires the accumulation of values, i.e., the final value is the result of obtaining contributions from multiple parts. Assume we want to find the value of the following series, where N is some integer value:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N}.$$

You may already be familiar with the mathematical representation of this series where one would write:

$$\sum_{k=1}^{N} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N}.$$

The symbol Σ is the Greek letter sigma which we use to stand for "sum." The term below Σ specifies the starting value of the "summation variable." In this case the summation variable is k and it starts with a value of 1. The term above Σ specifies the final value of this variable. In this particular case we haven't yet said explicitly what the final numeric value is. Instead, we write that the final value is N with the understanding that this has to be specified (as an integer) when we actually calculate the series. The expression immediately to the right of Σ is a general expression for the individual terms in the sum—the actual value is obtained by plugging in the value of the summation variable as it varies between the initial and final values.

We can't calculate this series all at once. Instead, we start with the first term in the series. We then add the next term. We store this result and then add the next term, and so on. Thus we *accumulate* the terms until we have added all of them. For example, if N is 4, we start with the first term of 1 ($k = 1$). We add 0.5 ($k = 2$) to obtain 1.5. We add 0.33333$\ldots$ ($k = 3$) to obtain 1.83333$\ldots$; and then we add 0.25 ($k = 4$) to obtain 2.0833333$\ldots$.

Problems that involve the accumulation of data are quite common. Typically we initialize an identifier to serve as an *accumulator*. This initialization involves setting the accumulator to an appropriate value outside a `for`-loop. Then, in the body of the loop, the desired data contribute to the accumulator as necessary. An accumulator is essentially a variable that is used to accumulate information (or data) with each pass of a loop. When an accumulator is modified, its new value is based in some way on its previous value. The example in Listing 6.21 actually also involves an accumulator, albeit of a different type. In that example we accumulated strings into a list. The accumulator is the list `names` which started as an empty list.

We want to write a function that calculates the series shown above (and returns the resulting value). In this case, for which we are summing numeric values, the accumulator should be a `float` that is initialized to `0.0`. This accumulator should be thought of as the running sum of

the terms in the series. Listing 6.22 shows the appropriate code to implement this. The function `calc_series()` takes a single argument which is the number of terms in the series.

Listing 6.22 Function to calculate the series $\sum_{k=1}^{N} = 1/k$. The variable `total` serves as an accumulator.

```
1  >>> def calc_series(num_terms):
2  ...     total = 0.0    # The accumulator.
3  ...     for k in range(num_terms):
4  ...         total = total + 1 / (k + 1) # Add to accumulator.
5  ...     return total   # Return accumulator.
6  ...
7  >>> calc_series(1)     # 1 term in series.
8  1.0
9  >>> calc_series(2)     # 2 terms in series.
10 1.5
11 >>> calc_series(4)     # 4 terms in series.
12 2.083333333333333
13 >>> calc_series(400)   # 400 terms in series.
14 6.5699296911765055
15 >>> calc_series(0)     # No terms in series.
16 0.0
```

The function `calc_series()` is defined in lines 1 through 5. It takes one argument, `num_terms`, which is the number of terms in the series (corresponding to N in the equation for the series given above). In line 2 the accumulator `total` is initialized to zero. The body of the `for`-loop in lines 3 and 4 will execute the number of times specified by `num_terms`. The loop variable `k` takes on the values `0` through `num_terms - 1`. Thus, in line 4, we add `1` to `k` to get the desired denominator. Alternatively, we can implement the loop as follows:

```
    for k in range(1, num_terms + 1):
        total = total + 1 / k
```

Or, using the augmented assignment operator for addition, which was discussed in Sec. 2.8.4, an experienced programmer would likely write:

```
    for k in range(1, num_terms + 1):
        total += 1 / k
```

For our purposes, any of these implementations is acceptable although this final form is probably the one that most clearly represents the original mathematical expression.

6.9.3 Fibonacci Sequence

The Fibonacci sequence starts with the numbers 0 and 1. Then, to generate any other number in the sequence, you add the previous two. In general, let's identify numbers in the sequence as F_n where n is an index that starts from 0. We identify the first two numbers in the sequence as $F_0 = 0$

and $F_1 = 1$. Now, what is the next number in the sequence, F_2? Since a number in the sequences is always the sum of the previous two numbers, F_2 is given by $F_1 + F_0 = 1 + 0 = 1$. Moving on to the next number, we have $F_3 = F_2 + F_1 = 2$. Thinking about this for a moment leads to the following equation for the numbers in the Fibonacci sequence:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

There is a rather elegant way Python can be used to generate the numbers in this sequence. The function `fib()` shown in Listing 6.23 calculates F_n for any value of n. Although elegant, this function is somewhat sophisticated because of the way it employs simultaneous assignment. The function is defined in lines 1 through 5. In line 2 the variables `old` and `new` are defined. These are the first two numbers in the sequence. Assume the user only wants the first number of the sequence, i.e., F_0, so that the argument to `fib()` is `0`. When this is the case, the `for`-loop is not executed and the value of `old` (which is `0`) is returned. Even though this function internally has a "`new`" value corresponding to F_1, it returns the "`old`" value.

In fact, this function always returns the "`old`" value and this corresponds to the desired number in the sequence, i.e., the nth value. The function always calculates the $(n + 1)$th value and stores it in `new`, but ultimately it is not returned.

Listing 6.23 Demonstration of a function to calculate the numbers in the Fibonacci sequence.

```
1  >>> def fib(n):
2  ...     old, new = 0, 1      # Initialize starting values.
3  ...     for i in range(n):
4  ...         old, new = new, old + new
5  ...     return old
6  ...
7  >>> fib(0)
8  0
9  >>> fib(1)
10 1
11 >>> fib(2)
12 1
13 >>> fib(3)
14 2
15 >>> fib(4)
16 3
17 >>> fib(45)
18 1134903170
```

The right side of line 4 has both the current value in the sequence and the expression corresponding to the *next* value in the sequence, i.e., `old + new`. These two values are simultaneously assigned to `old` and `new`. Thus the current value gets assigned to `old` and the "next" value gets assigned to `new`. This is repeated until `old` contains the desired number. The calls to `fib()` in lines 7 through 17 demonstrate that the function works properly.

Of course, one does not need to use simultaneous assignment to implement the Fibonacci sequence (recall that not all computer languages allow simultaneous assignment). Listing 6.24 shows an alternate implementation that does not employ simultaneous assignment. In line 5 a temporary variable has to be used to store the value of `new` so that its value can subsequently be assigned to `old` (i.e., after `old` has been used to update `new` in line 6). Both implementations of the Fibonacci sequence require a bit of thought to be fully understood.

Listing 6.24 An alternative implementation of the Fibonacci sequence that does not use simultaneous assignment.

```
>>> def fib_alt(n):
...     old = 0
...     new = 1
...     for i in range(n):
...         temp = new
...         new = old + new
...         old = temp
...     return old
...
>>> fib_alt(0)
0
>>> fib_alt(1)
1
>>> fib_alt(4)
3
>>> fib_alt(45)
1134903170
```

6.10 Chapter Summary

A `list` is a sequential collection of data. The data can differ in terms of type, i.e., a `list` can be inhomogeneous.

`lists` can be created by enclosing comma-separated expressions in square brackets, e.g., `[2, "t", 1 + 1]`.

An empty `list` has no elements, i.e., `[]` is an empty list.

Two `lists` can be concatenated using the + operator.

Repetition of a `list` can be obtained using the `*` operator (where one of the operands is a `list` and the other is an integer).

The **`append()`** method can be used to append its argument to the end of a `list`. The **`extend()`** method can be used to add the elements of the argument `list` to the `list` for which the method is invoked. The **`sort()`** method sorts the elements of a `list` in place, i.e., a new `list` isn't created but rather the original `list` is changed.

An individual element of a `list` can be ac-

cessed via an integer index. The index is given in square brackets following the `list`. The index represents an offset from the first element; hence the first element has an index of `0`, e.g., `xlist[1]` is the *second* element of `xlist`.

`len()`: returns the length of its argument as an integer. When the argument is a `list`, `len()` returns the number of elements.

In general, for a list `xlist`, the last element has an index of `len(xlist) - 1`.

A **`for`-loop** uses the following template:

```
for <item> in <iterable>:
    <body>
```

where `<item>` corresponds to the *loop variable* and is any valid identifier (or lvalue), `<iterable>` is an object such as a `list` that returns data sequentially, and the body is an arbitrary number of statements that are indented to the same level.

`range()`: function used to produce integers. The general form is `range(start, stop, inc)`. The integers that are produced start at `start`. Each successive term is incremented by `inc`. The final value produced is the "last" one before `stop`. Both `inc` and `start` are optional and have default values of `1` and `0`, respectively. `inc` may be positive or negative.

Given a list `xlist`, `range(len(xlist))` will produce, in order, all the valid indices for this `list`.

The `range()` function can be used as the iterable in the header of a `for`-loop. This can be done either to produce a counted loop where the loop variable is not truly of interest or to produce the valid indices of a `list` (in which case the loop variable is used to access the elements of the `list`).

`list()`: returns the `list` version of its argument. (This function can be used to obtain a `list` containing all the values generated by the `range()` function. However, in practice, `list()` is *not* used with `range()`.)

`tuple`s are similar to `list`s but the elements of a `tuple` cannot be changed while the elements of a `list` can be, i.e., `tuple`s are *immutable* while `list`s are *mutable*.

`list`s and `tuple`s can be used in simultaneous assignments. They appear on the right side of the equal sign and the number of lvalues to the left of the equal sign must equal the number of elements in the `list` or `tuple`.

6.11 Review Questions

1. True or False: The length of a list is given by the `length()` function.

2. True or False: The index for the first element of a list is 1, e.g., `xlist[1]` is the first element of the list `xlist`.

3. What is the output produced by the following code?

```
xlist = []
xlist.append(5)
xlist.append(10)
print(xlist)
```

(a) `[5, 10]`

(b) `[]`

(c) `5, 10`

(d) `5 10`

(e) This produces an error.

(f) None of the above.

4. What is the output produced by the following code?

```
zlist = []
zlist.append([3, 4])
print(zlist)
```

(a) `[3, 4]`

(b) `[[3, 4]]`

(c) `3, 4`

(d) `3 4`

(e) None of the above.

5. What is the value of `xlist2` after the following statement has been executed?

```
xlist2 = list(range(-3, 3))
```

(a) `[-3, -2, -1, 0, 1, 2, 3]`

(b) `[-3, -2, -1, 0, 1, 2]`

(c) `[-2, -1, 0, 1, 2]`

(d) `[-3, 0, 3]`

(e) This produces an error.

6. What is the value of `xlist3` after the following statement has been executed?

```
xlist3 = list(range(-3, 3, 3))
```

(a) `[-3, 0, 3]`

(b) `[-3, 0]`

(c) `[-2, 1]`

(d) This produces an error.

7. What is the value of `xlist4` after the following statement has been executed?

```
xlist4 = list(range(-3))
```

(a) `[]`
(b) `[-3, -2, -1]`
(c) `[-3, -2, -1, 0]`
(d) This produces an error.

8. What is output produced by the following?

```
xlist = [2, 1, 3]
ylist = xlist.sort()
print(xlist, ylist)
```

(a) `[2, 1, 3] [1, 2, 3]`
(b) `[3, 2, 1] [3, 2, 1]`
(c) `[1, 2, 3] [2, 1, 3]`
(d) `[1, 2, 3] None`
(e) This produces an error.

9. To what value is the variable `x` set by the following code?

```
def multiply_list(start, stop):
    product = 1
    for element in range(start, stop):
        product = product * element
        return product

x = multiply_list(1, 4)
```

(a) `24`
(b) `6`
(c) `2`
(d) `1`

10. Consider the following function:

```
def f1(x, y):
    print([x, y])
```

True or False: This function returns a `list` consisting of the two parameters passed to the function.

11. Consider the following function:

```
def f2(x, y):
    return x, y
```

True or False: This function returns a list consisting of the two parameters passed to the function.

12. Consider the following function:

```
def f3(x, y):
    print(x, y)
    return [x, y]
```

True or False: This function returns a list consisting of the two parameters passed to the function.

13. Consider the following function:

```
def f4(x, y):
    return [x, y]
    print(x, y)
```

True or False: This function prints a list consisting of the two parameters passed to the function.

14. Consider the following function:

```
def f5(x, y):
    return [x, y]
    print([x, y])
```

True or False: This function prints a list consisting of the two parameters passed to the function.

15. What output is produced by the following code?

```
xlist = [3, 2, 1, 0]
for item in xlist:
   print(item, end=" ")
```

   (a) 3210
   (b) 3 2 1 0
   (c) [3, 2, 1, 0]
   (d) This produces an error.
   (e) None of the above.

16. What output is produced by the following code?

```
a = 1
b = 2
xlist = [a, b, a + b]
a = 0
b = 0
print(xlist)
```

(a) `[a, b, a` b]+

(b) `[1, 2, 3]`

(c) `[0, 0, 0]`

(d) This produces an error.

(e) None of the above.

17. What output is produced by the following code?

```
xlist = [3, 5, 7]
print(xlist[1] + xlist[3])
```

(a) `10`

(b) `12`

(c) `4`

(d) This produces an error.

(e) None of the above.

18. What output is produced by the following code?

```
xlist = ["aa", "bb", "cc"]
for i in [2, 1, 0]:
   print(xlist[i], end=" ")
```

(a) `aa bb cc`

(b) `cc bb aa`

(c) This produces an error.

(d) None of the above.

19. What does the following code do?

```
for i in range(1, 10, 2):
    print(i)
```

(a) Prints all odd numbers in the range [1, 9].

(b) Prints all numbers in the range [1, 9].

(c) Prints all even numbers in the range [1, 10].

(d) This produces an error.

20. What is the result of evaluating the expression `list(range(5))`?

    (a) `[0, 1, 2, 3, 4]`

    (b) `[1, 2, 3, 4, 5]`

    (c) `[0, 1, 2, 3, 4, 5]`

    (d) None of the above.

21. Which of the following headers is appropriate for implementing a counted loop that executes 4 times?

    (a) `for i in 4:`

    (b) `for i in range(5):`

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

    (d) `for i in range(1, 4):`

22. Consider the following program:

```
def main():
    num = eval(input("Enter a number: "))
    for i in range(3):
        num = num * 2
    print(num)

main()
```

Suppose the input to this program is 2, what is the output?

(a)
```
2
4
8
```

(b)
```
4
8
```

(c)
```
4
8
16
```

(d) `16`

23. The following fragment of code is in a program. What output does it produce?

```
fact = 1
for factor in range(4):
    fact = fact * factor
print(fact)
```

    (a) 120
    (b) 24
    (c) 6
    (d) 0

24. What is the output from the following program if the user enters 5.

```
def main():
    n = eval(input("Enter an integer: "))
    ans = 0
    for x in range(1, n):
        ans = ans + x
    print(ans)

main()
```

    (a) 120
    (b) 10
    (c) 15
    (d) None of the above.

25. What is the output from the following code?

```
s = ['s', 'c', 'o', 'r', 'e']
for i in range(len(s) - 1, -1, -1):
    print(s[i], end = " ")
```

    (a) s c o r e
    (b) e r o c s
    (c) 4 3 2 1 0
    (d) None of the above.

26. The following fragment of code is in a program. What output does it produce?

```
s = ['s', 'c', 'o', 'r', 'e']
sum = 0
for i in range(len(s)):
    sum = sum + s[i]
print(sum)
```

(a) score

(b) erocs

(c) scor

(d) 01234

(e) None of the above.

27. The following fragment of code is in a program. What output does it produce?

```
s = ['s', 'c', 'o', 'r', 'e']
sum = ""
for i in range(len(s)):
    sum =  s[i] + sum
print(sum)
```

(a) score

(b) erocs

(c) scor

(d) 01234

(e) None of the above.

28. What is the value returned by the following function when it is called with an argument of 3 (i.e., summer1(3))?

```
def summer1(n):
   sum = 0
   for i in range(1, n + 1):
      sum = sum + i
      return sum
```

(a) 3

(b) 1

(c) 6

(d) 0

29. What is the value returned by the following function when it is called with an argument of `4` (i.e., `summer2(4)`)?

```
def summer2(n):
    sum = 0
    for i in range(n):
        sum = sum + i
    return sum
```

   (a) `3`

   (b) `1`

   (c) `6`

   (d) `0`

30. Consider the following function:

```
def foo():
    xlist = []
    for i in range(4):
        x = input("Enter a number: ")
        xlist.append(x)
    return xlist
```

   Which of the following best describes what this function does?

   (a) It returns a list of four numbers that the user provides.

   (b) It returns a list of four strings that the user provides.

   (c) It returns a list of three numbers that the user provides.

   (d) It produces an error.

ANSWERS: 1) False; 2) False; 3) a; 4) b; 5) b; 6) b; 7) a; 8) d; 9) d (the `return` statement is in the body of the loop); 10) False (this is a void function); 11) False (this function returns a `tuple`); 12) True; 13) False (`print()` statement comes after the `return` statement and thus will not be executed); 14) False; 15) b; 16) b; 17) d; 18) b; 19) a; 20) a; 21) c; 22) d; 23) d; 24) b; 25) b; 26) e; 27) b; 28) b; 29) c; 30) b.

Chapter 7

More on `for`-Loops, Lists, and Iterables

The previous chapter introduced `lists`, `tuples`, the `range()` function, and `for`-loops. The reason for introducing these concepts in the same chapter is because either they are closely related (as is true with `lists` and `tuples`) or they are often used together (as is true, for example, with the `range()` function and `for`-loops). In this chapter we want to extend our understanding of the ways in which `for`-loops and iterables can be used. Although the material in this chapter is often presented in terms of `lists`, you should keep in mind that the discussion almost always pertains to `tuples` too—you could substitute a `tuple` for a `list` in the given code and the result would be the same. (This is not true only when it comes to code that assigns values to individual elements. Recall that `lists` are mutable but `tuples` are not. Hence, once a `tuple` is created, we cannot change its elements.)

The previous chapter mentioned that a `list` can have elements that are themselves `lists`, but no details were provided. In this chapter we will dive into some of these details. We will also consider nested `for`-loops, two new ways of indexing (specifically *negative indexing* and *slicing*), and the use of strings as *sequences* or iterables. We start by considering nested `for`-loops.

7.1 `for`-Loops within `for`-Loops

There are many algorithms that require that one loop be *nested* inside another. For example, nested loops can be used to generate data for a table where the inner loop dictates the column and the outer loop dictates the row. In fact, it is not uncommon for algorithms to require several levels of nesting (i.e., a loop within a loop within a loop and so on). As you will see, nested loops are also useful for processing data organized in the form of `lists` within a `list`. We start this section by showing some of the general ways in which `for`-loops can be nested. This is presented primarily in terms of generating various patterns of characters. After introducing `lists` of `lists` (in Sec. 7.2) we will consider more practical applications for nested `for`-loops.

Assume we want to generate the following output:

```
1
12
123
```

From the file: `more-on-iterables.tex`

```
4 1234
5 12345
6 123456
7 1234567
```

There are seven lines. The first line consists of a single `1`. Each successive line has one more digit than the previous line. This output can be generated using *nested* `for`-loops. Nested `for`-loops consist of an "outer" `for`-loop and an "inner" `for`-loop. The body of the outer `for`-loop is executed the number of times dictated by its header. Of course, the number of times the body of the inner loop is executed is also dictated by its header. However, the contents of the inner-loop's header can change with each pass of the outer loop.

Turning our attention back to the collection of characters above, we can use the outer loop to specify that we want to generate seven lines of output, i.e., the body of the outer loop will be executed seven times. We can then use the inner loop to generate the characters on each individual line.

Listing 7.1 shows nested `for`-loops that produce the arrangement of characters shown above.

Listing 7.1 Nested `for`-loops that generate seven lines of integers.

```
1  >>> for i in range(7):  # Header for outer loop.
2  ...     for j in range(1, i + 2): # Cycle through integers.
3  ...         print(j, end="")      # Suppress newline.
4  ...     print()                   # Add newline.
5  ...
6  1
7  12
8  123
9  1234
10 12345
11 123456
12 1234567
```

Line 1 contains the header of the outer `for`-loop. The body of this loop executes seven times because the argument of the `range()` function is `7`. Thus, the loop variable `i` takes on values `0` through `6`. The header for the inner `for`-loop is on line 2. This header contains `range(1, i + 2)`. Notice that the inner loop variable is `j`. Given the header of the inner loop, `j` will take on values between `1` and `i + 1`, inclusive. So, for example, when `i` is 1, corresponding to the *second* line of output, `j` varies between `1` and `2`. When `i` is 2, corresponding to the *third* line of output, `j` varies between `1` and `3`. This continues until `i` takes on its final value of `6` so that `j` varies between `1` and `7`.

The body of the inner `for`-loop, in line 3, consists of a `print()` statement that prints the value of `j` and suppresses the newline character (i.e., the optional argument `end` is set to the empty string). Following the inner loop, in line 4, is a `print()` statement with no arguments. This is used simply to generate the newline character. This `print()` statement is outside the

body of the inner loop but inside the body of the outer sloop. Thus, this statement is executed seven times: once for each line of output.[1]

Changing gears a bit, consider the following collection of characters. This again consists of seven lines of output. The first line has a single character and each successive line has one additional character.

```
1 &
2 &&
3 &&&
4 &&&&
5 &&&&&
6 &&&&&&
7 &&&&&&&
```

How do you implement this? You can use nested loops, but Python actually provides a way to generate this using a single `for`-loop. To do so, you need to recall string repetition which was introduced in Sec. 5.6. When a string is "multiplied" by an integer, a new string is produced that is the original string repeated the number of times given by the integer. So, for example, `"q" * 3` evaluates to the string `"qqq"`. Listing 7.2 shows two implementations that generate the collection of ampersands shown above: one implementation uses a single loop while the other uses nested loops.

Listing 7.2 A triangle of ampersands generated using a single `for`-loop or nested `for`-loops. The implementation with a single loop takes advantage of Python's string repetition capabilities.

```
1  >>> for i in range(7):          # Seven lines of output.
2  ...     print("&" * (i + 1))  # Num. characters increases as i increases.
3  ...
4  &
5  &&
6  &&&
7  &&&&
8  &&&&&
9  &&&&&&
10 &&&&&&&
11 >>> for i in range(7):            # Seven lines of output.
12 ...     for j in range(i + 1):  # Inner loop for ampersands.
13 ...         print("&", end="")
14 ...     print()                   # Newline.
15 ...
16 &
17 &&
18 &&&
```

[1] It may be worth mentioning that it is not strictly necessary to use two `for`-loops to obtain the output shown in Listing 7.1. As the code in the coming discussion suggests (but doesn't fully describe), it is possible to obtain the same output using a single `for`-loop (but then one has to provide a bit more code to construct the string that should appear on each line).

```
&&&&
&&&&&
&&&&&&
&&&&&&&
```

What if we wanted to invert this triangle so that the first line is the longest (with seven characters) and the last line is the shortest (with one character)? The code in Listing 7.3 provides a solution that uses a single loop. (Certainly other solutions are possible. For example, the `range()` function in the header of the `for`-loop can be used to directly generate the multipliers, i.e., integers that range from `7` to `1`. Then, the resulting loop variable can directly "multiply" the ampersand in the `print()` statement.)

Listing 7.3 An "inverted triangle" realized using a single `for`-loop.

```
>>> for i in range(7):          # Seven lines of output.
...     print("&" * (7 - i))  # Num. characters decreases as i increases.
...
&&&&&&&
&&&&&&
&&&&&
&&&&
&&&
&&
&
```

The header in line 1 is the same as the ones used previously: the loop variable `i` still varies between `0` and `6`. In line 2 the number of repetitions of the ampersand is `7 - i`. Thus, as `i` increases, the number of ampersands decreases.

As another example, consider the code given in Listing 7.4. The body of the outer `for`-loop contains, in line 2, a `print()` statement similar to the one in Listing 7.3 that was used to generate the inverted triangle of ampersands. Here, however, the newline character at the end of the line is suppressed. Next, in lines 3 and 4, a `for`-loop renders integers as was done in Listing 7.1. Outside the body of this inner loop a `print()` statement (line 5) simply generates a new line. Combining the inverted triangle of ampersands with the upright triangle of integers results in the rectangular collection of characters shown in lines 7 through 13.

Listing 7.4 An inverted triangle of ampersands is combined with an upright triangle of integers to form a rectangular structure of characters.

```
>>> for i in range(7):                  # Seven lines of output.
...     print("&" * (7 - i), end="") # Generate ampersands.
...     for j in range(1, i + 2):     # Inner loop to display digits.
...         print(j, end="")
...     print()                         # Newline.
...
```

```
7  &&&&&&&1
8  &&&&&&12
9  &&&&&123
10 &&&&1234
11 &&&12345
12 &&123456
13 &1234567
```

Using similar code, let's construct an upright *pyramid* consisting solely of integers (and blank spaces). This can be realized with the code in Listing 7.5. The first four lines of Listing 7.5 are identical to those of Listing 7.4 except the ampersand in line 2 has been replaced by a blank space. In Listing 7.5, the `for`-loop that generates integers of increasing value (i.e., the loop in lines 3 and 4), is followed by the `for`-loop that generates integers of decreasing value (lines 5 and 6). In a sense, the values generated by this second loop are tacked onto the right side of the rectangular figure that was generated in Listing 7.4.

Listing 7.5 Pyramid of integers that is constructed with an outer `for`-loop and two inner `for`-loops. The first inner loop, starting on line 3, generates integers of increasing value while the second loop, starting on line 5, generates integers of decreasing value.

```
1  >>> for i in range(7):
2  ...     print(" " * (7 - i), end="") # Generate leading spaces.
3  ...     for j in range(1, i + 2):     # Generate 1 through peak value.
4  ...         print(j, end="")
5  ...     for j in range(i, 0, -1):     # Generate peak - 1 through 1.
6  ...         print(j, end="")
7  ...     print()                       # Newline.
8  ...
9         1
10       121
11      12321
12     1234321
13    123454321
14   12345654321
15  1234567654321
```

Let's consider one more example of nested loops. Here, unlike in the previous examples, the loop variable for the outer loop does *not* appear in the header of the inner loop. Let's write a function that shows, as an ordered pair, the row and column numbers of positions in a table or matrix. Let's call this function `matrix_indices()`. It has two parameters corresponding to the number of rows and number of columns, respectively. In any previous experience you may have had with tables or matrices, the row and column numbers almost certainly started with one. Here, however, we use the numbering convention that is used for `lists`: the first row and column have an index of zero.

Listing 7.6 gives a function that generates the desired output.[2]

Listing 7.6 Function to display the row and column indices for a two-dimensional table or matrix where the row and column numbers start at zero.

```
1  >>> def matrix_indices(nrow, ncol):
2  ...     for i in range(nrow):      # Loop over the rows.
3  ...         for j in range(ncol): # Loop over the columns.
4  ...             print("(", i, ", ", j, ")", sep="", end=" ")
5  ...         print()
6  ...
7  >>> matrix_indices(3, 5)   # Three rows and five columns.
8  (0, 0) (0, 1) (0, 2) (0, 3) (0, 4)
9  (1, 0) (1, 1) (1, 2) (1, 3) (1, 4)
10 (2, 0) (2, 1) (2, 2) (2, 3) (2, 4)
11 >>> matrix_indices(5, 3)   # Five rows and three columns.
12 (0, 0) (0, 1) (0, 2)
13 (1, 0) (1, 1) (1, 2)
14 (2, 0) (2, 1) (2, 2)
15 (3, 0) (3, 1) (3, 2)
16 (4, 0) (4, 1) (4, 2)
```

The function defined in lines 1 through 5 has two parameters that are named `nrow` and `ncol`, corresponding to the desired number of rows and columns, respectively. `nrow` is used in the header of the outer loop in line 2 and `ncol` is used in the header of the inner loop in line 3.

In line 7 `matrix_indices()` is called to generate the ordered pairs for a matrix with three rows and five columns. The output appears in lines 8 through 10. In line 11 the function is called to generate the ordered pairs for a matrix with five rows and three columns.

In all the examples in this section the headers of the `for`-loops have used `range()` to set the loop variable to appropriate integer values. There is, however, another way in which nested `for`-loops can be constructed so that the iterables appearing in the headers are `list`s. This is considered in the next section.

7.2 `list`s of `list`s

A `list` can contain a `list` as an element or, in fact, contain any number of `list`s as elements. When one `list` is contained within another, we refer to this as *nesting* or we may say that one `list` is *embedded* within another. We may also refer to an *inner* `list` which is contained in a surrounding *outer* `list`. Nesting can be done to any level. Thus, for example, you can have a `list` that contains a `list` that contains a `list` and so on.

Listing 7.7 illustrates the nesting of one `list` within another.

---

[2]Unfortunately, if the user specifies that the number of rows or columns is greater than 11 (so that the row or column indices have more than one digit), the ordered pairs will no longer line up as nicely as shown here. When we cover string formatting, we will see ways to ensure the output is formatted "nicely" even for multiple digits.

Listing 7.7 Demonstration of nesting of one `list` as an element of another.

```
>>> # Create list with a string and a nested list of two strings.
>>> al = ['Weird Al', ['Like a Surgeon', 'Perform this Way']]
>>> len(al)  # Check length of al.
2
>>> al[0]
'Weird Al'
>>> al[1]
['Like a Surgeon', 'Perform this Way']
>>> for item in al:  # Cycle through the elements of list al.
...     print(item)
...
Weird Al
['Like a Surgeon', 'Perform this Way']
```

In line 2 the `list` `al` is defined with two elements. The first element is the string `'Weird Al'` and the second is a `list` that has two elements, both of which are themselves strings. The creation of this `list` immediately raises a question: How many elements does it have? Reasonable arguments can be made for either two or three but, in fact, as shown in lines 3 and 4, the `len()` function reports that there are two elements in `al`. Lines 5 and 6 show the first element of `al` and lines 7 and 8 show the second element, i.e., the second element of `al` is itself a complete two-element `list`. As before, a `for`-loop can be used to cycle through the elements of a `list`. This is illustrated in lines 9 through 13 where the `list` `al` is given as the iterable in the header.

Listing 7.8 provides another example of nesting one `list` within another; however, here there are actually three `lists` contained within the surrounding outer `list`. Importantly, as seen in lines 3 through 5, this code also demonstrates that the contents of a `list` can span multiple lines. The open bracket (`[`) acts similarly to open parentheses—it tells Python there is more to come. Thus, the `list` can be closed (with the closing bracket) on a subsequent line.[3]

Listing 7.8 Nesting of multiple `lists` inside a `list`. Here the `list` `produce` consists of `lists` that each contain a string as well as either one or two integers. The contents of a `list` may span multiple lines since an open bracket serves as a multi-line delimiter in the same way as an open parenthesis.

```
>>> # Create a list of three nested lists, each of which contains
>>> # a string and one or two integers.
>>> produce = [['carrots', 56],
...            ['celery', 178, 198],
...            ['bananas', 59]]
```

[3]Note, however, that any line breaks in the `list` must be between elements. One cannot, for instance, have a string that spans multiple lines just because it is enclosed in brackets. A string that spans multiple lines may appear in a `list` but it must adhere to the rules governing a multi-line string, i.e., it must be enclosed in triple quotes or the newline character at the end of each line must be escaped.

```
6  >>> print(produce)
7  [['carrots', 56], ['celery', 178, 198], ['bananas', 59]]
8  >>> for i in range(len(produce)):
9  ...     print(i, produce[i])
10 ...
11 0 ['carrots', 56]
12 1 ['celery', 178, 198]
13 2 ['bananas', 59]
```

In line 3 the `list produce` is created. Each element of this `list` is itself a `list`. These inner `lists` are composed of a string and one or two integers. The string corresponds to a type of produce and the integer might represent the price per pound (in cents) of this produce. When there is more than one integer, this might represent the price at different stores. The `print()` statement in line 6 displays the entire `list` as shown in line 7. The `for`-loop in lines 8 and 9 uses indexing to show the elements of the outer `list` together with the index of the element.

Let's consider a slightly more complicated example in which we create a `list` that contains three `lists`, each of which contains another `list`! The code is shown in Listing 7.9 and is discussed following the listing.

Listing 7.9 Creation of a `list` within a `list` within a `list`.

```
1  >>> # Create individual artists as lists consisting of a name and a
2  >>> # list of songs.
3  >>> al = ["Weird Al", ["Like a Surgeon", "Perform this Way"]]
4  >>> gaga = ["Lady Gaga", ["Bad Romance", "Born this Way"]]
5  >>> madonna = ["Madonna", ["Like a Virgin", "Papa Don't Preach"]]
6  >>> # Collect individual artists together in one list of artists.
7  >>> artists = [al, gaga, madonna]
8  >>> print(artists)
9  [['Weird Al', ['Like a Surgeon', 'Perform this Way']], ['Lady Gaga',
10  ['Bad Romance', 'Born this Way']], ['Madonna', ['Like a Virgin',
11  "Papa Don't Preach"]]]
12 >>> for i in range(len(artists)):
13 ...     print(i, artists[i])
14 ...
15 0 ['Weird Al', ['Like a Surgeon', 'Perform this Way']]
16 1 ['Lady Gaga', ['Bad Romance', 'Born this Way']]
17 2 ['Madonna', ['Like a Virgin', "Papa Don't Preach"]]
```

In lines 3 through 5, the lists `al`, `gaga`, and `madonna` are created. Each of these `lists` consists of a string (representing the name of an artist) and a `list` (where the `list` contains two strings that are the titles of songs by these artists). In line 7 the `list artists` is created. It consists of the three `lists` of individual artists. The `print()` statement in line 8 is used to print the `artists` lists.[4] The `for`-loop in lines 12 and 13 is used to display the `list` corresponding to each individual artist.

[4]Line breaks have been added to the output to aid readability. In the interactive environment the output would

7.2.1 Indexing Embedded lists

We know how to index the elements of a list, but now the question is: How do you index the elements of a list that is embedded within another list? To do this you simply add another set of brackets and specify within these brackets the index of the desired element. So, for example, if the third element of xlist is itself a list, the second element of this embedded list is given by xlist[2][1]. The code in Listing 7.10 illustrates this type of indexing.

Listing 7.10 Demonstration of the use of multiple brackets to access an element of a nested list.

```
1  >>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
2  >>> toyota[0]
3  'Toyota'
4  >>> toyota[1]
5  ['Prius', '4Runner', 'Sienna', 'Camry']
6  >>> toyota[1][0]
7  'Prius'
8  >>> toyota[1][3]
9  'Camry'
10 >>> len(toyota)      # What is length of outer list?
11 2
12 >>> len(toyota[1])   # What is length of embedded list?
13 4
14 >>> toyota[1][len(toyota[1]) - 1]
15 'Camry'
```

In line 1 the list toyota is created with two elements: a string and an embedded list of four strings. Lines 2 and 3 display the first element of toyota. In lines 4 and 5 we see that the second element of toyota is itself a list. In line 6 two sets of brackets are used to specify the desired element. Interpreting these from right to left, these brackets (and the integers they enclose) specify that we want the first element of the second element of toyota. The first set of brackets (i.e., the left-most brackets) contains the index 1, indicating the second element of toyota, while the second set of brackets contains the index 0, indicating the first element of the embedded list. Despite the fact that we (humans) might read or interpret brackets from right to left, Python evaluates multiple brackets from left to right. So, in a sense, you can think of toyota[1][0] as being equivalent to (toyota[1])[0], i.e., first we obtain the list toyota[1] and then from this we obtain the first element. You can, in fact, add parentheses in this way, but it isn't necessary or recommended. You should, instead, become familiar and comfortable with this form of multi-index notation as it is used in many computer languages.

Lines 10 and 11 of Listing 7.10 show the length of the toyota list is 2—as before, the embedded list counts as one element. Lines 12 and 13 show the length of the embedded list is 4. Line 14 uses a rather general approach to obtain the last element of a list (which here

be wrapped around at the border of the screen. This wrapping is not because of newline characters embedded in the output, but rather is a consequence of the way text is handled that is wider than can be displayed on a single line. Thus, if the screen size changes, the location of the wrapping changes correspondingly.

happens to be the embedded list given by toyota[1]). This approach, in which we subtract 1 from the length of the list, is really no different from the approach first demonstrated in Listing 6.7 for accessing the last element of any list.

As you may have guessed, a for-loop can be used to cycle through the elements of an embedded list. This is illustrated in the code in Listing 7.11.

Listing 7.11 Demonstration of cycling through the elements of an embedded list using a for-loop.

```
>>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
>>> for model in toyota[1]: # Cycle through embedded list.
...     print(model)
...
Prius
4Runner
Sienna
Camry
```

Line 1 again creates the toyota list with an embedded list as its second element. The for-loop in lines 2 and 3 prints each element of this embedded list (which corresponds to a Toyota car model).

Let us expand on this a bit and write a function that displays a car manufacturer (i.e., a brand or make) and the models made by each manufacturer (or at least a subset of the models—we won't bother trying to list them all). It is assumed that manufacturers are organized in lists where the first element is a string giving the brand name and the second element is a list of strings giving model names. Listing 7.12 provides the code for this function as well as examples of its use. The code is described following the listing.

Listing 7.12 A function to display the make and list of models of a car manufacturer.

```
>>> def show_brand(brand):
...     print("Make:", brand[0])
...     print("  Model:")
...     for i in range(len(brand[1])):
...         print("     ", i + 1, brand[1][i])
...
>>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
>>> ford = ["Ford", ["Focus", "Taurus", "Mustang", "Fusion", "Fiesta"]]
>>> show_brand(toyota)
Make: Toyota
  Model:
     1 Prius
     2 4Runner
     3 Sienna
     4 Camry
>>> show_brand(ford)
```

```
17 Make: Ford
18   Model:
19       1 Focus
20       2 Taurus
21       3 Mustang
22       4 Fusion
23       5 Fiesta
```

The show_brand() function is defined in lines 1 through 5. This function takes a single argument, named brand, that is a list that contains the make and models for a particular brand of car. Line 2 prints the make of the car while line 3 prints a label announcing that what follows are the various models. Then, in lines 4 and 5, a for-loop cycles through the second element of the brand list, i.e., cycles through the models. The print() statement in line 5 has four blank spaces, then a counter (corresponding to the element index plus one), and then the model. Lines 7 and 8 define lists that are appropriate for the Totoya and Ford brands of cars. The remaining lines show the output produced when these lists are passed to the show_brand() function.

Listing 7.13 presents the function show_nested_lists() that can be used to display all the (inner) elements of a list of lists. This function has a single parameter which is assumed to be the outer list.

Listing 7.13 Function to display the elements of lists that are nested within another list.

```
1  >>> def show_nested_lists(xlist):
2  ...     for i in range(len(xlist)):
3  ...         for j in range(len(xlist[i])):
4  ...             print("xlist[", i, "][", j, "] = ", xlist[i][j], sep="")
5  ...         print()
6  ...
7  >>> produce = [['carrots', 56], ['celery', 178, 198], ['bananas', 59]]
8  >>> show_nested_lists(produce)
9  xlist[0][0] = carrots
10 xlist[0][1] = 56
11
12 xlist[1][0] = celery
13 xlist[1][1] = 178
14 xlist[1][2] = 198
15
16 xlist[2][0] = bananas
17 xlist[2][1] = 59
```

The function is defined in lines 1 through 5. The sole parameter is named xlist. The for-loop whose header is in line 2 cycles through the indices for xlist. On the other hand, the for-loop with the header in line 3 cycles through the indices that are valid for the lists nested within xlist. The body of this inner for-loop consists of a single print() statement that incorporates the indices as well as the contents of the element. Importantly, note that the nested lists do not

have to be the same length. The first and third elements of xlist are each two-element lists. However, the second element of xlist (corresponding to the list with "celery") has three elements.

7.2.2 Simultaneous Assignment and lists of lists

We have seen that lists can be used with simultaneous assignment (Sec. 6.8). When dealing with embedded lists, simultaneous assignment is sometimes used to write code that is much more readable than code that does not employ simultaneous assignment. For example, in the code in Listings 7.11 and 7.12 the list toyota was created. The element toyota[0] contains the make while toyota[1] contains the models. When writing a program that may have multiple functions and many lines of code, it may be easy to lose sight of the fact that a particular element maps to a particular thing. One way to help alleviate this is to "unpack" a list into parts that are associated with appropriately named variables. This unpacking can be done with simultaneous assignment. Listing 7.14 provides an example.[5]

Listing 7.14 Demonstration of "unpacking" a list that contains an embedded list. Simultaneous assignment is used to assign the elements of the toyota list to appropriately named variables.

```
1  >>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
2  >>> make, models = toyota      # Simultaneous assignment.
3  >>> make
4  'Toyota'
5  >>> for model in models:
6  ...     print("    ", model)
7  ...
8       Prius
9       4Runner
10      Sienna
11      Camry
```

In line 1 the toyota list is created with the brand name (i.e., the make) and an embedded list of models. Line 2 uses simultaneous assignment to "unpack" this two-element list to two appropriately named variables, i.e., make and models. Lines 3 and 4 show make is correctly set. The for-loop in lines 5 and 6 cycles through the models list to produce the output shown in lines 8 through 11.

Let us consider another example where the goal now is to write a function that cycles through a list of artists similar to the list that was constructed in Listing 7.9. Let's call the function show_artists(). This function should cycle through each element (i.e., each artist) of the list it is passed. For each artist it should display the name and the songs associated with the artist. Listing 7.15 shows a suitable implementation of this function.

[5]This is somewhat contrived in that a list is created and then immediately unpacked. Try instead to imagine this type of unpacking being done in the context of a much larger program with multiple lists of brands and lists being passed to functions.

Listing 7.15 A function to display a `list` of artists in which each element of the artists `list` contains an artist's name and a `list` of songs.

```
>>> def show_artists(artists):
...     for artist in artists:          # Loop over the artists.
...         name, songs = artist        # Unpack name and songs.
...         print(name)
...         for song in songs:          # Loop over the songs.
...             print("    ", song)
...
>>> # Create a list of artists.
>>> performers = [
...     ["Weird Al", ["Like a Surgeon", "Perform this Way"]],
...     ["Lady Gaga", ["Bad Romance", "Born this Way"]],
...     ["Madonna", ["Like a Virgin", "Papa Don't Preach"]]]
>>> show_artists(performers)
Weird Al
     Like a Surgeon
     Perform This Way
Lady Gaga
     Bad Romance
     Born This Way
Madonna
     Like a Virgin
     Papa Don't Preach
```

The `show_artists()` function is defined in lines 1 through 6. Its sole parameter is named `artists` (plural). This parameter is subsequently used as the iterable in the header of the `for`-loop in line 2. The loop variable for this outer loop is `artist` (singular). Thus, given the presumed structure of the `artists` list, for each pass of the outer loop, the loop variable `artist` corresponds to a two-element `list` containing the artist's name and a `list` of songs. In line 3 the loop variable `artist` is unpacked into a `name` and a `list` of `songs`. Note that we don't explicitly know from the code itself that, for example, `songs` is a `list` nor even that `artist` is a two-element `list`. Rather, this code relies on the presumed structure of the `list` that is passed as an argument to `show_artists()`. Having obtained the `name` and `songs`, these values are displayed using the `print()` statement in line 4 and the `for`-loop in lines 5 and 6.

Following the definition of the function, in lines 9 through 12, a `list` named `performers` is created that is suitable for passing to `show_artists()`. The remainder of the listing shows the output produced when `show_artists()` is passed the `performers list`.

Perhaps somewhat surprisingly, simultaneous assignment can be used directly in the header of a `for`-loop. To accomplish this, each item of the iterable in the header must have nested within it as many elements as there are lvalues to the left of the keyword `in`. Listing 7.16 provides a template for using simultaneous assignment in a `for`-loop header. In the header, there are N comma-separated lvalues to the left of the keyword `in`. There must also be N elements nested within each element of the iterable.

Listing 7.16 Template for a for-loop that employs simultaneous assignment in the header.

```
for <item1>, ..., <itemN> in <iterable>:
    <body>
```

Listing 7.17 provides a demonstration of this form of simultaneous assignment. In lines 1 through 3 a list is created of shoe designers and, supposedly, the cost of a pair of their shoes. Note that each element of the shoes list is itself a two-element list. The discussion of the code continues following the listing.

Listing 7.17 Demonstration of simultaneous assignment used in the header of a for-loop. This sets the value of two loop variables in accordance with the contents of the two-element lists that are nested in the iterable (i.e., nested in the list shoes).

```
>>> shoes = [["Manolo Blahnik", 120], ["Bontoni", 96],
...          ["Maud Frizon", 210], ["Tinker Hatfield", 54],
...          ["Lauren Jones", 88], ["Beatrix Ong", 150]]
>>> for designer, price in shoes:
...     print(designer, ": $", price, sep="")
...
Manolo Blahnik: $120
Bontoni: $96
Maud Frizon: $210
Tinker Hatfield: $54
Lauren Jones: $88
Beatrix Ong: $150
```

The header of the for-loop in line 4 uses simultaneous assignment to assign a value to both the loop variables designer and price. So, for example, prior to the first pass through the body of the loop it is as if this statement had been issued:

```
    designer, price = shoes[0]
```

or, thought of in another way

```
    designer, price = ["Manolo Blahnik", 120]
```

Then, prior to the second pass through the body of the for-loop, it is as if this statement had been issued

```
    designer, price =  ["Bontoni", 96]
```

And so on. The body of the for-loop consists of a print() statement that displays the designer and the associated price (complete with a dollar sign).

As illustrated in Listing 7.18, this form of simultaneous assignment works when dealing with tuples as well. In line 2 a list of two-element tuples is created. The for-loop in lines 3

and 4 cycles through these tuples, assigning the first element of the tuple to count and the second element of the tuple to fruit. The body of the loop prints these values.

Listing 7.18 Demonstration that tuples and lists behave in the same way when it comes to nesting and simultaneous assignment in for-loop headers.

```
>>> # Create a list of tuples.
>>> flist = [(21, 'apples'), (17, 'bananas'), (39, 'oranges')]
>>> for count, fruit in flist:
...     print(count, fruit)
...
21 apples
17 bananas
39 oranges
>>> # Create a tuple of tuples.
>>> ftuple = ((21, 'apples'), (17, 'bananas'), (39, 'oranges'))
>>> for count, fruit in ftuple:
...     print(count, fruit)
...
21 apples
17 bananas
39 oranges
```

Lines 10 through 16 are nearly identical to lines 2 through 8. The only difference is that the data are collected in a tuple of tuples rather than a list of tuples.

7.3 References and list Mutability

In Sec. 4.4 we discussed the scope of variables. We mentioned that we typically want to pass data into a function via the parameters and obtain data from a function using a return statement. However, because a list is a mutable object, the way it behaves when passed to a function is somewhat different than what we have seen before. In fact, we don't even have to pass a list to a function to observe this different behavior. In this section we want to explore this behavior. The primary goal of the material presented here is to help you understand the cause of bugs that may appear in your code. We do not seek to fully flesh out the intricacies of data manipulation and storage in Python.

First, recall that when we assign a value, such as an integer or float, to a variable and then assign that variable to a second variable, changes to the first variable do not subsequently affect the second variable (or vice versa). This is illustrated in the following (please read the comments embedded in the code):

```
>>> x = 11          # Assign 11 to x.
>>> y = x           # Assign value of x to y
>>> print(x, y)    # Show x and y.
11 11
```

```
5  >>> x = 22         # Change x to 22.
6  >>> print(x, y)   # Show that x has changed, but not y.
7  22 11
8  >>> y = 33         # Now change y to 33.
9  >>> print(x, y)   # Show that y has changed, but not x.
10 22 33
```

This is *not* how `lists` behave. In Python a `list` may have hundreds, thousands, or even millions of elements—the data within a `list` may occupy a significant amount of computer memory. Thus, when a `list` is assigned to a new variable the default action is *not* to make an independent copy of the underlying data for this new variable. Instead, the assignment makes the new variable a *reference* (or an *alias*) that points to the same underlying data (i.e., it points to the same memory location where the original `list` is stored). This is demonstrated in Listing 7.19 which parallels the code above except now the data is contained in a `list`.

Listing 7.19 Demonstration that when a `list` is assigned to a variable, the variable serves as a reference (or alias) to the underlying data in memory.

```
1  >>> xlist = [11]           # Create xlist with a single element.
2  >>> ylist = xlist          # Make ylist an alias for xlist.
3  >>> print(xlist, ylist)   # Show xlist and ylist.
4  [11] [11]
5  >>> xlist[0] = 22          # Change value of the element in xlist.
6  >>> print(xlist, ylist)   # Change is visible in both xlist and ylist.
7  [22] [22]
8  >>> ylist[0] = 33          # Change value of the element in ylist.
9  >>> print(xlist, ylist)   # Change is visible in both xlist and ylist.
10 [33] [33]
11 >>> # Use built-in is operator to show xlist and ylist are the same.
12 >>> xlist is ylist
13 True
```

In line 1 a single-element `list` is created and assigned to the variable `xlist`. In line 2 `xlist` is assigned to `ylist`. This assignment makes `ylist` point to the same location in the computer's memory that `xlist` points to, i.e., they both point to the single-element `list` that contains the integer `11`. The `print()` statement in line 3 and the subsequent output in line 4 show that the contents of `xlist` and `ylist` are the same. In line 5 the value of the element in `xlist` is changed to `22`. The `print()` statement in line 6 shows this change is reflected in the contents of both `xlist` and `ylist`! In line 8 the value of the element within `ylist` is changed to `33`. The subsequent `print()` statement in line 9 shows this change is also evident in both `lists`. This behavior is a consequence of the fact that there is really only one location in memory where this `list` is stored and both `xlist` and `ylist` point to this location.

There is a built-in operator called `is` that can be used to determine whether two variables point to the same memory location. The `is` operator is used in line 12 to ask if `xlist` and `ylist` point to the same memory location. The answer, shown on line 13, is `True`—they do point to the same

memory location. Thus, because *a `list` is mutable*, a change to the `list` made using one of the variables will affect the underlying data seen by both variables.

Now, let's consider code that may initially look the same as the code in Listing 7.19; however, there is an important distinction. The first four lines of Listing 7.20 are identical to those of Listing 7.19. A single-element `list` is created and assigned to the variable `xlist` which, in turn, is assigned to `ylist` in line 2. The `is` operator is used in line 5 to show that `xlist` and `ylist` are aliases for the same data. The discussion continues following the listing.

Listing 7.20 As demonstrated in the following, although two variables may initially be aliases for the same underlying data, if one of the variables is assigned new data, this will not affect the original data.

```
1  >>> xlist = [11]          # Create xlist with a single element.
2  >>> ylist = xlist         # Make ylist an alias for xlist.
3  >>> print(xlist, ylist)   # Show xlist and ylist.
4  [11] [11]
5  >>> xlist is ylist        # See that xlist and ylist are aliases.
6  True
7  >>> xlist = [22]          # Change xlist to point to a new list.
8  >>> print(xlist, ylist)   # See that xlist changes, but not ylist.
9  [22] [11]
10 >>> xlist is ylist   # See that xlist and ylist are no longer the same.
11 False
```

In line 7 `xlist` is assigned to a new one-element `list`, i.e., `xlist` now points to a new `list`. Note, importantly, that in line 7 we are *not* changing the value of an element in the `list` that was assigned to `xlist` in line 1. (Please compare line 7 of Listing 7.20 to line 5 of Listing 7.19.) Instead, we are assigning `xlist` to an entirely new `list`. This assignment does not affect where `ylist` points in memory. Thus, the output of the `print()` statement in line 8 shows that `xlist` and `ylist` are now different. The `is` operator is used in line 10 to test if `xlist` and `ylist` are aliases for the same underlying data. The result of `False` in line 11 shows they are not.

With the understanding that assignment of a `list` to a variable actually creates a reference (or alias) to the underlying data in the computer's memory, let us consider passing a `list` to a function and the consequences of changing the elements of the `list` within the function. Listing 7.21 starts by defining a function in lines 1 through 3. This function takes a single argument which is assumed to be a `list`. The function contains a `for`-loop that multiplies each of the elements by `2` and assigns this value back to the original location in the `list`. Note that this function has *no* `return` statement. Thus, it is a void function—it does not return anything useful. Instead, we would call this function for its side effects, i.e., its ability to double the elements of a `list`. The discussion continues below the listing.

Listing 7.21 Demonstration that changes to a `list` that occur inside a function are visible outside the function.

```
1  >>> def doubler(xlist):
```

```
2 ...     for i in range(len(xlist)):
3 ...         xlist[i] = 2 * xlist[i]
4 ...
5 >>> zlist = [1, 2, 3]  # Create list of integers.
6 >>> print(zlist)       # Check contents.
7 [1, 2, 3]
8 >>> doubler(zlist)     # Call doubler().
9 >>> print(zlist)       # See that contents have doubled.
10 [2, 4, 6]
```

In line 5 zlist is created with three integer elements. The print() statement in line 6 displays the contents of the list. The doubler() function is called in line 8. Thus, within the function itself, xlist (the formal parameter of the function) serves as an alias for zlist. We see, with the print() statement in line 9, that the elements of zlist were indeed doubled.

To further illustrate how lists behave when involved in assignment operations, Fig. 7.1 depicts the contents of a namespace as various statements involving a list are executed. Recall that the "Name" portion of a namespace is a collection of identifiers and it is the role of the namespace to associate these identifiers with objects. In Python, a list is a form of "container." As such, each element of a list can reference (i.e., be associated with) any other type of object. In Fig. 7.1(a) we assume the statement x = [2, "two"] has been issued. The literal list [2, "two"] is stored in memory by creating a container of two elements. This is depicted by [·,·] where the dots can be references to any other object. The first element of this container refers to the integer 2 while the second element refers to the string two. The assignment operator is used to associate the indentifier x with this list.

Figure 7.1(b) indicates the effect of using the append() method to append an object to the list: the container grows by an additional element. Note that the expression that appears in the argument of the append() method is evaluated prior to appending the object to the end of the list. In Fig. 7.1(c) the identifier y is assigned x. This merely serves to associate y with the same object to which x currently refers. In Fig. 7.1(d) the second element of y is assigned x the sum of the first and third elements of x. But, there is only one list stored in memory so we would have used any combination of the identifiers x and y in this statement and the outcome would have been the same. In Fig. 7.1(e), the identifier x is assigned the integer value 5. Thus, x no longer refers to the list, but this assignment does *not* affect the value of y.

Now consider the namespace depicted in Fig. 7.2 where the statement that gave rise to this arrangement of objects is given above the namespace. To the right of the assignment operator is a list of lists. Thus, some of the elements of the various containers refer to other containers. The object associated with any element of any of these lists can be changed because lists are mutable and there are no restrictions to what an element of a container refers. Furthermore, the lengths of the lists may be changed using the appropriate methods.[6]

[6]The methods append() and extend() will increase the length of a list while pop() and remove() can be used to remove elements from a list and hence decrease the length.

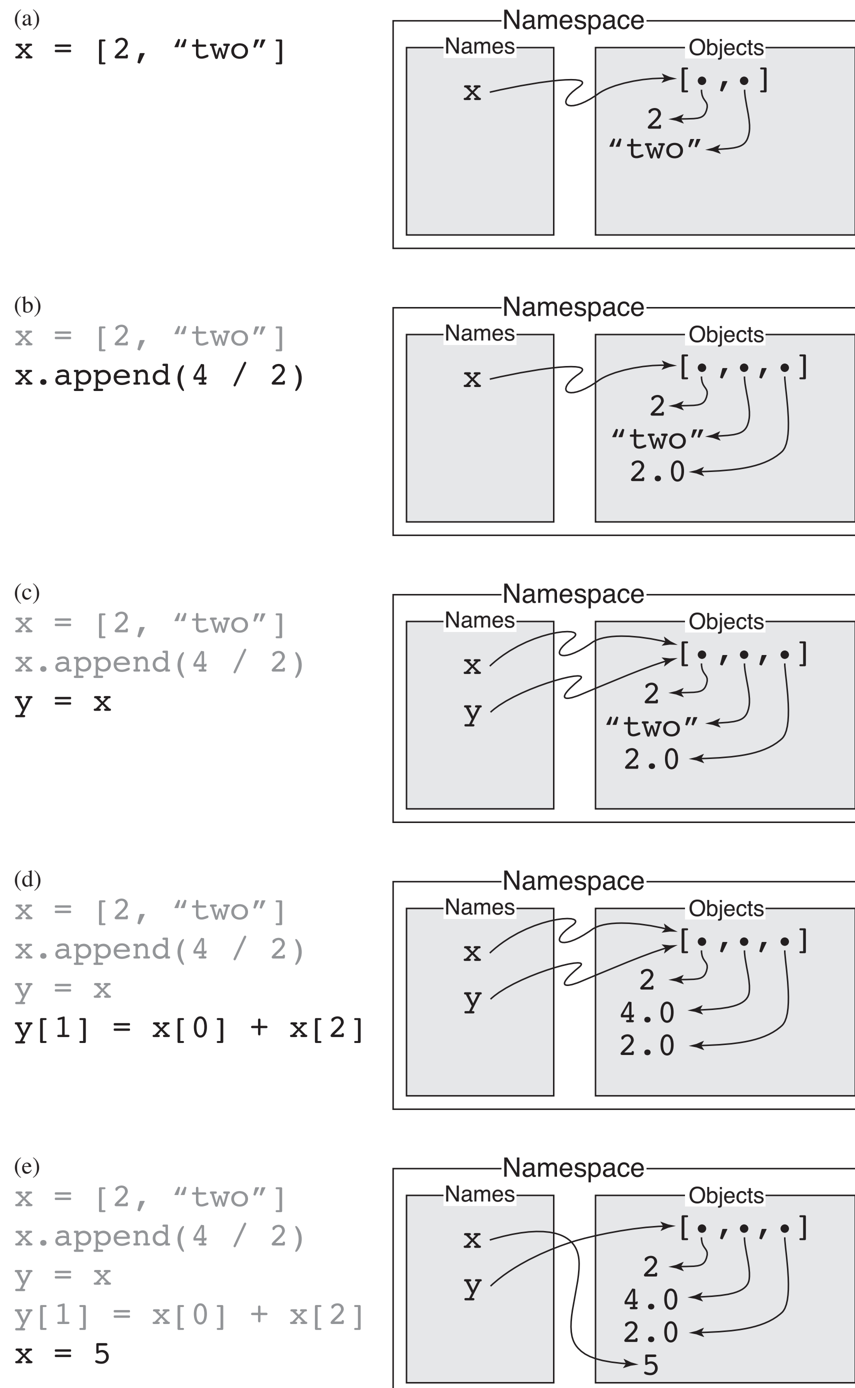


Figure 7.1: Depiction of the behavior of a namespace when various statements are issued involving a `list`.

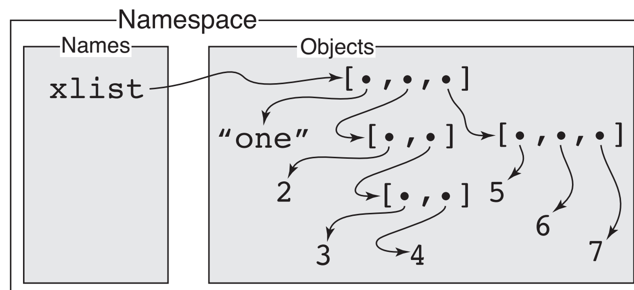


Figure 7.2: Depiction of a namespace after the assignment of a `list` of `lists` to the identifier `xlist`. The statement corresponding to the depiction is shown above the namespace.

7.4 Strings as Iterables or Sequences

We will have much more to say about strings in Chap. 9, but it is appropriate now to consider a couple of aspects of strings. As you know, strings are a sequential collection of characters. As such, each character has a well defined position within the string. This is similar to the elements within `lists` and `tuples`. Thus, it may come as no surprise that we can access individual characters in a string by indicating the desired character with an integer index enclosed in square brackets following the string. As with `lists` and `tuples`, an index of `0` is used for the first character of a string—the index represents the offset from the beginning. Additionally, the `len()` function provides the length of a string, i.e., the total number of characters. The individual characters of a string are themselves strings (with lengths of 1).[7]

Listing 7.22 illustrates accessing individual characters of a string by explicit indexing.

Listing 7.22 Explicit indexing used to access the characters of a string. As with `tuples` and `lists`, the index represents the offset from the beginning.

```
1  >>> hi = "Hello World!"
2  >>> hi[0]                # First character.
3  'H'
4  >>> hi[6]                # Seventh character.
5  'W'
6  >>> len(hi)              # Length of string.
7  12
8  >>> hi[len(hi) - 1]      # Last character.
9  '!'
10 >>> print(hi[0], hi[1], hi[5], hi[7], hi[9], hi[10], hi[11], sep = "")
11 He old!
```

[7] In some computer languages individual characters have a different data type than strings, but this is *not* the case in Python.

Listing 7.23 demonstrates how a for-loop can be used with explicit indexing to loop over all the characters in a string. In line 1 the string "Hey!" is assigned to the variable yo. The header of the for-loop in line 2 is constructed to cycle through all the valid indices for the characters of yo. The body of the for-loop, in line 3, prints the index and the corresponding character.

Listing 7.23 Use of a for-loop and explicit indexing to loop over all the characters in a string.

```
1 >>> yo = "Hey!"
2 >>> for i in range(len(yo)):
3 ...     print(i, yo[i])
4 ...
5 0 H
6 1 e
7 2 y
8 3 !
```

There is another way to loop over the individual characters of a string. Another feature that strings share with lists and tuples is that they are iterables[8] and thus can be used directly as the iterable in the header of a for-loop. This is demonstrated in Listing 7.24.

Listing 7.24 Demonstration that a string can be used as the iterable in the header of a for-loop. When a string is used this way, the loop variable is successively set to each individual character of the string for each pass of the for-loop.

```
1 >>> yo = "Hey!"
2 >>> for ch in yo:
3 ...     print(ch)
4 ...
5 H
6 e
7 y
8 !
```

Note that individual characters of strings can be accessed via an index. Similarly, individual elements of tuples and lists can be accessed via an index. Objects whose contents can be accessed in this way are known as *sequences*.[9] Recall that a tuple differs from a list in that a tuple is immutable while a list is mutable (its elements can be changed). A string is similar to a tuple in that the individual characters of a string cannot be changed, i.e., a string is immutable.

[8]Technically, an iterable is any object that has either a __iter__() or __getitem__() method. Behind the scenes Python uses these methods to get the values of the iterable. But, this sort of detail really doesn't concern us.

[9]A sequence is an object that has a __getitem__() method. Thus, sequences are automatically iterables, but the converse is not true: not all iterables are sequences. Assume s is a sequence (whether a string, list, or tuple). When we write s[i] Python essentially translates this to s.__getitem__(i). You can, in fact, use this method call yourself to confirm that the brackets are really just providing a shorthand for the calling of __getitem__().

7.5 Negative Indices

In addition to the positive indexing we have been using, an element of a sequence can also be specified using a *negative* index. In terms of negative indexing, the last element of a sequence has an index of `-1`. When the last element of a sequence is desired, it is generally more convenient to use negative indexing. For example, if one is interested in the last element of the `list` `xlist`, it is certainly more convenient to write `xlist[-1]` than `xlist[len(xlist) - 1]`.

Listing 7.25 shows the positive and negative indices for the string `"Hello!"`. The positive indices vary between zero and one less than the length (corresponding to the first and last characters, respectively). The negative indices vary between the negative of the length to `-1` (again, corresponding to the first and last characters, respectively).

Listing 7.25 Positive and negative indices for the string `"Hello!"`.

```
   0   1   2   3   4   5    <=> Positive indices
 +---+---+---+---+---+---+
 | H | e | l | l | o | ! |
 +---+---+---+---+---+---+
  -6  -5  -4  -3  -2  -1    <=> Negative indices
```

Listing 7.26 demonstrates negative indexing to access characters of a string. A six-character string is created in line 1 and assigned to the variable `hi`. The first element is accessed in line 2 by explicitly putting the negative index corresponding to the first character. In line 4 the `len()` function is used to obtain an expression that, in general, produces the first character of a string of arbitrary length. In line 6 the last character of the string is accessed.

Listing 7.26 Demonstration of negative indexing for a string. Elements of a `tuple` or a `list` can also be accessed using negative indexing.

```
1  >>> hi = "Hello!"    # Create a six-character string.
2  >>> hi[-6]           # Explicitly access first element.
3  'H'
4  >>> hi[-len(hi)]     # General construct for accessing first element.
5  'H'
6  >>> hi[-1]           # Last element of string.
7  '!'
8  >>> for i in range(1, len(hi) + 1):
9  ...     print(-i, hi[-i])
10 ...
11 -1 !
12 -2 o
13 -3 l
14 -4 l
15 -5 e
16 -6 H
```

The for-loop in lines 8 and 9 cycles through all the negative indices which are displayed in the output together with the corresponding character.

Although these examples used strings, negative indexing behaves the same way when accessing the elements of tuples and lists.

7.6 Slicing

Slicing provides a convenient way to extract or access a subset of a sequence. In many ways slicing is akin to the two-argument or three-argument forms of the range() function. To obtain a slice of a sequence, one specifies an integer start value and an integer stop value. These values are given in square brackets following the sequence (where one would normally provide a single index). These start and stop values are separated by a colon. The resulting slice is the portion of the original sequence that starts from the index start and ends just before stop. For example, assuming xlist is a list and that the start and stop values are both non-negative, the expression

```
xlist[start : stop]
```

returns a new list with elements

```
[xlist[start], xlist[start + 1], ..., xlist[stop - 1]]
```

If start is omitted, it defaults to 0. If stop is omitted, it defaults to the length of the sequence.

As a slight twist on the description above, positive or negative indices can be used for either the start or stop value. As before, when one index is positive and the other negative, the resulting slice starts with the specified start index and progresses up to, but does not include, the stop index. However, when using indices of mixed sign, one cannot interpret the resulting slice in terms of the indexing shown above (where the index of successive elements is incremented by +1 from the start). Instead, one has to translate a negative index into its equivalent positive value. This fact will become more clear after considering a few examples.

Listing 7.27 provides various examples of slicing performed on a string. The comments in the code provide additional information.

Listing 7.27 Demonstration of slicing a string.

```
1  >>> #    0123456789012345  -> index indicator
2  >>> s = "Washington State" # Create a string.
3  >>> s[1 : 4]     # Second through fourth characters.
4  'ash'
5  >>> s[ : 4]      # Start through fourth character.
6  'Wash'
7  >>> s[4 : 6]     # Fifth and sixth characters.
8  'in'
9  >>> s[4 : ]      # Fifth character through the end.
10 'ington State'
11 >>> s[-3 : ]     # Third character from end to end.
```

```
12 'ate'
13 >>> # Fifth from start to (but not including) third from end.
14 >>> s[4 : -3]    # Equivalent to s[4 : 13]
15 'ington St'
16 >>> # Tenth from end through tenth from start.
17 >>> s[-10 : 10] # Equivalent to s[6 : 10]
18 'gton'
19 >>> s[-9 : -2]  # Negative start & stop.
20 'ton Sta'
21 >>> s[7 : 14]   # Positive start & stop; equiv. to previous expression.
22 'ton Sta'
23 >>> s[ : ]         # The entire sequence.
24 'Washington State'
```

In line 14 and line 17 slices are obtained that use indices with mixed signs. The slice of `s[4 : -3]` is equivalent to the slice `s[4 : 13]`. Note that if the `range()` function were passed these `start` and `stop` values, i.e., `range(4, -3)`, it would not produce any integers. Thus, although slicing is closely related to the `range()` function, there are some differences.

Note the expression in line 23. Both the `start` and `stop` values are omitted so that these take on the default values of `0` and the length of the string, respectively. Thus, the resulting slice, shown in line 24, is the entire string. This last expression may seem rather silly, but it can actually prove useful in that slices return copies of the original sequence. If the slice is specified by `[ : ]`, then Python will make a copy of the *entire* sequence. This can come in handy when one needs to make a copy of an entire `list`.

Listing 7.28 provides an example where an "original" `list` is assigned to `xlist` in line 1. A slice that spans all of `xlist` is assigned to `ylist` in line 2. In line 3 `zlist` is simply assigned the value `xlist`. As we know from Listing 7.19, `zlist` is an alias for `xlist`. But, `ylist` is a completely independent copy of the original data. As lines 6 through 11 illustrate, changes to `xlist` do not affect `ylist` and, likewise, changes to `ylist` do not affect `xlist`. The `is` operator is used in line 15 to confirm what we already know: `xlist` and `ylist` point to different memory locations.

Listing 7.28 Demonstration that a slice can be used to obtain an independent copy of an entire `list`.

```
1 >>> xlist = [1, 2, 3]    # Original list.
2 >>> ylist = xlist[ : ]  # Independent copy of list.
3 >>> zlist = xlist        # Alias for original list.
4 >>> print(xlist, ylist, zlist) # Show lists.
5 [1, 2, 3] [1, 2, 3] [1, 2, 3]
6 >>> xlist[1] = 200               # Change original list.
7 >>> print(xlist, ylist, zlist) # See change in original but not copy.
8 [1, 200, 3] [1, 2, 3] [1, 200, 3]
9 >>> ylist[2] = 300               # Change copy.
10 >>> print(xlist, ylist, zlist) # See change in copy but not "original."
11 [1, 200, 3] [1, 2, 300] [1, 200, 3]
```

```
12 >>> zlist[0] = 100               # Change original via the alias.
13 >>> print(xlist, ylist, zlist) # See change in original but not copy.
14 [100, 200, 3] [1, 2, 300] [100, 200, 3]
15 >>> xlist is ylist    # Use is to show original and copy are different.
16 False
```

A slice can be specified with three terms (or three arguments). The third term is the *step* or *increment* and it is separated from the `stop` value by a colon. So, for the `list` `xlist`, one can write

```
xlist[start : stop : increment]
```

When the increment is not provided, it defaults to `1`. The increment may be negative. When the increment is negative, this effectively inverts the default values for `start` and `stop`. For a negative increment the default `start` is `-1` (i.e., the last element or character) and the default `start` is the negative of the quantity length plus one (e.g., `-(len(xlist) + 1)`). As will be demonstrated in a moment, by doing this, if one uses an increment of `-1` and the default `start` and `stop`, the entire sequence is inverted.

Listing 7.29 demonstrates slicing where an increment is specified. In lines 2, 4, and 7 an increment of `2` is used to obtain every other character over the specified `start` and `stop` values. In lines 9, 11, and 13 the increment is `-1` so that characters are displayed in reverse order. The expression in line 9 inverts the entire string as does the expression in line 13. However the expression in line 9 relies on the default `start` and `stop` values while the expression in line 13 gives these explicitly.

Listing 7.29 Demonstration of slicing where the increment is provided as the third term in brackets.

```
1  >>> s = "Washington State"
2  >>> s[ : : 2]       # Every other character from the start.
3  'Wsigo tt'
4  >>> s[ 5 : : 2]     # Every other character from the sixth.
5  'ntnSae'
6  >>> # Every other character from sixth, excluding last character.
7  >>> s[ 5 : -1 : 2]
8  'ntnSa'
9  >>> s[ : : -1]      # Negative increment -- invert string.
10 'etatS notgnihsaW'
11 >>> s[10 : 1 : -1] # Eleventh character to third; negative increment.
12 ' notgnihs'
13 >>> s[-1 : -len(s) - 1 : -1]
14 'etatS notgnihsaW'
```

For slices it is not an error to specify `start` or `stop` values that are out of range, i.e., specify indices that are before the beginning or beyond the end of the given sequence. It is, however, an error to specify an individual index that is out of range. This is demonstrated in Listing 7.30. A

three-element list is created in line 1. In line 2 a slice is created that specifies the start is the second element and the stop value is beyond the end of the list. The result, shown in line 2, starts at the second element and goes until the end of the list. Similarly, in line 4, the start value is before the first element and the stop value says to stop before the second element. The result, in line 5, is simply the first element of the list. In line 6 the start value is before the start of the list and the stop value is beyond the end. In this case, the result is the entire list as shown in line 7. The next two statements, in lines 8 and 12, show that if we try to access an individual element outside the range of valid indices, we obtain an IndexError.

Listing 7.30 Demonstration that out-of-range values can be given for a slice but cannot be given for an individual item in a sequence.

```
1  >>> xlist = ['a', 'b', 'c']
2  >>> xlist[1 : 100]          # Slice: Stop value beyond end.
3  ['b', 'c']
4  >>> xlist[-100 : 1]         # Slice: Start value before beginning.
5  ['a']
6  >>> xlist[-100 : 100]       # Slice: Start and stop out of range.
7  ['a', 'b', 'c']
8  >>> xlist[-100]             # Individual element out of range.
9  Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 IndexError: list index out of range
12 >>> xlist[100]              # Individual element out of range.
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in <module>
15 IndexError: list index out of range
```

7.7 list Comprehension (optional)

We often need to iterate through the elements of a list, perform operation(s) on those elements, and store the resulting values in a new list, leaving the original list unchanged. Although this can be accomplished using a for-loop, to make this pattern easier to implement and more concise, Python provides a construct known as *list comprehension*. The syntax may look strange at first, but if you understand the fundamentals of lists and for-loops you have all the tools you need to understand list comprehensions![10]

Recall the function doubler() from Listing 7.21 that doubled the elements of a list. This doubling was done "in place" such that the list provided as an argument to the function had its elements doubled. Now we want to write a function called newdoubler() that, like doubler(), doubles every element of the list provided as an argument. However, newdoubler() does not modify the original list argument. Instead, it returns a new list whose elements are double

[10]Despite the elegance and utility of list comprehensions, they will not be used in the remainder of this book and hence the material in this section is not directly relevant to any material in the subsequent chapters.

those of the list argument while the elements of the original list are unchanged. (You can think of the "new" of newdoubler() as being indicative of the fact that this function returns a *new* list.) Listing 7.31 provides the code for newdoubler() and demonstrates its behavior. The code is discussed further following the listing.

Listing 7.31 A modified version of doubler() from Listing 7.21 that returns a new list.

```
>>> def newdoubler(xlist):
...     doublelist = []
...     for x in xlist:
...        doublelist.append(2 * x)
...     return doublelist
...
>>> mylist = [1, 2, 3]
>>> newdoubler(mylist)
[2, 4, 6]
>>> doubledlist = newdoubler(mylist)
>>> doubledlist
[2, 4, 6]
>>> mylist
[1, 2, 3]
```

In line 1 we see that newdoubler() has a single formal parameter xlist. In line 2, the first statement of the body of the function, doublelist is assigned to the empty list. doublelist serves as an accumulator into which we append the doubled values from xlist. This is accomplished with the for-loop in lines 3 and 4. Finally, after the loop is complete, in line 5 doublelist is returned.

Outside the function, in line 7, mylist is set equal to a list of three integers. In line 8 mylist is passed to newdoubler() and we see the returned list in line 9. The values of this list are double those of mylist. newdoubler() is called again in line 10 and the return value stored as doubledlist. Lines 11 through 14 demonstrates that doubledlist contains the doubled values from mylist.

Now that we understand newdoubler() consider the code shown in Listing 7.32 where we now use list comprehension to obtain the doubled list. The code is discussed following the listing.

Listing 7.32 Demonstration of list comprehension in which a new list is produced where the value of its elements are twice that of an existing list.

```
>>> mylist = [1, 2, 3]
>>> # Use list comprehension to obtain a new list where the values are
>>> # twice that of mylist.
>>> [2 * x for x in mylist]
[2, 4, 6]
>>> def lcdoubler(xlist):
```

```
7  ...     return [2 * x for x in xlist]
8  ...
9  >>> doubledlist = lcdoubler(mylist)
10 >>> doubledlist
11 [2, 4, 6]
12 >>> mylist
13 [1, 2, 3]
```

In line 1 we again assign mylist to a list of three integers. Line 4 provides a list comprehension expression that produces a new list where the values are double those of mylist. (The details of this expression will be considered further below.) Given that we can create a new list this way, in lines 6 and 7 we define the function lcdoubler() that has a single argument xlist. The body of the function is merely a return statement that returns the list produced by the list comprehension expression. Lines 9 through 13 demonstrate that lcdoubler() behaves in the same way as newdoubler() (ref. Listing 7.31).

Both lcdoubler() and newdoubler() accomplish the same task, doubling the elements in a list and returning a new list, but using list comprehension lcdoubler() accomplishes in two lines what newdoubler() did in five! list comprehensions can make code smaller, cleaner, and more readable. Of course, until you become comfortable with the syntax of list comprehensions, you may not find them "more readable" than the for-loop constructs we have previously discussed. But, you don't need to work with list comprehensions for long before you realize they are really just a slight syntactic variation on using accumulators, for-loops, and lists to construct a new list from the elements of an existing list. Now, however, is an appropriate time to point out we are not restricted to constructing the new list from an existing list. As will be shown, the new list can be constructed from *any* iterable (such as a tuple or a string).

Let's take another look at the list comprehension in line 4 of Listing 7.32 which is repeated below:

```
[2 * x for x in mylist]
```

Given the output on line 5, we know that this list comprehension produces a new list with values double that of mylist. Now let's consider the various components of this expression and see how it works.

The syntax for a list comprehension consists of brackets containing an expression followed by a for clause. Although this clause can be followed by any number of additional for and if clauses,[11] we will only consider the simplest form of list comprehension with a single for clause. Recall the pattern we are trying to implement: the creation of a new list based on the values in an existing list (or the values from some other iterable). Assuming this new list has the "placeholder" name of *<accumulator>*, the following is a template for implementing this pattern using a for-loop:

```
<accumulator> = []
for <item> in <iterable>:
```

[11]The initial for clause can actually be preceeded by an if clause to perform a form of filtering. if statements are considered in Chap. 11, but not in the context of list comprehensions.

```
    <accumulator>.append(<expression>)
```

In contrast to this, the following is the template for accomplishing the same thing using list comprehension where everything to the right of the assignment operator (equal sign) is the actual list comprehension:

```
<accumulator> = [<expression> for <item> in <iterable>]
```

Of course, with list comprehension, we aren't obligated to assign the list to an accumulator. We could, for example, have the list comprehension entered directly at the interactive prompt (such as in line 4 of Listing 7.32), or be part of a return statement (such as in line 7 of Listing 7.32), or appear directly as the argument of a print() statement.

In the list comprehensions in Listing 7.32 we didn't use explicit indexing, but this is certainly an option just as it is within for-loops. This is illustrated in Listing 7.33. In line 2 a list comprehension is again used to double the values of list, but here explicit indexing is used. As we saw from the previous examples, we do not need to use indexing to accomplish this doubling. However, in lines 4 and 5 we define a function called scaler() that "scales" each element of a list based on its index (elements are scaled by the sum of their index plus one). Lines 7 through 10 demonstrate that this function returns the proper result whether passed a list of numbers or strings.

Listing 7.33 Demonstration of the use of explicit indexing within list comprehensions.

```
1  >>> mylist = [1, 2, 3]
2  >>> [2 * mylist[i] for i in range(len(mylist))]
3  [2, 4, 6]
4  >>> def scaler(xlist):
5  ...     return [(i + 1) * xlist[i] for i in range(len(xlist))]
6  ...
7  >>> scaler(mylist)
8  [1, 4, 9]
9  >>> scaler(['a', 'b', 'c', 'd'])
10 ['a', 'bb', 'ccc', 'dddd']
```

7.8 Chapter Summary

One for-loop can be nested inside another.

One list can be nested inside another. Elements of the outer list are specified by one index enclosed in brackets. An element of an interior list is specified by two indices enclosed in two pairs of brackets. The first index specifies the outer element and the second element specifies the inner element. For example, if x corresponds to the list [[1, 2], ['a', 'b', 'c']], then x[1] corresponds to ['a', 'b', 'c'] and x[1][2] corresponds to 'c'.

Nesting can be done to any level. Specification of an element at each level requires a separate

index.

Simultaneous assignment can be used to assign values to multiple loop variables in the header of a `for`-loop.

`lists` are mutable, i.e., the elements of a `list` can be changed without creating a new `list`.

When a `list` is assigned to a variable, the variable becomes a reference (or alias) to the `list`. If one `list` variable is assigned to another, both variables point to the same underlying `list`. When a `list` is changed, it is changed globally.

Strings can be used as the iterable in a `for`-loop header in which case the loop variable is assigned the individual characters of the string.

Negative indexing can be used to specify an element of a sequence (e.g., a string, `list`, or `tuple`). The last element of a sequence has an index of `-1`. The first element of a sequence `s` has an index of `-len(s)`.

Slicing is used to obtain a portion of a sequence. For a sequence `s` a slice is specified by

```
s[<start> : <end>]
```

or

```
s[<start> : <end> : <inc>]
```

The resulting sequence takes elements from `s` starting at an index of `<start>` and going up to, but not including, `<end>`. The increment between successive elements is `<inc>` which defaults to `1`. The start and end values can be specified with either positive or negative indexing. `<start>` defaults to `0`. `<end>` defaults to the length of the sequence. `inc` may be negative. When it is negative, the default `<start>` is `-1` and the default `<end>` is `-len(s) - 1`. Reversal of a sequence can be achieved using `[ : : -1]`, and a `list`, e.g., `xlist`, can be copied using `xlist[ : ]`.

7.9 Review Questions

1. What output isproduced by the following code?

```
xlist = [1, [1, 2], [1, 2, 3]]
print(xlist[1])
```

2. What output is produced by the following code?

```
xlist = [1, [1, 2], [1, 2, 3]]
print(xlist[1][1])
```

3. What output is produced by the following code?

```
xlist = [1, [1, 2], [1, 2, 3]]
print(xlist[1] + [1])
```

4. What output is produced by the following code?

```
def sum_part(xlist, n):
    sum = 0
    for x in xlist[n]:
```

```
        sum = sum + x
    return sum

ylist = [[1, 2], [3, 4], [5, 6], [7, 8]]
x = sum_part(ylist, 2)
print(x)
```

5. Assume `xlist` is a `list` of `lists` where the inner `lists` have two elements. The second element of these inner `lists` is a numeric value. Which of the following will sum the values of the second element of the nested `lists` and store the result in `sum`?

   (a)
   ```
   sum = 0
   for item in xlist:
       sum = sum + item[1]
   ```

   (b)
   ```
   sum = 0
   for one, two in xlist:
       sum = sum + two
   ```

   (c)
   ```
   sum = 0
   for i in range(len(xlist)):
       sum = sum + xlist[i][1]
   ```

   (d) All of the above.

6. What output is produced by the following code?

```
for i in range(3):
    for j in range(3):
       print(i * j, end="")
```

   (a) `123246369`

   (b) `0000012302460369`

   (c) `000012024`

   (d) None of the above.

7. What output is produced by the following code?

```
s = "abc"
for i in range(1, len(s) + 1):
    sub = ""
    for j in range(i):
       sub = s[j] + sub
    print(sub)
```

(a)
```
a
ba
cba
```

(b)
```
a
ab
abc
```

(c)
```
a
ab
```

(d) This code produces an error.

8. What output is produced by the following code?

```
s = "grasshopper"
for i in range(1, len(s), 2):
    print(s[i], end="")
```

(a) `gasopr`

(b) `gr`

(c) `rshpe`

(d) `rshper`

9. What output is produced by the following code?

```
x = [7]
y = x
x[0] = x[0] + 3
y[0] = y[0] - 5
print(x, y)
```

10. What output is produced by the following code?

```
x = [7]
y = x
x = [8]
print(x, y)
```

11. What output is produced by the following code?

```
x = [1, 2, 3, 4]
y = x
y[2] = 0
z = x[1 : ]
x[1] = 9
print(x, y, z)
```

12. What output is produced by the following code?

```
s = "row"
for i in range(len(s)):
    print(s[ : i])
```

(a)
```

r
ro
```

(b)
```
r
ro
row
```

(c)
```
ro
row
```

(d) None of the above.

13. What output is produced by the following code?

```
s = "stab"
for i in range(len(s)):
    print(s[i : 0 : -1])
```

(a)
```
s
ts
ats
bats
```

(b)
```

t
at
bat
```

(c)
```

s
st
sta
```

(d) None of the above.

14. What output is produced by the following code?

```
s = "stab"
for i in range(len(s)):
    print(s[i : -5 : -1])
```

(a)
```
s
ts
ats
bats
```

(b)
```

t
at
bat
```

(c)
```

s
st
sta
```

(d) None of the above.

15. What output is produced by the following code?

```
s = "stab"
for i in range(len(s)):
    print(s[0 : i : 1])
```

(a)
```
s
ts
ats
bats
```

(b)
```

t
at
bat
```

(c)
```

s
st
sta
```

(d) None of the above.

ANSWERS: 1) `[1, 2]`; 2) `2`; 3) `[1, 2, 1]`; 4) `11`; 5) d; 6) c; 7) a; 8) c; 9) `[5] [5]`; 10) `[8] [7]`; 11) `[1, 9, 0, 4] [1, 9, 0, 4] [2, 0, 4]`; 12) a; 13) b; 14) a; 15) c.

Chapter 8

Modules and `import` Statements

The speed and accuracy with which computers perform computations are obviously important for solving problems or implementing tasks. However, quick and accurate computations, by themselves, cannot account for the myriad ways in which computers have revolutionized the way we live. Assume a machine is invented that can calculate the square root of an arbitrary number to any desired precision and can perform this calculation in a single attosecond (10^{-18} seconds). This is far beyond the ability of any computer currently in existence. But, assume this machine can only calculate square roots. As remarkable as this machine might be, it will almost certainly not revolutionize human existence.

Coupled with their speed and accuracy, it is flexibility that makes computers such remarkable devices. Given sufficient time and resources, computers are able to solve any problem that can be described by an algorithm. As discussed in Chap. 1, an algorithm must be translated into instructions the computer understands. In this book we write algorithms in the form of Python programs. So far, the programs we have written use built-in operators, built-in functions (or methods), as well as functions that we create.

At this point it is appropriate to ask: What else is built into Python? Perhaps an engineer wants to use Python and needs to perform calculations involving trigonometric functions such as sine, cosine, or tangent. Since inexpensive calculators can calculate these functions, you might expect a modern computer language to be able to calculate them as well. We'll return to this point in a moment. Let's first consider a programmer working in the information technology (IT) group of a company who wants to use Python to process Web documents. For this programmer the ability to use sophisticated string-matching tools that employ what are known as *regular expressions* might be vitally important. Are functions for using regular expressions built into Python? Note that an engineer may never need to use regular expressions while a typical IT worker may never need to work with trigonometric functions. When you consider all the different ways in which programmers want to use a computer, you quickly realize that it is a losing battle to try to provide all the built-in features needed to satisfy everybody.

Rather than trying to anticipate the needs of all programmers, at its core Python attempts to remain fairly simple. We can, in fact, obtain a list of all the "built-ins" in Python. If you issue the command `dir()` with no arguments, you obtain a `list` of the methods and attributes currently defined. One of the items in this `list` is `__builtins__`. If we issue the command

From the file: `import.tex`

dir(__builtins__) we obtain a list of everything built into Python. At the start of this list are all the exceptions (or errors) that can occur in Python (exceptions start with an uppercase letter). The exceptions are followed by seven items, which we won't discuss here, that start with an underscore. The remainder of this list provides all the built-in functions. Listing 8.1 demonstrates how this list can be obtained (for brevity, the exceptions and items starting with an underscore have been removed from the list that starts on line 8).

Listing 8.1 In the following, the integer variable z and the function f() are defined. When the dir() function is called with no arguments, we see these together with various items Python provides. Calling dir() with an argument of __builtins__ provides a list of all the functions built into Python.

```
1  >>> z = 111
2  >>> def f(x):
3  ...     return 2 * x
4  ...
5  >>> dir()
6  ['__builtins__', '__doc__', '__name__', '__package__', 'f', 'z']
7  >>> dir(__builtins__)
8  [ <<START OF LIST DELETED>>
9    'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
10   'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
11   'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
12   'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
13   'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
14   'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
15   'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object',
16   'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
17   'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
18   'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
19   'zip']
```

In lines 1 through 3 of Listing 8.1, the integer variable z and the function f() are defined. In line 5 the dir() function is called with no arguments. The resulting list, shown in line 6, contains these two items together with items that Python provides. The first item, __builtins__, is the one that is of current interest. Using this as the argument of dir() yields the list of functions built into Python (items in the list that are not functions have been deleted). The 72 built-in functions are given in lines 9 through 19.[1]

Several of the functions listed in Listing 8.1 have already been discussed, such as dir(), divmod(), eval(), and float(). There are some functions that we haven't considered yet but we can probably guess what they do. For example, we might guess that exit() causes Python to exit and that copyright() gives information about Python's copyright. We might even guess that abs() calculates the absolute value of its argument. All these guesses are, in fact, correct.

[1]In fact, not all of these are truly functions or methods. For example, int() is really a constructor for the class of integers, but for the user this distinction is really unimportant and we will continue to identify such objects as functions.

Of course, there are many other functions whose purpose we would have trouble guessing. We can use `help()` to provide information about these, but such information isn't really of interest now. What is important is that there are, when you consider it, a surprisingly small number of built-in functions. Note that there is nothing in this `list` that looks like a trigonometric function and, indeed, there are no trig functions built into Python. Nor are there any functions for working with regular expressions.

Why in the world would you want to provide a `copyright()` function but not provide a function for calculating the cosine of a number? The answer is that Python distributions include an extensive *standard library*. This library provides a math module that contains a large number of mathematical functions. The library also provides a module for working with regular expressions as well as much, much more.[2] A programmer can extend the capabilities of Python by *importing* modules or packages using an `import` statement.[3] There are several variations on the use of `import`. For example, a programmer can import an entire module or just the desired components of a module. We will consider the various forms in this chapter and describe a couple of the modules in the standard library. To provide more of a "real world" context for some of these constructs, we will introduce Python's implementation of complex numbers. We will also consider how we can import our own modules.

8.1 Importing Entire Modules

The `math` module provides a large number of mathematical functions. To gain access to these functions, we can write the keyword `import` followed by the name of the module, e.g., `math`. We typically write this statement at the start of a program or file of Python code. By issuing this statement we create a `module` object named `math`. The "functions" we want to use are really methods of this object. Thus, to access these methods we have to use the "dot-notation" where we provide the object name, followed by a dot (i.e., the access operator), followed by the method name. This is illustrated in Listing 8.2. The code is discussed following the listing.

Listing 8.2 Use of an `import` statement to gain access to the methods and attributes of the `math` module.

```
1  >>> import math
2  >>> type(math)
3  <class 'module'>
4  >>> dir(math)
5  ['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
6    'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
7    'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
8    'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
9    'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
10   'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
```

[2] The complete documentation for the standard library can be found at **docs.python.org/py3k/library/**.

[3] We will not distinguish between modules and packages. Technically a module is a single Python `.py` file while a package is a directory containing multiple modules. To the programmer wishing to use the module or package, the distinction between the two is inconsequential.

```
11   'sqrt', 'tan', 'tanh', 'trunc']
12 >>> help(math.cos)       # Obtain help on math.cos.
13 Help on built-in function cos in module math:
14 
15 cos(...)
16     cos(x)
17 
18     Return the cosine of x (measured in radians).
19 
20 >>> math.cos(0)          # Cosine of 0.
21 1.0
22 >>> math.pi              # math module provides pi = 3.1414...
23 3.141592653589793
24 >>> math.cos(math.pi)    # Cosine of pi.
25 -1.0
26 >>> cos(0)               # cos() is not defined.  Must use math.cos().
27 Traceback (most recent call last):
28   File "<stdin>", line 1, in <module>
29 NameError: name 'cos' is not defined
```

In line 1 the `math` module is imported. This creates an object called `math`. In line 2 the `type()` function is used to check `math`'s type and we see, in line 3, it is a `module`. In line 4 `dir()` is used to obtain a `list` of the methods and attributes of `math`. In the `list` that appears in lines 5 through 11 we see names that look like trig functions, e.g., `cos`, `sin`, and `tan`. There are other functions whose purpose we can probably guess from the name. For example, `sqrt` calculates the square root of its argument while `log10` calculates the logarithm, base 10, of its argument. However, rather than guessing what these functions do, we can use the `help()` function to learn about them. This is demonstrated in line 12 where the `help()` function is used to obtain help on `cos()`. We see, in line 18, that it calculates the cosine of its argument, where the argument is assumed to be in radians.

In line 20 the `cos()` function is used to calculate the cosine of `0` which is `1.0`. Not only does the `math` module provide functions (or methods), it also provides attributes, i.e., data or numbers. For example, `math.pi` is a finite approximation of π (i.e., $3.14159\cdots$, the ratio of the circumference to the diameter of a circle) while `math.e` is an approximation of Euler's constant (i.e., $e = 2.71828\cdots$). The cosine of π is -1 and this is confirmed in line 24 of the code. The statement in line 26 shows that the function `cos()` is not defined. For the way we have imported the `math` module, we must use the proper dot notation to gain access to its methods/functions, i.e., we must write `math.cos()`.

Assuming the `math` module has been imported, Listing 8.3 shows a portion of the output obtained from the command `help(math)`. After importing a module one can usually obtain extensive information about the module in this way.

Listing 8.3 Information about the `math` module can be obtained via `help(math)`. The following shows a portion of this information.

```
>>> help(math)
Help on module math:

NAME
    math

MODULE REFERENCE
    http://docs.python.org/3.2/library/math

    The following documentation is automatically generated from the
    Python source files.  It may be incomplete, incorrect or include
    features that are considered implementation detail and may vary
    between Python implementations.  When in doubt, consult the module
    reference at the location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

         Return the hyperbolic arc cosine (measured in radians) of x.
<<REMAINING OUTPUT DELETED>>
```

8.2 Introduction to Complex Numbers

Another data type that Python provides is the `complex` type for complex numbers. Complex numbers are extremely important in a wide range of problems in math, science, and engineering. Complex numbers and the complex mathematical framework that surrounds them provide beautifully elegant solutions to problems that are cumbersome or even impossible to solve using only real numbers. Complex numbers can be thought of as points in a plane, which is commonly referred to as the *complex plane*, as depicted in Fig. 8.1.

Instead of specifying the points as an ordered pair of numbers as you might with points in a Cartesian plane (i.e., an "(x, y) pair"), we describe points in a complex plane as consisting of a real part and an imaginary part. The use of the word "imaginary" is rather unfortunate since imaginary can imply something fictitious or made up. But imaginary numbers are just as real as "real" numbers—they provide ways for us to describe and quantify the world. The real part of a complex number represents the projection of the complex number onto the "real axis" (i.e., the

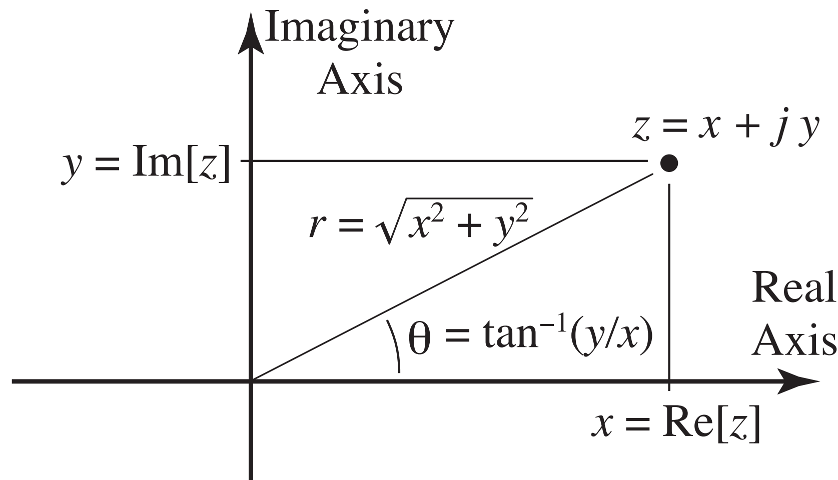


Figure 8.1: Representation of the complex number z in the *complex plane*. The number has a real part $\Re[z] = x$ (which gives the displacement along the horizontal, i.e., real, axis) and an imaginary part $\Im[z] = y$ (which gives the displacement along the vertical, i.e., imaginary, axis). The number can also be specified in polar form, $z = r\underline{/\theta}$ where the magnitude r is the displacement from the origin and the phase θ is the angle between the positive real axis and a ray from the origin to the point.

real number line). The *imaginary part* of a complex number is itself a real number! This number represents the projection of the complex number onto the "imaginary axis" which is orthogonal to the real axis. You can think of the real axis as corresponding to the x axis and the imaginary axis as corresponding to the y axis in a Cartesian plane. In fact, we often write a complex number z as $x + jy$ where x is the real part of z and y is the imaginary part of z. Again, y is a real number. However, j is defined as the "imaginary" number such that it is the square root of -1, i.e., $\sqrt{-1} = \pm j$ or, thought of another way, $(\pm j)^2 = -1$. In the complex plane, the number j is located along the imaginary axis a unit distance from the origin (a distance of 1 from the origin). When we write jy, we mean a displacement of y from the origin along the imaginary axis.

As with the usual representation of points in a plane, complex numbers can be also be expressed in polar form (ref. Fig. 8.1). We can write $z = r\underline{/\theta}$, where r is known as the magnitude (or modulus or absolute value) of z and is the displacement from the origin to the point z, and θ is known as the phase (or argument) and is the angle between the positive real axis and a ray from the origin to the point z. One can relate the Cartesian and polar forms of a complex number via $x = r\cos(\theta)$ and $y = r\sin(\theta)$. Though we won't explore it here, one of the remarkable and beautiful properties of complex numbers is described by Euler's formula which states $e^{j\theta} = \cos(\theta) + j\sin(\theta)$. Given this, we can write a complex number z either as $x + jy$ or $re^{j\theta}$.

To add two complex numbers, we simply add their real and imaginary parts (this is like vector addition for position vectors in a plane). So, in general, the sum of two complex numbers z_1 and z_2 is given by

$$z_1 + z_2 = (x_1 + jy_1) + (x_2 + jy_2) = (x_1 + x_2) + j(y_1 + y_2)$$

To multiply two complex numbers, we can follow the rules of multiplication that pertain to multi-

plying sums of real numbers. Thus, the product of z_1 and z_2 is given by:

$$z_1 \times z_2 = (x_1 + jy_1) \times (x_2 + jy_2) = x_1x_2 + j^2y_1y_2 + j(x_1y_2 + x_2y_1) = (x_1x_2 - y_1y_2) + j(x_1y_2 + x_2y_1)$$

A literal complex number is written in Python by writing `j` at the end of the numeric value representing the imaginary part. For example, in Python we write `1 + 1j` to obtain the complex number that is typically written in a math, engineering, or science course as $1 + j$. Note that if we write simply `j` or `j1`, these are treated as identifiers (i.e., treated as variables since these are valid identifiers). If these identifiers have not been previously defined, we obtain a syntactic error (an error in grammar). If they have been defined previously, we will obtain a semantic error (i.e., an error in meaning—Python will not produce an exception, but we will not obtain the complex number we wanted). In Python the complex number $5.6 - j7.3$ is written as `5.6 - 7.3j`. Writing `5.6 - j7.3` will result in an error since `j7.3` is neither a valid number nor a valid identifier.

The code in Listing 8.4 demonstrates the use of complex numbers. In line 1 the variable `z1` is assigned the complex value that is typically written as $1 + j$ in a math or engineering textbook. In line 15 the variable `z3` is assigned the value that is typically written as $5.6 - j7.3$. The discussion continues following the listing.

Listing 8.4 Demonstration of the use of complex numbers.

```
1  >>> z1 = 1 + 1j       # Create complex number z1.
2  >>> type(z1)          # Check z1's type.
3  <class 'complex'>
4  >>> z1                # Check z1.
5  (1+1j)
6  >>> j = 10            # Set j to 10.
7  >>> z2 = 1 + j        # Create integer z2 -- not complex!
8  >>> z2                # Check z2.
9  11
10 >>> z3 = 5.6 - j7.3   # Attempt to create complex number z3.
11   File "<stdin>", line 1
12     z3 = 5.6 - j7.3
13                  ^
14 SyntaxError: invalid syntax
15 >>> z3 = 5.6 - 7.3j   # Correct way to create complex number z3.
16 >>> z1 + z3           # Sum of complex numbers.
17 (6.6-6.3j)
18 >>> z1 * z3           # Product of complex numbers.
19 (12.899999999999999-1.7000000000000002j)
```

In line 2 the `type()` function is used to show that `z1` is a `complex` object. In line 6 the variable `j` is assigned the integer value `10`. The statement in line 7 looks deceptively like the statement in line 1. However, while the statement in line 1 creates the complex number with a real and imaginary part of `1`, the statement in line 7 adds `1` to the current value of the variable `j`. Thus, `z2` is assigned the value `11` as shown in line 9.

The statement in line 10 is a failed attempt to assign the complex value $5.6 - j7.3$ to the variable `z3`. Again, the `j` used to indicate an imaginary number must be the trailing part of the number. Line 15 shows the correct way to make this assignment.

Line 16 produces the sum of `z1` and `z3` while line 18 yields their product. We can mix complex numbers with `floats` and integers in most arithmetic operations. (There are just a few exceptions to this. For example, we cannot use complex numbers with floor division or modulo operators.)

Listing 8.5 illustrates additional aspects of complex numbers. In line 1 the value $3 + j4$ is assigned to the identifier `z`. Then, in line 2, the `dir()` function is used to show the methods and attributes of this `complex` value. We are only concerned with the attributes `real` and `imag` that appear at the end of the listing. Note that they are attributes and not methods. Thus, to obtain the real and imaginary parts of `z` we write (without parentheses) `z.real` and `z.imag`, respectively. This is demonstrated in lines 13 through 16. Both the real and imaginary parts of a `complex` value are stored as `floats` as indicated in lines 17 and 18. The discussion continues following the listing.

Listing 8.5 The methods and attributes associated with a complex number.

```
1  >>> z = 3 + 4j
2  >>> dir(z)
3  ['__abs__', '__add__', '__bool__', '__class__', '__delattr__',
4   '__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__',
5   '__format__', '__ge__', '__getattribute__', '__getnewargs__',
6   '__gt__', '__hash__', '__init__', '__int__', '__le__', '__lt__',
7   '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__',
8   '__pow__', '__radd__', '__rdivmod__', '__reduce__',
9   '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__',
10  '__rmul__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__',
11  '__sizeof__', '__str__', '__sub__', '__subclasshook__',
12  '__truediv__', 'conjugate', 'imag', 'real']
13 >>> z.real     # The real part of z.
14 3.0
15 >>> z.imag     # The real imaginary part of z.
16 4.0
17 >>> type(z.real)
18 <class 'float'>
19 >>> (z.real ** 2 + z.imag ** 2) ** 0.5   # Magnitude of z.
20 5.0
21 >>> abs(z)     # Simpler way to obtain magnitude.
22 5.0
```

As mentioned above, the magnitude of z is given by $r = \sqrt{x^2 + y^2}$. Line 19 shows one way to calculate this value. However, the built-in function `abs()` can also be used to obtain the magnitude of a complex number as demonstrated in lines 21 and 22.

8.3 Complex Numbers and the cmath Module

Given this background on complex numbers, we now can ask: How do we calculate the square root or cosine of a complex number? These are, in fact, well defined operations mathematically, but can Python do them?

Listing 8.6 illustrates what happens when we try to use complex numbers with the functions from the math module. In line 1 the math module is imported. In line 2 the complex number z1 is created with a real part of zero and an imaginary part of 1.0. (Note that if no real part is given, such as the case here, then it is zero. Also, both the real and imaginary parts of a complex number are stored as floats.) Line 3 shows that when we square z1, we obtain the complex number (-1+j0). Another thing to note is that complex numbers are not converted back to floats when their imaginary part is zero nor are they converted to integers even when they correspond to whole numbers. The discussion continues following the listing.

Listing 8.6 Attempt to use complex numbers with functions from the math module.

```
1  >>> import math
2  >>> z1 = 1j      # z1 has zero real part and imaginary part of 1.
3  >>> z1 * z1      # z1 squared is negative 1.
4  (-1+0j)
5  >>> # Cannot use math.sqrt() on a complex number.
6  >>> math.sqrt(z1)
7  Traceback (most recent call last):
8    File "<stdin>", line 1, in <module>
9  TypeError: can't convert complex to float
10 >>> # Cannot use math.sqrt() on a complex number.
11 >>> math.sqrt(-1)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 ValueError: math domain error
15 >>> # Cannot use math.cos() on a complex number.
16 >>> math.cos(5.6 - 7.3j)
17 Traceback (most recent call last):
18   File "<stdin>", line 1, in <module>
19 TypeError: can't convert complex to float
```

In lines 6 through 9 we see that we cannot use math.sqrt() with a complex number. In fact, lines 11 through 14 show that we cannot even use math.sqrt() with a negative real number. Lines 16 through 19 show that math.cos() cannot be used with complex numbers.

So, does that mean that one cannot calculate things such as $\sqrt{j}$ or $\cos(5.6 - j7.3)$ in Python? Not at all. Rather this serves as a reminder that Python tries to compartmentalize the functionality it provides. There are many programmers who may need to work with real numbers and real functions and yet will never need to work with complex numbers. In the interest of speed and program size, it is best if these programmers do not have to work with functions that are designed to deal with the “complexity” of complex numbers.

If a programmer needs to have complex functions at his or her disposal, they should import the `cmath` module. This is demonstrated in Listing 8.7.

Listing 8.7 Demonstration of the `cmath` module which provides support for complex numbers.

```
1  >>> import cmath
2  >>> dir(cmath)
3  ['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
4    'asin', 'asinh', 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp',
5    'isfinite', 'isinf', 'isnan', 'log', 'log10', 'phase', 'pi',
6    'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
7  >>> z1 = 1j
8  >>> cmath.sqrt(z1)
9  (0.7071067811865476+0.7071067811865475j)
10 >>> cmath.cos(5.6 - 7.6j)
11 (774.8664714775274-630.6970442971782j)
12 >>> z2 = 3 + 4j
13 >>> cmath.polar(z2)
14 (5.0, 0.9272952180016122)
```

The `cmath` module is imported in line 1. The `dir()` function is used to show the methods and attributes in this module. If you compare lines 3 through 6 to lines 5 through 11 in Listing 8.2, you notice that the names of the functions in the `cmath` module largely duplicate those in the `math` module. There are just a few functions with new names in the `cmath` module.[4] But, despite the common names, the functions in these two modules are different, e.g., `cmath.sqrt()` is different from `math.sqrt()`. This is evident from the fact that we obtain the square root of `z1` in line 8 of Listing 8.7 but a similar statement produced an error in line 6 of Listing 8.6. Line 10 shows that we can calculate the cosine of a complex number if we use `cmath.cos()`.

In line 12 the complex value $3 + j4$ is assigned to the identifier `z2`. Then, in line 13, this value is given as the argument of the `polar()` function which returns, as a tuple, the magnitude and phase of the complex number. We see that the first element of this tuple agrees with the magnitude that was calculated in lines 19 and 21 of Listing 8.5. The second value of the tuple is the phase and is given in radians. Note that 0.927295 radians corresponds to approximately 53.13 degrees, i.e., $\tan^{-1}(4/3)$ expressed in degrees.

If one needs to import multiple modules, one either writes multiple `import` statements or writes a single `import` statement with the modules separated by commas. Thus,

```
import math, cmath
```

is equivalent to this

```
import math
import cmath
```

[4]The new functions are `phase()`, `polar()`, and `rect()`.

Again, despite the fact that there are methods within these modules that share a common name, these methods are distinct because the only way to access them is via the dot notation with the module objects (and the module objects do have distinct names).

There is one variation of the `import` statement that is appropriate to mention here. We are not required to use the module's name as the identifier we use within our code. We can import a module **as** some other identifier. For example, the statement

```
import math as realmath
```

will import the `math` module as `realmath` so that we must write `realmath.sqrt()` to access the square root function within this module. We can import more than one module with a single statement and use alternate identifiers for each. As an example, the code in Listing 8.8 imports the `math` module as `realmath` and the `cmath` module as `complexmath`.

Listing 8.8 Example of importing multiple modules with a single statement and using alternate identifiers for the modules.

```
1 >>> import math as realmath, cmath as complexmath
2 >>> realmath.sqrt(2)      # Real function, integer argument.
3 1.4142135623730951
4 >>> complexmath.sqrt(2)   # Complex function, integer argument
5 (1.4142135623730951+0j)
```

8.4 Importing Selected Parts of a Module

Often there is no reason to import all the methods and attributes contained in a module. Perhaps an engineer needs to work with the cosine function and the number π but does not need anything more than these from the `math` module. There are alternate forms of the `import` statement by which one can specify the individual objects to be imported. Listing 8.9 provides templates for these alternatives. Each statement starts with the keyword `from`. This is followed by the module name and then the keyword `import`. The remainder of the statement specifies what is to be imported and, if an `as` qualifier is present, what the identifier should be for the specified object (`as` is also a keyword). If more than one object is to be imported, the objects are separated by commas. When importing multiple objects, the `as` qualifier may be added or omitted as appropriate.

Listing 8.9 Using the `from-import` construct to select individual components of a module.

```
1 # Import a single object.
2 from <module> import <object>
3 # Import a single object and assign to an alternative name.
4 from <module> import <object> as <ident>
5 # Import multiple objects.
6 from <module> import <object1>, <object2>, ..., <objectN>
7 # Import multiple objects and assign to alternative names.
```

```
8 # The as qualifier may be omitted as appropriate.
9 from <module> import <object1> as <ident1>, ..., <objectN> as <identN>
```

To demonstrate importation of only selected objects from a module, consider the code shown in Listing 8.10 where the `sqrt()` function and the number `pi` are imported from the `math` module. The statement in line 1 accomplishes the desired importing. Since the `as` qualifier was not used, the objects are assigned to the identifiers they have within the module. The statements in lines 2 and 4 show that these objects were successfully imported. Line 6 shows that the `help()` function can still be used to obtain help on a function that has been imported. However, since the entire `math` module itself has not been imported, we cannot obtain help on the entire module. Thus, the statement in line 13 produces an error.

Listing 8.10 Demonstration of an `import` statement that employs `from` to select individual objects for importation.

```
1  >>> from math import sqrt, pi
2  >>> sqrt(2)
3  1.4142135623730951
4  >>> print(pi)
5  3.141592653589793
6  >>> help(sqrt)
7  Help on built-in function sqrt in module math:
8
9  sqrt(...)
10     sqrt(x)
11
12     Return the square root of x.
13 >>> help(math)
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 NameError: name 'math' is not defined
```

Now assume that we want to import the square root functions from both the `math` module and the `cmath` module. An attempt to do this is shown in Listing 8.11. In line 1 the `sqrt()` function is imported from the `cmath` module. In line 2 the `sqrt()` function is imported from the `math` module. Since these functions have the same name, the second `import` statement in line 2 associates the identifier `sqrt` with the `sqrt()` function from the `math` module—the `sqrt()` function from the `cmath` module is lost to us. Thus, in line 3, we produce an error when we attempt to take the square root of negative one (i.e., mathematically the result should be the complex number j, but the `sqrt()` function from the `math` module is not capable of producing this result).

Listing 8.11 Attempt to import the `sqrt()` function from both the `math` and `cmath` modules.

```
>>> from cmath import sqrt
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Listing 8.12 demonstrates how, using the `as` qualifier, we can import both `sqrt()` functions and keep them separate. In line 1 we import `sqrt()` from the `math` module and assign it to the identifier `rsqrt`. Also in line 1 we import the number `pi`. Since we do not provide an `as` qualifier, this is imported simply as `pi`. In line 2 the `sqrt()` function from the `cmath` module is imported and assigned to the identifier `csqrt`. Additionally, `pi` is imported and assigned to `cpi`. The statements in lines 3 and 5 show that these two different square-root functions are now available for use. The statement in line 7 is used to show that the value of `pi` in both `math` and `cmath` modules is indeed identical.

Listing 8.12 Demonstration of `import` statements that employ `from` to select individual objects for importation.

```
>>> from math import sqrt as rsqrt, pi
>>> from cmath import sqrt as csqrt, pi as cpi
>>> rsqrt(2)
1.4142135623730951
>>> csqrt(2+2j)
(1.5537739740300374+0.6435942529055826j)
>>> print(pi, cpi, pi - cpi)
3.141592653589793 3.141592653589793 0.0
```

8.5 Importing an Entire Module Using *

Assume a programmer needs to work with most, or even all, of the objects contained in a module. In this case it can be tiresome to have to type `module.object` whenever accessing one of the objects. As you saw in the previous section, one can use the `from` variant of an `import` statement to import objects so that they are available directly in the local scope, i.e., we don't need to use the `module`-dot notation (in fact, we *cannot* use it). So, one possibility is to use a `from-import` statement where one lists all the objects in a module. However, this is rather cumbersome. Fortunately there is a better way.

Python allows you to import everything from a module if, in a `from-import` statement, you simply write an asterisk for the object you want to import. The asterisk serves as a "wildcard," meaning everything. The code in Listing 8.13 demonstrates how this is done.

Listing 8.13 Demonstration of directly importing all the contents of a module into the local scope.

```
>>> from math import *  # Import everything from the math module.
>>> pi
3.141592653589793
>>> sqrt(2)
1.4142135623730951
>>> # Cosine of pi / 2 is zero.  But, because of finite precision, the
>>> # result isn't exactly zero (but it's close!).
>>> cos(pi / 2)
6.123233995736766e-17
```

In line 1 the entire contents of the `math` module are imported. Line 2 shows `pi` is available. Line 4 shows the `sqrt()` function is available. Finally, line 8 shows the `cos` function is available. As something of an aside and related to the statement in line 8, cosine of $\pi/2$ is identically zero. But, because `pi` is a finite approximation to π, the result shown in line 9 differs slightly from zero.

There are some things to note when using the asterisk form of importing. Although it is certainly convenient, it has some drawbacks. Because this imports everything from a module, there is a chance that identifiers you use in your code will conflict with the identifiers used within the module. This can cause problems. Also, by importing in this way you will not have access to `help()` on the module as a whole (although help will be available for the individual functions that have been imported).

8.6 Importing Your Own Module

A module is simply a `.py` file. Thus, we can easily create our own modules that contain a collection of functions or other objects. Let us consider an example. Assume the code shown in Listing 8.14 has been placed in the file `my_mod.py`. The file starts with a multi-line string that gives a brief description of the module. In line 6 the variable `scale` is assigned the value `2`. Since this assignment is made outside any function, `scale` is globally defined within this module—any function can use or change this variable. Also, as we will see, this variable is "visible" wherever this module is imported. The module also contains two function definitions. The first function, `scaler()`, returns `scale` times its argument while the second function, `reverser()` returns the reverse of its argument (thus, of course, there are restrictions on what arguments can be passed to these functions). Note that the body of each function contains a docstring (docstrings were discussed in Sec. 4.3).

Listing 8.14 Code used to demonstrate importing a module of our own.

```
"""
This module contains two functions and one globally defined
integer.
"""

scale = 2

```

```
8  def scaler(arg):
9      """Return the scaled value of the argument."""
10     return scale * arg
11
12 def reverser(xlist):
13     """Return the reverse of the argument."""
14     # Reverse using a slice with negative increment and default start
15     # and stop.
16     return xlist[ : : -1]
```

Now let's consider, as shown in Listing 8.15, what happens when we import this module and access the objects it contains. In line 1 we import the module. Importantly, note that the file itself is `my_mod.py` but *we do not include the `.py` in the `import` statement*. Lines 2 and 3 show us that `my_mod.scale` is set to `2`. In line 4 the `my_mod.scaler()` function is called with an argument of `42.0`. Since this function returns the product of its argument and `my_mod.scale`, we obtain `84.0` as shown in line 5. `my_mod.scaler()` can be called with a string argument as shown in line 6. Lines 8 and 9 demonstrate that `my_mod.reverser()` works properly. The discussion continues following the listing.

Listing 8.15 Importation and use of the module shown in Listing 8.14.

```
1  >>> import my_mod
2  >>> my_mod.scale
3  2
4  >>> my_mod.scaler(42.0)
5  84.0
6  >>> my_mod.scaler("liar")
7  'liarliar'
8  >>> my_mod.reverser("!pleH")
9  'Help!'
10 >>> my_mod.scale = 3     # Change value of my_mod.scale.
11 >>> my_mod.scaler("La")  # See that change affects my_mod.scaler().
12 'LaLaLa'
13 >>> help(my_mod)         # See what help is available for my_mod.
14 Help on module my_mod:
15
16 NAME
17     my_mod
18
19 DESCRIPTION
20     This modules contains two functions and one globally defined
21     integer.
22
23 FUNCTIONS
24     reverser(xlist)
25         Return the reverse of the argument.
```

```
26
27      scaler(arg)
28          Return the scaled value of the argument.
29
30  DATA
31      scale = 3
32
33  FILE
34      /Users/schneidj/Documents/my_mod.py
```

In line 10 the value of `my_mod.scale` is changed to `3`. The call of `my_mod.scaler()` in line 11 shows that the output is affected by this change, i.e., the argument is now repeated three times instead of two. You should be aware, however, that this change did *not* change the value stored in the file `my_mod.py`—it remains `2` so that the next time the module is imported, `my_mod.scale` is `2`.[5]

The remaining lines of Listing 8.15 show the output obtained when we ask `help()` to tell us about `my_mod`. The docstrings that were provided in the bodies of the functions as well as the string at the start of the file are used to describe the module. We also see, in line 31, that `scale` is part of the "`DATA`" in the module. Rather interestingly, we are told the current value of `scale` rather than the value it had when the module was first imported.

When working with our own modules, there are some additional things we must note. First, in the interest of speed, Python will import a module only once. So, for example, if you import a module into the interactive environment, notice a bug in the module, use an editor to correct the bug, and then issue another `import` statement in the same interactive session, Python does *not* re-import the module. There is a way to force Python to reload a module, but this can cause some subtle problems and thus we will not describe how to do this.[6] In situations like this, it is best simply to close the Python interpreter and start again. If you are using IDLE, you can simply select "Restart Shell" from the Shell menu or, if you are running your code from a file, IDLE will automatically restart the interpreter for you when you select "Run Module."

The second thing to be aware of is that Python needs to find your module in order to import it. Perhaps the simplest way to ensure a module is found is to put the module in one of the following places (which are appropriate for Python 3.2.x). On a Windows machine the module can be placed in

```
C:\Python32\Lib\site-packages
```

On a Macintosh, the module can be placed in

```
/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-packages
```

On a Debian/Ubuntu Linux machine, the module can be placed in

```
/usr/lib/python3.2/dist-packages
```

[5]However, after a module has been imported once during a session, it takes some additional commands in order to be able to re-import it.

[6]But, for the truly curious, you can do this by first importing the `imp` module and then using the `reload()` method of this module. So, for example, to "reload" the `my_mod` module, you would issue these commands: `import imp; imp.reload(my_mod)`.

Alternatively, one can modify where Python searches for modules. The directories where Python searches for modules are collectively known as the *path*. One of the places Python always looks is what is known as the “current working directory” (which is often abbreviated cwd). To see what Python considers the current working directory, you can issue these commands

```
import os     # Import the "operating system" module os.
os.getcwd()   # Display the current working directory.
```

If the current working directory doesn’t match where the module is located, one can use the `os.chdir()` to change the current working directory. The argument to `os.chdir()` should be a string appropriate for the operating system. For example, on a Macintosh, the current working directory can be set to the `Documents` folder for user `guido` by issuing the following statement

```
os.chdir("/Users/guido/Documents")
```

On a Windows box, where the desired folder is `My Documents`, the command might be

```
os.chdir("C:/Users/guido/My Documents")
```

Note the use of single forward slashes in this command! Typically one uses backslashes in Windows, but backslashes are used to indicate escape sequences in strings in Python. So, if you prefer to use backslashes, you have to use two of them, i.e.,

```
os.chdir("C:\\Users\\guido\\My Documents")
```

Rather than changing the working directory, one can modify the path over which Python searches for modules. Thus, on a Macintosh, to ensure Python searches in the `Documents` directory for the user `guido`, the appropriate commands are

```
import sys
sys.path.append("/Users/guido/Documents")
```

On a Windows machine where we want to search in the `My Documents` folder, the commands are

```
import sys
sys.path.append("C:/Users/guido/My Documents")
```

Again, backslashes can be used instead of forward slashes, but they have to be repeated twice.

8.7 Chapter Summary

Modules can by imported to enhance the capabilities of Python. To accomplish this, one uses an **`import`** statement.

The contents of an entire module can be imported using any of the following statements:

1. `import <module>`
2. `import <module> as <id>`
3. `from <module> import *`

In the second statement `<id>` serves as an alias for `<module>`. For the first and second statements, dot-notation is required to access an imported object, e.g., `<module>.<object>`. For the third statement, imported objects are di-

rectly visible, e.g., an imported function can be called simply using `<function>()`.

Multiple modules can be specified using the first statement with module names separated by commas. Multiple modules and their corresponding identifiers can be specified using the second statement with pairs of modules and identifiers separated by commas, e.g.,

```
import <mod1> as <id1>,\
       <mod2> as <id2>
```

Multiple modules cannot be specified using the third statement. Instead, multiple (separate) statements must be used.

A portion of a module can be imported using either of the following:

1. `from <module> import <object>`
2. `from <module> import <object> as <id>`

Multiple objects can be specified using the first statement with objects separated by commas. Multiple objects and their corresponding identifiers can be specified using the second statement with the pairs separated by commas, e.g.,

```
from <module> import \
     <obj1> as <id1>,\
     <obj2> as <id2>
```

A module must be in Python's path to be imported.

8.8 Review Questions

1. Which statement would import all of the objects of a module called `mymodule` so that they could be used directly (without dot notation).

   (a) `import mymodule`

   (b) `import mymodule.*`

   (c) `import * from mymodule`

   (d) `from mymodule import all`

   (e) `from mymodule import *`

2. What statement allows the `math` module to be used in a program?

   (a) `input math`

   (b) `import * from math`

   (c) `import math`

   (d) Either (b) or (c).

ANSWERS: 1) e; 2) c.

Chapter 9

Strings

You are probably accustomed to referring to a collection of readable characters as "text." By "readable characters," we mean the symbols that one typically finds on a keyboard such as letters of the alphabet, digits zero through nine, and punctuation marks (including a blank space). As we will soon see, computers use a mapping between characters and numbers: each character has a corresponding unique numeric value. With this mapping in place, we can interpret (and display) the ones and zeros stored in the computer as a character or we can interpret (and display) them as a number. In either case, we are talking about the same combination of ones and zeros. The only thing that differs is whether we decide to interpret the combination as a number or, using the established mapping, as a character.

The first standardized mapping created for numbers and characters was the American Standard Code for Information Interchange (ASCII) which is still in use today. ASCII specifies the numeric values corresponding to 128 characters and we will focus our attention on these 128 characters.

If text is stored in a computer using only ASCII characters, we refer to this as "plain text." So, for example, a Word document is *not* a plain text file. A Word document contains information that goes far beyond the text you ultimately read in the rendered document. For example, a Word document contains information pertaining to the page layout, the fonts to be used, and the history of previous edits. On the other hand, a plain-text file specifies none of this. The Python files you create with IDLE (i.e., the `.py` files) are plain-text files. They contain the ASCII characters corresponding to the code and nothing more. Essentially every byte of information in the file is "visible" when you open the file in IDLE (this "visibility" may be in the form of space or a new line). The file contains no information about fonts or page size or anything other than the code itself.

Many cutting-edge problems involve the manipulation of plain text. For example, Web pages typically consist of plain text (i.e., hyper-text markup language [HTML]) commands. The headers of email messages are in plain text. Data files often are written in the form of plain text. XML (extensible markup language) files are written in plain text. Thus, the ability to manipulate plain text is a surprisingly important skill when solving problems in numerous disciplines.

As you have seen, in Python we store a collection of characters in a data type known as a string, i.e., a `str`. When we say "text" we will typically be using it in the usual sense of the word to refer to readable content. On the other hand, when we say "string" we will be referring to the

From the file: `strings.tex`

Python `str` data type which consists of a collection of methods and attributes (and the attributes include readable content, i.e., text). In this chapter we will review some of the points we have already learned about strings and introduce several new features of strings and tools for working with them.

9.1 String Basics

The following summarizes several string-related features or tools that we discussed previously:

- A string literal is enclosed in either single quotes, double quotes, or either quotation mark repeated three times. When the quotation mark is repeated three times, the string may span multiple lines.
- Strings are immutable.
- Strings can be concatenated using the plus operator.
- A string can be repeated by multiplying it by an integer.
- The `str()` function converts its argument to a string.
- A string is a sequence. Thus:
  - A string can be used as the iterable in the header of a `for`-loop (in which case the loop variable takes on the values of the individual characters of the string).
  - The `len()` function returns the length of a string, i.e., the number of characters.
  - An individual character in a string can be accessed by specifying the character's index within brackets.
  - Slicing can be used to obtain a portion (or all) of a string.

Listing 9.1 demonstrates several basic aspects of strings and string literals. Note that in line 5 triple quotes are used for a string literal that spans two lines. In this case the newline character becomes part of the string. In contrast to this, in line 13 a string is enclosed in single quotes and the string itself is written across two lines. To do this, the newline character is escaped (i.e., the backslash is entered immediately before typing return). In this case `s3` is a single-line string—the newline character is not embedded in the string. In line 21 a string is created that includes the hash symbol (`#`). When the hash symbol appears in a string, it becomes part of the string and does not specify the existence of a comment.

Listing 9.1 Demonstration of several basic string operations including the different ways in which a literal can be quoted.

```
1 >>> s1 = 'This is a string.'
2 >>> print(s1)
3 This is a string.
4 >>> # Concatenation of string literals quoted in different ways.
```

```
5  >>> s2 = "This" + 'is' + """also""" + '''a
6  ... string.'''
7  >>> s2          # Check contents of s2.
8  'Thisisalsoa\nstring.'
9  >>> print(s2) # Printed version of s2.
10 Thisisalsoa
11 string.
12 >>> # Single-line string written across multiple lines.
13 >>> s3 = "this is a \
14 ... single line"
15 >>> print(s3)
16 this is a single line
17 >>> s4 = "Why? " * 3   # String repetition.
18 >>> print(s4)
19 Why? Why? Why?
20 >>> # Within a string the hash symbol does not indicate a comment.
21 >>> s5 = "this is # not a comment"
22 >>> print(s5)
23 this is # not a comment
```

Listing 9.2 demonstrates the conversion of an integer, a `float`, and a `list` to strings using the `str()` function. In Python every object has a `__str__()` method that is used, for instance, by `print()` to determine how to display that object. When an object is passed to the `str()` function, this simply serves to call the `__str__()` method for that object. So, when we write `str(<object>)` we are essentially asking for that object's string representation (as would be used in a `print()` statement). In line 5 in Listing 9.2 each of the objects is converted to a string and concatenated with the other strings to make a rather messy string. Note that in line 5 the string `s0` is passed as the argument to the `str()` function. Because `s0` is already a string, this is unnecessary, but it is done here to illustrate it is not an error to pass a string to `str()`.

Listing 9.2 Demonstration of the use of the `str()` function to convert objects to strings.

```
1 >>> x = 10
2 >>> y = 23.4
3 >>> zlist = [1, 'a', [2, 3]]
4 >>> s0 = "Hi!"
5 >>> all = str(x) + str(y) + str(zlist) + str(s0)
6 >>> print(all)
7 1023.4[1, 'a', [2, 3]]Hi!
```

Listing 9.3 provides code that demonstrates that strings are immutable sequences. A string is created in line 1 that has embedded quotes. If the embedded quotes are distinct from the surrounding quotes, this does not pose a difficulty.[1] In line 10 an attempt is made to change the second

[1]To embed a quotation mark of the same type as the surrounding quotation marks requires the embedded quotation mark to be escaped as discussed in Sec. 9.3.

character of the string from an uppercase U to a lowercase u. This fails because of the immutability of strings. The remaining code illustrates two different constructs for looping over the characters of a string. Because the print() statement in line 16 has end set to a hyphen, the subsequent interactive prompt appears on the same line as the output (as shown in line 18). In this particular session the user typed return to obtain another prompt on a new line (line 19), but a command could have been entered on line 18. The for-loop in lines 19 and 20 has the loop variable vary between zero and one less than the length of the string. The print() statement in line 20 uses the negative of this variable as the index for a character of the string. Since zero is neither positive nor negative, -0, +0, and 0 are all equivalent. Thus, the first character produced by the loop (line 22) is the first character of the string. The subsequent output (lines 23 through 27) progresses from the back of the string toward the front. Although not shown here, strings can be used with the enumerate() function so that characters and their corresponding indices can be obtained simultaneously in the header of a for-loop.

Listing 9.3 Demonstration that strings are immutable sequences.

```
>>> s1 = '"Unbroken" by Laura Hillenbrand'
>>> print(s1)
"Unbroken" by Laura Hillenbrand
>>> len(s1)       # Number of characters.
32
>>> s1[16 : 20]    # Slice of s1.
'aura'
>>> s1[1]         # Second character of s1.
'U'
>>> s1[1] = 'u'   # Attempt to change s1.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s2 = "Loopy!"
>>> for ch in s2:
...     print(ch, end="-")
...
L-o-o-p-y-!->>>
>>> for i in range(len(s2)):
...     print(-i, s2[-i])
...
0 L
-1 !
-2 y
-3 p
-4 o
-5 o
```

9.2 The ASCII Characters

As discussed in Chap. 1 and in the introduction to this chapter, all data are stored as collections of ones and zeros. This is true of the characters of a string as well. In the early 1960's a standard was published, known as the American Standard Code for Information Interchange (ASCII, pronounced "ask-ee"), which specifies a mapping of binary values to characters. The word "code" is not meant to imply anything having to do with encryption. Rather, this is a mapping of numeric values to 128 characters. These characters, together with their ASCII values, are shown in Listing 9.4. This character set consists of both non-printable characters and graphic (or printable) characters. The non-printable characters are indicated by a two- or three-letter abbreviation and these are often identified as "control characters." Some of the non-printable characters are discussed following the listing and the remainder are listed in an appendix.

Listing 9.4 The 128 ASCII characters together with their corresponding ASCII value. Non-printable characters are listed by their two- or three-letter abbreviations. The space character (32) is indicated by a blank space.

<table>
<tr><td>0 NUL</td><td>1 SOH</td><td>2 STX</td><td>3 ETX</td><td>4 EOT</td><td>5 ENQ</td><td>6 ACK</td><td>7 BEL</td></tr>
<tr><td>8 BS</td><td>9 HT</td><td>10 LF</td><td>11 VT</td><td>12 FF</td><td>13 CR</td><td>14 SO</td><td>15 SI</td></tr>
<tr><td>16 DLE</td><td>17 DC1</td><td>18 DC2</td><td>19 DC3</td><td>20 DC4</td><td>21 NAK</td><td>22 SYN</td><td>23 ETB</td></tr>
<tr><td>24 CAN</td><td>25 EM</td><td>26 SUB</td><td>27 ESC</td><td>28 FS</td><td>29 GS</td><td>30 RS</td><td>31 US</td></tr>
<tr><td>32</td><td>33 !</td><td>34 "</td><td>35 #</td><td>36 $</td><td>37 %</td><td>38 &</td><td>39 '</td></tr>
<tr><td>40 (</td><td>41)</td><td>42 *</td><td>43 +</td><td>44 ,</td><td>45 -</td><td>46 .</td><td>47 /</td></tr>
<tr><td>48 0</td><td>49 1</td><td>50 2</td><td>51 3</td><td>52 4</td><td>53 5</td><td>54 6</td><td>55 7</td></tr>
<tr><td>56 8</td><td>57 9</td><td>58 :</td><td>59 ;</td><td>60 <</td><td>61 =</td><td>62 ></td><td>63 ?</td></tr>
<tr><td>64 @</td><td>65 A</td><td>66 B</td><td>67 C</td><td>68 D</td><td>69 E</td><td>70 F</td><td>71 G</td></tr>
<tr><td>72 H</td><td>73 I</td><td>74 J</td><td>75 K</td><td>76 L</td><td>77 M</td><td>78 N</td><td>79 O</td></tr>
<tr><td>80 P</td><td>81 Q</td><td>82 R</td><td>83 S</td><td>84 T</td><td>85 U</td><td>86 V</td><td>87 W</td></tr>
<tr><td>88 X</td><td>89 Y</td><td>90 Z</td><td>91 [</td><td>92 \</td><td>93]</td><td>94 ^</td><td>95 _</td></tr>
<tr><td>96 `</td><td>97 a</td><td>98 b</td><td>99 c</td><td>100 d</td><td>101 e</td><td>102 f</td><td>103 g</td></tr>
<tr><td>104 h</td><td>105 i</td><td>106 j</td><td>107 k</td><td>108 l</td><td>109 m</td><td>110 n</td><td>111 o</td></tr>
<tr><td>112 p</td><td>113 q</td><td>114 r</td><td>115 s</td><td>116 t</td><td>117 u</td><td>118 v</td><td>119 w</td></tr>
<tr><td>120 x</td><td>121 y</td><td>122 z</td><td>123 {</td><td>124 |</td><td>125 }</td><td>126 ~</td><td>127 DEL</td></tr>
</table>

The ASCII character set is obviously limited in that it was not designed to handle characters from other languages. Newer codes exist, such as Unicode, that attempt to accommodate all languages. The Unicode standard currently defines more than 110,000 characters! Although Python provides support for Unicode, we will use ASCII almost exclusively. One thing to be aware of is that the larger codes invariably include ASCII as a subset. Thus, everything we will consider here remains true in the larger character sets.

ASCII is also somewhat dated in that it was designed to accommodate hardware that existed in the early 1960's and yet it did not precisely specify the behavior associated with some of the non-printable characters. This led to some confusion and incompatibilities. For example, some

manufacturers interpreted Line Feed (10) to indicate that output should continue at the beginning of the next line. However, other manufacturers required a Carriage Return (13) preceding a Line Feed to indicate that output should continue at the beginning of the next line. In Python when we say a “newline character” or simply “newline,” we mean the ASCII Line Feed character (10) and take this, by itself, to mean output should continue on the beginning of the next line.

The distinction between graphic (printable) and non-printable characters is somewhat fuzzy. For example, Horizontal Tab (9) is considered a non-printable character while the space character (32) is classified as a “non-printing graphic character.” However the Horizontal Tab typically produces between one and eight spaces. So, why consider Horizontal Tab a non-printable character and the space character a graphic character? Also, although Line Feed doesn’t produce ink on the page, so to speak, it does affect the appearance of the output. So, why not also consider it a non-printing graphic character? But these types of question do not truly concern us so we will simply accept that there are 33 non-printable characters, 95 graphic characters, and 128 total characters in the ASCII character set.

Because there are 128 total ASCII characters, all the characters can be represented using seven bits, i.e., $2^7 = 128$. However, characters are almost invariably stored using eight bits, otherwise known as one byte.[2]

9.3 Escape Sequences

Escape sequences were introduced in Sec. 2.5 where we saw we could write `\n` to include a newline character within a string. The backslash is used to indicate that the character following it does not have the usual meaning we associate with it within a string context (sometimes the backslash escapes multiple characters as will be shown). Given the description above of ASCII characters, we know the newline character (LF with a value of 10) is really no more than a binary value corresponding to the decimal number 10. To represent a newline character we write `\n`. Put another way, `\n` serves to place the ASCII Line Feed (LF) character in a string (i.e., the numeric equivalent of 10 is placed in the string). This escape sequence is used by Python and most other major computer languages.

As listed in Listing 9.5, Python provides escape sequences for eight of the ASCII characters. It is unlikely you will ever need Null (`\0`), Backspace (`\b`), Vertical Tab (`\v`), Form Feed (`\f`), or Carriage Return (`\r`). On the other hand, Bell (`\a`) can be useful. It typically causes the computer to “beep” or sound some tone.[3] Horizontal Tab (`\t`) and Line Feed (`\n`) are used frequently and we often refer to them simply as tab and newline, respectively.

Listing 9.5 ASCII characters for which Python provides a two-character escape sequence.

[2]In the early days of computers the eighth bit was sometimes used as a *parity bit*. A parity bit can be used to check whether an error has occurred in the storage or transmission of data. For example, the value of the bit can ensure the total number of bits in the byte that are set to one sums to an even number. To do so, the parity bit is zero if the other bits sum to an even number and is one if the other bits sum to an odd number. A parity bit cannot be used to directly correct an error, but it can signal the occurrence of an error and then appropriate steps can be taken.

[3]However, if you try to print `\a` from within IDLE, generally this will *not* succeed in sounding the system “bell.”

ASCII Value	Abbrv.	Escape Sequence	Description
0	NUL	\0	Null character
7	BEL	\a	Bell (alert)
8	BS	\b	Backspace
9	HT	\t	Horizontal Tab (tab)
10	LF	\n	Line Feed (newline)
11	VT	\v	Vertical Tab
12	FF	\f	Form Feed
13	CR	\r	Carriage Return

You will often hear the term "whitespace" mentioned in connection with strings. Collectively, whitespace is any character that introduces space into a document (without printing anything). Thus, in addition to the space character, Horizontal Tab (tab), Line Feed (newline), Vertical Tab, Form Feed, and Carriage Return are all considered whitespace (ASCII values 9 through 13).

There are other escape sequences in Python that aren't directly associated with ASCII characters. These are listed in Listing 9.6. Several of these are not of direct concern but are given for the sake of completeness and with the understanding that you may encounter them even if you don't use them yourself (we will see an example of this in a moment).

Listing 9.6 Escape sequences that aren't explicitly related to ASCII characters. In the first escape sequence, \\<*newline*> means a backslash followed by the newline character that has been entered by hitting the enter (or return) key. For the other escape sequences, the terms between angle brackets are place-holders for an appropriate value. For example, \N{registered sign} produces ® while \x41 produces the character A since the decimal number 65 is written 41 in hexadecimal.

Escape Sequence	Description
\\<*newline*>	Ignore the following newline character
\\\\	Literal backslash (\\)
\\'	Literal single quote (')
\\"	Literal double quote (")
\N{<*name*>}	Unicode character <*name*>
\\<*ooo*>	Character with octal value <*ooo*>
\u<*hhhh*>	Character with 16-bit hexadecimal value <*hhhh*>
\U<*hhhhhhhh*>	Character with 32-bit hexadecimal value <*hhhhhhhh*>
\x<*hh*>	Character with 8-bit hexadecimal value <*hh*>

Listing 9.7 provides several examples of escape sequences. The string created in line 1 uses two backslashes to specify that there is a single backslash in the string. This string also includes the escape sequence for a newline character. The code following this demonstrates various ways of handling a string with embedded quotation marks. One way to handle an embedded quote is by enclosing the string in triple quotes. This is demonstrated in line 16 where the string is surrounded by a single quotation mark repeated three times. Python is not confused by the embedded single quote. However, we see on line 22 that this approach falls apart if we try to repeat the double

quotation mark three times. The problem is that the trailing edge of the string is a double quote. Python attempts to end the string with the first three double quotes it encounters on the right side of the string. But, this leaves the last double quote unaccounted for.

Listing 9.7 Demonstration of escape sequences for backslash and embedded quotes.

```
1  >>> s0 = "\\ and\nburn"
2  >>> print(s0)
3  \ and
4  burn
5  >>> s1 = 'I didn\'t know he said, "Know!"'
6  >>> print(s)
7  I didn't know he said, "Know!"
8  >>> s1
9  'I didn\'t know he said, "Know!"'
10 >>> s2 = "I didn't know he said, \"Know!\""
11 >>> print(s2)
12 I didn't know he said, "Know!"
13 >>> s2
14 'I didn\'t know he said, "Know!"'
15 >>> # Triple quotes can do away with need to escape embedded quotations.
16 >>> s3 = '''Now, I didn't know he said, "Know!"'''
17 >>> print(s3)
18 Now, I didn't know he said, "Know!"
19 >>> # Tripling the double quotation mark does not work for this string
20 >>> # because the desired string itself is terminated by a double
21 >>> # quotation mark.
22 >>> s4 = """I didn't know he said, "Know!""""
23   File "<stdin>", line 1
24     s4 = """I didn't know he said, "Know!""""
25                                              ^
26 SyntaxError: EOL while scanning string literal
```

Listing 9.8 demonstrates the embedding of the Null and Bell characters in a string (the embedding of Nulls is not something that is typically done, but the embedding of Bell characters is useful). In line 2 the variable `koan` is assigned to a string that contains four Null characters.[4] When this is printed, the Null characters do not appear (it is as if they are not in the string). However, the length of the string, as shown in lines 5 and 6, does reflect the fact that the string contains Null characters (the Null characters each occupy one byte of computer memory). Note that each Null character counts as a single character (and is stored as a single byte) despite the fact that we represent it in a string as the two-character escape sequence `\0`. The expression in line 9 serves to echo the contents of `koan`. The output in line 10 displays the Null characters as the escape sequence `\x00`. This is a two-digit hexadecimal (base 16) representation of the value. Thus, even though we wrote `\0`, Python displays this as `\x00`. A string with three Bell characters is created

[4] In some computer languages, such as C, a Null is used to indicate the termination of a string. This is not the case in Python.

in line 11. When this is printed, these characters do not appear visually, but they should appear aurally as a sounding of the computer's "bell." When we display the contents of this string, as done in line 14, Python again displays the characters in their hexadecimal form despite the fact that we entered the characters as `\a`.

Listing 9.8 Embedding Null and Bell in a string and demonstration that Python displays these as two-digit hexadecimal values.

```
>>> # Create a string with four embedded Null characters.
>>> koan = "I contain\0\0\0\0 nothing."
>>> print(koan)   # Null characters do not appear in output.
I contain nothing.
>>> len(koan)     # Each Null counts as one character.
22
>>> # Python represents the Null character by its two-digit
>>> # hexadecimal value.
>>> koan
'I contain\x00\x00\x00\x00 nothing.'
>>> alert = "Wake Up!\a\a\a"  # String with Bell characters.
>>> print(alert)              # Will sound computer's "bell."
Wake Up!
>>> alert                     # Internal representation of string.
'Wake Up!\x07\x07\x07'
>>>
```

9.4 chr() and ord()

The built-in function `chr()` takes an integer argument and produces the corresponding character. The built-in function `ord()` is effectively the inverse of `chr()`. `ord()` takes a string of length one, i.e., a single character, and returns the corresponding ASCII value. The code in Listing 9.9 illustrates the behavior of these functions.

Listing 9.9 Demonstration of the use of `ord()` and `chr()`.

```
>>> ord('A'), ord('B'), ord('!'), ord(' ')
(65, 66, 33, 32)
>>> chr(65), chr(66), chr(33), chr(32)
('A', 'B', '!', ' ')
>>> ord('Z'), chr(90) # 90 and Z are ASCII pairs.
(90, 'Z')
>>> ord(chr(90))      # ord(chr()) returns original argument.
90
>>> chr(ord('Z'))     # chr(ord()) returns original argument.
'Z'
```

```
11 >>> for i in range(65, 75):
12 ...     print(chr(i), end="")
13 ...
14 ABCDEFGHIJ>>>
15 >>> for ch in "ASCII = numbers":
16 ...     print(ord(ch), end=" ")
17 ...
18 65 83 67 73 73 32 61 32 110 117 109 98 101 114 115 >>>
```

In line 1 ord() is used to obtain the numeric values for four characters. Comparing the result in line 2 to the values given in Listing 9.4, we see these are indeed the appropriate ASCII values. In line 3 the numeric values produced by line 1 (i.e., the values on line 2) are used as the arguments of the chr() function. The output in line 4 corresponds to the characters used in line 1. The statements in lines 7 and 9 also confirm that ord() and chr() are inverses (i.e., one does the opposite of the other); each result is identical to the value provided as the argument of the inner function.

The for-loop in lines 11 and 12 sets the loop variable i to values between 65 and 74 (inclusive). Each value is used as the argument of chr() to produce the characters ABCDEFGHIJ, i.e., the first ten characters of the alphabet as shown in line 8 (the interactive prompt also appears on this line since the end parameter of the print() statement is set to the empty string). The for-loop in lines 15 and 16 sets the loop variable ch to the characters of the string that appears in the header. This variable is used as the argument of ord() to display the corresponding ASCII value for each character. The output is shown in line 18.

We have seen that we cannot add integers to strings. But, of course, we can add integers to integers and thus we can add integers to ASCII values. Note there is an offset of 32 between the ASCII values for uppercase and lowercase letters. Thus, if we have a string consisting of all uppercase letters we can easily convert it to lowercase by adding 32 to each letter's ASCII value and then converting that back to a character. This is demonstrated in Listing 9.10 where, in line 1, we start with an uppercase string. Line 2 is used to display the ASCII value of the first character. Lines 4 and 5 show the integer value obtained by adding 32 to the first character's ASCII value. Then, in lines 6 and 7, we see that by converting this offset value back to a character, we obtain the lowercase equivalent of the first character. Line 8 initializes the variable soft to the empty string. This variable will serve as a string accumulator with which we build the lowercase version of the uppercase string. For each iteration of the for-loop shown in lines 9 through 11 we concatenate an additional character to this accumulator. The print() statement in line 11 shows the accumulator for each iteration of the loop (and after the concatenation). The output in lines 13 through 18 show that we do indeed obtain the lowercase equivalent of the original string.

Listing 9.10 Demonstration of the use of an integer offset to convert one character to another.

```
1 >>> loud = "SCREAM"          # All uppercase.
2 >>> ord(loud[0])             # ord() of first character.
3 83
4 >>> ord(loud[0]) + 32        # ord() of first character plus 32.
5 115
```

```
6  >>> chr(ord(loud[0]) + 32) # Lowercase version of first character.
7  's'
8  >>> soft = ""              # Empty string accumulator
9  >>> for ch in loud:
10 ...     soft = soft + chr(ord(ch) + 32) # Concatenate to accumulator.
11 ...     print(soft)         # Show value of accumulator.
12 ...
13 s
14 sc
15 scr
16 scre
17 screa
18 scream
```

More interesting, perhaps, is when we add some other offset to the ASCII values of a string. If we make the offset somewhat arbitrary, the original string may start as perfectly readable text, but the resulting string will probably end up looking like nonsense. However, if we can later remove the offset that turned things into nonsense, we can get back the original readable text. This type of modification and reconstruction of strings is, in fact, the basis for encryption and decryption!

As an example of simple encryption, consider the code shown in Listing 9.11. We start, in line 1, with the unencrypted text `CLEARLY`, which we identify as the "clear text" and store as the variable `clear_text`. Line two contains a list of randomly chosen offsets that we store in the list `keys`. The number of offsets in this list corresponds to the number of characters in the clear text, but this is not strictly necessary—there should be at least as many offsets as characters in the string, but there can be more offsets (they simply won't be used). In line 3 we initialize the accumulator `cipher_text` to the empty string. The `for`-loop in lines 4 through 6 "zips" together the offsets and the character of clear text, i.e., for each iteration of the loop the loop variables `offset` and `ch` correspond to an offset from `keys` and a character from `clear_text`. Line 5, the first line in the body of the `for`-loop, adds the offset to the ASCII value of the character, concatenates this to the accumulator, and stores the result back in the accumulator. The `print()` statement in line 6 shows the progression of the calculation. The final result, shown in line 16, is a string that has no resemblance to the original clear text.

Listing 9.11 Demonstration of encryption by adding arbitrary offsets to the ASCII values of the characters of clear text.

```
1  >>> clear_text = "CLEARLY"             # Initial clear text.
2  >>> keys = [5, 8, 27, 45, 17, 31, 5]  # Offsets to be added.
3  >>> cipher_text = ""
4  >>> for offset, ch in zip(keys, clear_text):
5  ...     cipher_text = cipher_text + chr(ord(ch) + offset)
6  ...     print(ch, offset, cipher_text)
7  ...
8  C 5 H
9  L 8 HT
10 E 27 HT`
```

```
A 45 HT`n
R 17 HT`nc
L 31 HT`nck
Y 5 HT`nck^
>>> print(cipher_text)                  # Display final cipher text.
HT`nck^
```

Now, let's go the other way: let's start with the encrypted text, also known as the cipher text, and try to reconstruct the clear text. We must have the same keys in order to subtract the offsets. The code in Listing 9.12 illustrates how this can be done. This code is remarkably similar to the code in Listing 9.11. In fact, other than the change of variable names and the different starting string, the only difference is in the body of the `for`-loop where we subtract the offset rather than add it. Note that in line 1 we start with cipher text. Given the keys/offsets used to construct this cipher text, we are able to recreate the clear text as shown in line 16.

Listing 9.12 Code to decrypt the string that was encrypted in Listing 9.11.

```
>>> cipher_text = "HT`nck^"             # Initial cipher text.
>>> keys = [5, 8, 27, 45, 17, 31, 5]   # Offsets to be subtracted.
>>> clear_text = ""
>>> for offset, ch in zip(keys, cipher_text):
...     clear_text = clear_text + chr(ord(ch) - offset)
...     print(ch, offset, clear_text)
...
H 5 C
T 8 CL
` 27 CLE
n 45 CLEA
c 17 CLEAR
k 31 CLEARL
^ 5 CLEARLY
>>> print(clear_text)                   # Display final clear text.
CLEARLY
```

Let's continue to explore encryption a bit further. Obviously there is a shortcoming to this exchange of information in that both the sender and the receiver have to have the same keys. How do they exchange the keys? If a third party (i.e., an eavesdropper) were able to gain access to these keys, then they could easily decipher the cipher text, too. So exchanging (and protecting) keys can certainly be a problem. Suppose the parties exchanging information work out a system in which they generate keys "on the fly" and "in the open." Say, for instance, they agree to generate keys based on the first line of the third story published on National Public Radio's Morning Edition Web site each day. This information is there for all to see, but who would think to use this as the basis for generating keys? But keep in mind that the characters in the story, once converted to ASCII, are just collections of integers. One can use as many characters as needed from the story to encrypt a string (if the string doesn't have more characters than the story).

We'll demonstrate this in a moment, but let's first consider a couple of building blocks. We have previously used the `zip()` function which pairs the elements of two sequences. This function can be called with sequences of different lengths. When the sequences are of different lengths, the pairing is only as long as the shorter sequence. Let's assume the clear text consists only of spaces and uppercase letters (we can easily remove this restriction later, but it does make the implementation simpler). When we add the offset, we want to ensure we still have a valid ASCII value. Given that the largest ASCII value for the clear text can be 90 (corresponding to `Z`) and the largest possible ASCII value for a printable character is 126, we cannot use an offset larger than 36. If we take an integer modulo 37, we obtain a result between 0 and 36. Given this, consider the code in Listing 9.13 which illustrates the behavior of `zip()` and shows how to obtain an integer offset between 0 and 36 from a line of text.[5] The code is discussed below the listing.

Listing 9.13 Demonstration of `zip()` with sequences of different lengths. The `for`-loop in lines 5 and 6 converts a string to a collection of integer values. These values can be used as the "offsets" in an encryption scheme.

```
1  >>> keys = "The 19th century psychologist William James once said"
2  >>> s = "Short"
3  >>> list(zip(keys, s))
4  [('T', 'S'), ('h', 'h'), ('e', 'o'), (' ', 'r'), ('1', 't')]
5  >>> for ch in keys:
6  ...     print(ord(ch) % 37, end=" ")
7  ...
8  10 30 27 32 12 20 5 30 32 25 27 36 5 6 3 10 32 1 4 10 25 30 0 34 0 29
9  31 4 5 32 13 31 34 34 31 23 35 32 0 23 35 27 4 32 0 36 25 27 32 4
10 23 31 26
```

In line 1 the variable `keys` is assigned a "long" string. In line 2 the variable `s` is assigned a short string. In line 3 we zip these two strings together. The resulting list, shown in line 4, has only as many pairings as the shorter string, i.e., 5 elements. The `for`-loop in lines 5 and 6 loops over all the characters in `keys`. The ASCII value for each character is obtained using `ord()`. This value is taken modulo 37 to obtain a value between 0 and 36. These values are perfectly suited to be used as offsets for clear text that consists solely of uppercase letters (as well as whitespace).

Now let's use this approach to convert clear text to cipher text. Listing 9.14 starts with the clear text in line 1. We need to ensure that the string we use to generate the keys is at least this long. Line 2 shows the "key-generating string" that has been agreed upon by the sender and receiver. The `print()` statement in the body of the `for`-loop is not necessary but is used to show how each individual character of clear text is converted to a character of cipher text. Line 23 shows the resulting cipher text. Obviously this appears to be indecipherable nonsense to the casual observer. However, to the person with the key-generating string, it's meaningful!

Listing 9.14 Conversion of clear text to cipher text using offsets that come from a separate string (that the sender and receiver agree upon).

[5] This line came from **www.npr.org/templates/transcript/transcript.php?storyId=147296743**

```
>>> clear_text = "ONE IF BY LAND"
>>> keys = "The 19th century psychologist William James once said"
>>> cipher_text = ""
>>> for kch, ch in zip(keys, clear_text):
...     cipher_text = cipher_text + chr(ord(ch) + ord(kch) % 37)
...     print(kch, ch, cipher_text)
...
T O Y
h N Yl
e E Yl`
    Yl`@
1 I Yl`@U
9 F Yl`@UZ
t   Yl`@UZ%
h B Yl`@UZ%`
  Y Yl`@UZ%`y
c   Yl`@UZ%`y9
e L Yl`@UZ%`y9g
n A Yl`@UZ%`y9ge
t N Yl`@UZ%`y9geS
u D Yl`@UZ%`y9geSJ
>>> print(cipher_text)
Yl`@UZ%`y9geSJ
```

Listing 9.15 demonstrates how we can start from the cipher text and reconstruct the clear text. Again, this code parallels the code used to create the cipher text. The only difference is that we start with a different string (i.e., we're starting with the cipher text) and, instead of adding offsets, we subtract offsets. For the sake of brevity, the `print()` statement has been removed from the body of the `for`-loop. Nevertheless, lines 7 and 8 confirm that the clear text has been successfully reconstructed.

Listing 9.15 Reconstruction of the clear text from Listing 9.14 from the cipher text.

```
>>> cipher_text = "Yl`@UZ%`y9geSJ"
>>> keys = "The 19th century psychologist William James once said"
>>> clear_text = ""
>>> for kch, ch in zip(keys, cipher_text):
...     clear_text = clear_text + chr(ord(ch) - ord(kch) % 37)
...
>>> print(clear_text)
ONE IF BY LAND
```

9.5 Assorted String Methods

String objects provide several methods. These can be listed using the `dir()` function with an argument of a string literal, a string variable, or even the `str()` function (with or without the parentheses). So, for example, `dir("")` and `dir(str)` both list all the methods available for strings. This listing is shown in Listing 5.5 and not repeated here. To obtain help on a particular method, don't forget that `help()` is available. For example, to obtain help on the `count()` method, one could type `help("".count)`. In this section we will consider a few of the simpler string methods.

Something to keep in mind with all string methods is that they never change the value of a string. Indeed, they *cannot* change it because strings are immutable. If we want to modify the string associated with a particular identifier, we have to create a new string (perhaps using one or more string methods) and then assign this new string back to the original identifier. This is demonstrated in the examples below.

9.5.1 lower(), upper(), capitalize(), title(), and swapcase()

Methods dealing with case return a new string with the case of the original string appropriately modified. It is unlikely that you will have much need for the `swapcase()` method. `title()` and `capitalize()` can be useful at times, but the most useful case-related methods are `lower()` and `upper()`. These last two methods are used frequently to implement code that yields the same result independent of the case of the input, i.e., these methods are used to make the code "insensitive" to case. For example, if input may be in either uppercase or lowercase, or even a mix of cases, and yet we want to code to do the same thing regardless of case, we can use `lower()` to ensure the resulting input string is in lowercase and then process the string accordingly. (Or, similarly, we could use `upper()` to ensure the resulting input string is in uppercase.) These five methods are demonstrated in Listing 9.16. The `print()` statement in line 12 and the subsequent output in line 13 show that the value of the string `s` has been unchanged by the calls to the various methods. To change the string associated with `s` we have to assign a new value to it as is done in line 14 where `s` is reset to the lowercase version of the original string. The `print()` statement in line 15 confirms that `s` has changed.

Listing 9.16 Demonstration of the methods used to establish the case of a string.

```
1  >>> s = "Is this IT!?"
2  >>> s.capitalize() # Capitalize only first letter of entire string.
3  'Is this it!?'
4  >>> s.title()      # Capitalize first letter of each word.
5  'Is This It!?'
6  >>> s.swapcase()   # Swap uppercase and lowercase.
7  'iS THIS it!?'
8  >>> s.upper()      # All uppercase.
9  'IS THIS IT!?'
10 >>> s.lower()      # All lowercase.
11 'is this it!?'
```

```
12 >>> print(s)        # Show original string s unchanged.
13 Is this IT!?
14 >>> s = s.lower() # Assign new value to s.
15 >>> print(s)        # Show that s now all lowercase.
16 is this it!?
```

9.5.2 count()

The `count()` method returns the number of "non-overlapping" occurrences of a substring within a given string. By non-overlapping we mean that a character cannot be counted twice. So, for example, the number of times the (sub)string `zz` occurs in `zzz` is once. (If overlapping were allowed, the middle `z` could serve as the end of one `zz` and the start of a second `zz` and thus the count would be two. But this is *not* what is done.) The matching is case sensitive. Listing 9.17 demonstrates the use of `count()`. Please see the comments within the listing.

Listing 9.17 Demonstration of the `count()` method.

```
1 >>> s = "I think, therefore I am."
2 >>> s.count("I")    # Check number of uppercase I's in s.
3 2
4 >>> s.count("i")    # Check number of lowercase i's in s.
5 1
6 >>> s.count("re")   # Check number of re's in s.
7 2
8 >>> s.count("you") # Unmatched substrings result in 0.
9 0
```

9.5.3 strip(), lstrip(), and rstrip()

`strip()`, `lstrip()`, and `rstrip()` remove leading and/or trailing whitespace from a string. `lstrip()` removes leading whitespace (i.e., removes it from the left of the string), `rstrip()` removes trailing whitespace (i.e., removes it from the right of the string), and `strip()` removes both leading and trailing whitespace. As we will see in Sec. 10.1.1, when reading lines of text from a file, the newline character terminating the line is considered part of the text. Such terminating whitespace is often unwanted and can be conveniently removed with `strip()` or `rstrip()`. Listing 9.18 demonstrates the behavior of these methods. Please read the comments within the listing.

Listing 9.18 Demonstration of the `strip()`, `lstrip()`, and `rstrip()` methods.

```
1 >>> # Create a string with leading, trailing, and embedded whitespace.
2 >>> s = "   Indent a line\n    and another\nbut not this.   \n"
3 >>> print(s)            # Print original string.
```

```
4    Indent a line
5    and another
6  but not this.
7
8  >>> print(s.lstrip()) # Remove leading whitespace.
9  Indent a line
10    and another
11 but not this.
12
13 >>> print(s.rstrip()) # Remove trailing whitespace.
14    Indent a line
15    and another
16 but not this.
17 >>> print(s.strip())  # Remove leading and trailing whitespace.
18 Indent a line
19    and another
20 but not this.
21 >>> # Create a new string with leading and trailing whitespace removed.
22 >>> s_new = s.strip()
23 >>> # Use echoed value in interactive environment to see all leading and
24 >>> # trailing whitespace removed.
25 >>> s_new
26 'Indent a line\n   and another\nbut not this.'
```

9.5.4 `__repr__()`

All objects in Python contain the method `__repr__()`. This method provides the "official" string representation of a given object, i.e., you can think of `__repr__()` as standing for "representation." As with most methods that begin with underscores, this is not a method you typically use in day-to-day programming. However, it can be useful when debugging. Consider the string `s_new` created in line 22 of Listing 9.18. In line 25 this string is entered at the interactive prompt. The interactive environment echoes the string representation of this object. *But*, this is not the same as printing the object. Were we to print `s_new`, we would see the same output shown in lines 18 through 20. Note that in this output we cannot truly tell if the spaces were removed from the end of the line (we can tell the newline character was removed, but not the spaces that came before the newline character). However, from the output in line 26, we can see that these trailing spaces were removed.

Now assume you are writing (and running) a program but not working directly in the interactive environment. You want to quickly see if the internal representation of a string (such as `s_new` in the example above) is correct. What should you do? If you `print()` the string itself, it might mask the detail you seek. Instead, you can use the `__repr__()` method with `print()` to see the internal representation of the object. This is illustrated in Listing 9.19.

Listing 9.19 Demonstration that the `__repr__()` method gives the internal representation of a string.

```
>>> s = "   Indent a line\n   and another\nbut not this.   \n"
>>> s_new = s.strip()      # Create stripped string.
>>> # Print stripped string.  Cannot tell if all trailing space removed.
>>> print(s_new)
Indent a line
   and another
but not this.
>>> # Print __repr__() of stripped string.  Shows trailing space removed.
>>> print(s_new.__repr__())
'Indent a line\n   and another\nbut not this.'
>>> # Print __repr__() of original string.
>>> print(s.__repr__())
'   Indent a line\n   and another\nbut not this.   \n'
```

9.5.5 find() and index()

The `find()` and `index()` methods search for one string within another. The search is case sensitive. Both return the starting index where a substring can be found within a string. They only differ in that `find()` returns `-1` if the substring is not found while `index()` generates a `ValueError` (i.e., an exception) if the substring is not found. You may wonder why there is a need for these two different functions. Other than for convenience, there isn't a true need for two different functions. However, note that, because of negative indexing, `-1` is a valid index: it corresponds to the last element of the string. So, `find()` always returns a valid index and it is up to your code to recognize that `-1` really means "not found." In some situations it may be preferable to produce an error when the substring is not found. But, if you do not want this error to terminate your program, you must provide additional code to handle the exception.

The code in Listing 9.20 demonstrates basic operation of these two methods. In line 1 a string is created that has the character `I` in the first and 20th positions, i.e., in locations corresponding to indices of `0` and `19`. The `find()` method is used in line 2 to search for an occurrence of `I` and the result, shown in line 3, indicates `I` is the first character of the string. You may wonder if it is possible to find all occurrences of a substring, rather than just the first. The answer is yes and we'll return to this issue following the listing. Lines 4 and 5 show that the search is case sensitive, i.e., `i` and `I` do not match. Line 6 shows that one can search for a multi-character substring within a string. Here `index()` is used to search for `ere` in the string `s`. The resulting value of `11` gives the index where the substring starts within the string. In line 8 `find()` is used to search for `You` within the string. Since this does not occur in `s`, the result is `-1` as shown in line 9. Finally, in line 10, `index()` is used to search for `You`. Because this string is not found, `index()` produces a `ValueError`.

Listing 9.20 Demonstration of the use of the `find()` and `index()` methods.

```
1  >>> s = "I think, therefore I am."
2  >>> s.find("I")     # Find first occurrence of I.
3  0
4  >>> s.find("i")     # Search is case sensitive.
5  4
6  >>> s.index("ere") # Find first occurrence of ere.
7  11
8  >>> s.find("You")  # Search for You.  Not found.  Result of -1.
9  -1
10 >>> s.index("You") # Search for You.  Not Found.  Result is Error.
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 ValueError: substring not found
```

Let's return to the question of finding more than one occurrence of a substring. Both `find()` and `index()` have two additional (optional) arguments that can be used to specify the range of indices over which to perform a search. By providing both the optional `start` and `stop` arguments, the search will start from `start` and go up to, but not include, the `stop` value. If a `stop` value is not given, it defaults to the end of the string.

To demonstrate the use of these optional arguments, consider the code shown in Listing 9.21. In line 1 the same string is defined as in line 1 of Listing 9.20. This has `I`'s in the first and 20th positions. As in Listing 9.20, the `find()` method is used in line 2 without an optional argument to search for an occurrence of `I`. The result indicates `I` is the first character of the string. In line 4 `find()` is again asked to search for an `I` but the optional start value of `1` tells `find()` to start its search offset `1` from the beginning of the string, i.e., the search starts from the second character which is one beyond where the first occurrence is found. In this case `find()` reports that `I` is also the 20th character of the string. The statement in line 6 checks whether there is another occurrence by continuing the search from index 20. In this case `find()` reports, by returning `-1`, that the search failed. The discussion continues following the listing.

Listing 9.21 Demonstration of the use of the optional start argument for `find()`. `index()` behaves the same way except when the substring is not found (in which case an error is produced).

```
1  >>> s = "I think, therefore I am."
2  >>> s.find("I")        # Search for first occurrence.
3  0
4  >>> s.find("I", 1)     # Search for second occurrence.
5  19
6  >>> s.find("I", 20)    # Search for third occurrence.
7  -1
8  >>> s[1 : ].find("I") # Search on slice differs from full string.
9  18
```

As Listing 9.21 demonstrates, we can find multiple occurrences by continuing the search from the index one greater than that given for a previous successful search. Note that this continued search is related to, but slightly different from, searching a slice of the original string that starts just beyond the index of the previous successful search. This is illustrated in lines 7 and 8. In line 7 the slice `s[1 : ]` excludes the leading `I` from `s`. Thus, `find()` finds the second `I`, but it reports this as the 19th character (since it is indeed the 19th character of this slice but it is not the 19th character of the original string). If we exchange `index()` for `find()` in Listing 9.21, the results would be the same except when the substring is not found (in which case an exception is thrown instead of the method returning `-1`).

Let's consider a more complicated example in which we use a `for`-loop to search for all occurrences of a substring within a given string. Each occurrence is displayed with a portion of the "trailing context" in which the substring occurred. Code to accomplish this is shown in Listing 9.22. The code is discussed following the listing.

Listing 9.22 Code to search for multiple occurrences of a substring and to display the context in which the substring is found.

```
1  >>> s = "I think, therefore I am."
2  >>> look_for = "I" # Substring to search for.
3  >>> # Initialize variable that specifies where search starts.
4  >>> start_index = -1
5  >>> for i in range(s.count(look_for)):
6  ...     start_index = s.index(look_for, start_index + 1)
7  ...     print(s[start_index : start_index + len(look_for) + 5])
8  ...
9  I thin
10 I am.
11 >>> look_for = "th"
12 >>> start_index = -1
13 >>> for i in range(s.count(look_for)):
14 ...     start_index = s.index(look_for, start_index + 1)
15 ...     print(s[start_index : start_index + len(look_for) + 5])
16 ...
17 think,
18 therefo
```

In line 1 we again create a string with two `I`'s. In line 2 the variable `look_for`, which specifies the substring of interest, is set to `I`. In line 4 `start_index` is set to the integer value `-1`. This variable is used in the loop both to indicate where the substring was found and where a search should start. The starting point for a search is actually given by `start_index + 1`, hence `start_index` is initialized to `-1` to ensure that the first search starts from the character with an index of `0`. The header of the `for`-loop in line 5 yields a simple counted loop (i.e., we never use the loop variable). The `count()` method is used in line 5 to ensure the number of times the loop is executed is the number of times `look_for` occurs in `s`.

The first statement in the body of the `for`-loop, line 6, uses the `index()` method with a starting index of `start_index + 1` to determine where `look_for` is found. (Since the body of

the loop only executes as many times as the substring exists in s, we will not encounter a situation where the substring is not found. Thus, it doesn't matter if we use find() or index().) In line 6 the variable start_index is reset to the index where the substring was found. The print() statement in line 7 prints a slice of s that starts where the substring was found. The output is the substring itself with up to an additional five characters. Recall it is not an error to specify a slice that has a start or stop value that is outside the range of valid indices. For the second I in the string s there are, in fact, only four characters past the location of this substring. This is indicated in the output that appears in lines 9 and 10.

In line 11 the variable look_for is set to th (i.e., the substring for which the search will be performed is now two characters long) and start_index is reset to -1. Then, the same for-loop is executed as before. The output in lines 17 and 18 shows the two occurrences of the substring th within the string s together with the next five characters beyond the substring (although in line 17 the final character is the space character so it looks like only four additional characters are shown).

Something to note is that it is possible to store the *entire* contents of a file as a single string. Assume we want to search for occurrences of a particular (sub)string within this file. We can easily do so and, for each occurrence, display the substring with both a bit of leading and trailing context using a construct such as shown in Listing 9.22.

9.5.6 replace()

The replace() method is used to replace one substring by another. By default, all occurrences of the string targeted for replacement are replaced. An optional third argument can be given that specifies the maximum number of replacements. The use of the replace() method is demonstrated in Listing 9.23.

Listing 9.23 Demonstration of the replace() method. The optional third argument specifies the maximum number of replacements. It is not an error if the substring targeted for replacement does not occur.

```
1 >>> s = "I think, therefore I am."
2 >>> s.replace("I", "You")       # Replace I with You.
3 'You think, therefore You am.'
4 >>> s.replace("I", "You", 1)    # Replace only once.
5 'You think, therefore I am.'
6 >>> s.replace("I", "You", 5)    # Replace up to 5 times.
7 'You think, therefore You am.'
8 >>> s.replace("He", "She", 5)   # Target substring not in string.
9 'I think, therefore I am.'
```

9.6 split() and join()

split() breaks apart the original string and places each of the pieces in a list. If no argument is given, the splitting is done at every whitespace occurrence; consecutive whitespace characters

are treated the same way as a single whitespace character. If an argument is given, it is the substring at which the split should occur. We will call this substring the separator. Consecutive occurrences of the separator cause repeated splits and, as we shall see, produce an empty string in the resulting `list`.

Listing 9.24 demonstrates the behavior of `split()`. In line 1 a string is created containing multiple whitespace characters (spaces, newline, and tabs). The `print()` statement in line 2 shows the formatted appearance of the string. In line 5 the `split()` method is used to produce a `list` of the words in the string and we see the result in line 6. The `for`-loop in lines 7 and 8 uses `s.split()` as the iterable to cycle over each word in the string.

Listing 9.24 Demonstration of the `split()` method using the default argument, i.e., the splitting occurs on whitespace.

```
>>> s = "This  is \n only\t\ta test."
>>> print(s)
This  is
 only           a test.
>>> s.split()
['This', 'is', 'only', 'a', 'test.']
>>> for word in s.split():
...     print(word)
...
This
is
only
a
test.
```

Now consider the code in Listing 9.25 where `split()` is again called, but now an explicit argument is given. In line 1 a string is created with multiple repeated characters. In line 2 the splitting is done at the separator `ss`. Because there are two occurrences of `ss` in the original string, the resulting `list` has three elements. In line 4 the separator is `i`. Because there are four `i`'s in the string, the resulting `list` has five elements. However, since the final `i` is at the end of the string, the final element of the `list` is an empty string. When the separator is `s`, the resulting `list` contains two empty strings since there are two instances of repeated `s`'s in the string. The call to `split()` in line 8, with a separator of `iss`, results in a single empty string in the `list` because there are two consecutive occurrences of the separator `iss`.

Listing 9.25 Demonstration of the `split()` method when the separator is explicitly given.

```
>>> s = "Mississippi"
>>> s.split("ss")
['Mi', 'i', 'ippi']
>>> s.split("i")
['M', 'ss', 'ss', 'pp', '']
```

```
6  >>> s.split("s")
7  ['Mi', '', 'i', '', 'ippi']
8  >>> s.split("iss")
9  ['M', '', 'ippi']
```

Now let's consider an example that is perhaps a bit more practical. Assume there is `list` of strings corresponding to names. The names are written as a last name, then a comma and a space, and then the first name. The goal is to write these names as the first name followed by the last name (with no comma). The code in Listing 9.26 shows one way to accomplish this. The `list` of names is created in line 1. The `for`-loop cycles over all the names, setting the loop variable `name` to each of the individual names. The first line in the body of the loop uses the `split()` method with a separator consisting of both a comma and a space. This produces a `list` in which the last name is the first element and the first name is the second element. Simultaneous assignment is used to assign these values to appropriately named variables. The next line in the body of the for-loop simply prints these in the desired order. The output in lines 6 through 8 shows the code is successful.

Listing 9.26 Rearrangement of a collection of names that are given as last name followed by first name into output where the first name is followed by the last name.

```
1  >>> names = ["Obama, Barack", "Bush, George", "Jefferson, Thomas"]
2  >>> for name in names:
3  ...     last, first = name.split(", ")
4  ...     print(first, last)
5  ...
6  Barack Obama
7  George Bush
8  Thomas Jefferson
```

The `join()` method is, in many ways, the converse of the `split()` method. Since `join()` is a string method, it must be called with (or on) a particular string object. Let's also call this string object the separator (for reasons that will be apparent in a moment). The `join()` method takes as its argument a `list` of strings.[6] The strings from the `list` are joined together to form a single new string but the separator is placed between these strings. The separator may be the empty string. Listing 9.27 demonstrates the behavior of the `join` method.

Listing 9.27 Demonstration of the `join()` method.

```
1  >>> # Separate elements from list with a comma and space.
2  >>> ", ".join(["Obama", "Barack"])
3  'Obama, Barack'
4  >>> "-".join(["one", "by", "one"])  # Hyphen separator.
```

[6]In fact the argument can be any iterable that produces strings, but for now we will simply say the argument is a `list` of strings.

```
5  'one-by-one'
6  >>> "".join(["smashed", "together"]) # No separator.
7  'smashedtogether'
```

Let's construct something a little more complicated. Assume we want a function to count the number of printable characters (excluding space) in a string. We want to count only the characters that produce something visible on the screen or page (and we will assume the string doesn't contain anything out of the ordinary such as Null or Backspace characters). This can be accomplished by first splitting a string on whitespace and then joining the list back together with an empty-space separator. The resulting string will have all the whitespace discarded and we can simply find the length of this string. Listing 9.28 demonstrates how to do this and provides a function that yields this count.

Listing 9.28 Technique to count the visible characters in a string.

```
1  >>> text = "A B C D B's?\nM N R no B's.   S A R!"
2  >>> print(text)
3  A B C D B's?
4  M N R no B's.   S A R!
5  >>> tlist = text.split()     # List of strings with no whitespace.
6  >>> clean = "".join(tlist)   # Single string with no whitespace.
7  >>> clean
8  "ABCDB's?MNRnoB's.SAR!"
9  >>> len(clean)               # Length of string.
10 21
11 >>> def ink_counter(s):
12 ...     slist = s.split()
13 ...     return len("".join(slist))
14 ...
15 >>> ink_counter(text)
16 21
17 >>> ink_counter("Hi    Ho")
18 4
```

9.7 Format Strings and the `format()` Method

We have used the `print()` function to generate output and we have used the interactive interpreter to display values (by entering an expression at the interactive prompt and then hitting return). In doing so, we have delegated the task of formatting the output to the the `print()` function or to the interactive interpreter.[7] Often, however, we need to exercise finer control over the appearance of our output. We can do this by creating strings that have precisely the appearance we desire. These strings are created using *format strings* in combination with the `format()` method.

[7]Actually, the formatting is largely dictated by the objects themselves since it is the `__str__()` method of an object that specifies how an object should appear.

Format strings consist of literal characters that we want to appear "as is" as well as *replacement fields* that are enclosed in braces. A replacement field serves a dual purpose. First, it serves as a placeholder, showing the relative placement where additional text should appear within a string. Second, a replacement field *may* specify various aspects concerning the formatting of the text, such as the number of spaces within the string to dedicate to displaying an object, the alignment of text within these spaces, the character to use to fill any space that would otherwise be unfilled, the digits of precision to use for `float`s, etc. We will consider several of these formatting options, but not all.[8] However, as will be shown, a replacement field need not specify formatting information. When formatting information is omitted, Python uses default formats (as is done when one simply supplies an object as an argument to the `print()` function).

To use a format string, you specify the format string *and* invoke the `format()` method on this string. The arguments of the `format()` method supply the objects that are used in the replacement fields. A replacement field is identified via a pair of (curly) braces, i.e., `{` and `}`.

Without the `format()` method, braces have no special meaning. To emphasize this, consider the code in Listing 9.29. The strings given as arguments to the `print()` functions in lines 1 and 3 and the string in line 5 all contain braces. If you compare these strings to the subsequent output in lines 2, 4, and 6, you see that the output mirrors the string literals—nothing has changed. However, as we will see, this is *not* the case when the `format()` method is invoked, i.e., `format()` imparts special meaning to braces and their contents.

Listing 9.29 Demonstration that braces within a string have no special significance without the `format()` method.

```
1 >>> print("These {}'s are simply braces.  These are too: {}")
2 These {}'s are simply braces.  These are too: {}
3 >>> print("Strange combination of characters: {:<20}")
4 Strange combination of characters: {:<20}
5 >>> "More strange characters: {:=^20.5f}"
6 'More strange characters: {:=^20.5f}'
```

9.7.1 Replacement Fields as Placeholders

For examples of the use of format strings, consider the code in Listing 9.30. In these examples the replacement fields, which are the braces, are simply placeholders—they mark the location (or locations) where the argument(s) of the `format()` method should be placed. There is a one-to-one correspondence between the (placeholder) replacement fields and the arguments of `format()`. This code is discussed further following the listing.

Listing 9.30 Demonstration of format strings where the replacement fields provide only placement information (not formatting information).

```
1 >>> "Hello {}!".format("Jack")
2 'Hello Jack!'
```

[8] Complete details can be found at **docs.python.org/py3k/library/string.html#formatstrings**.

```
3 >>> "Hello {} and {}!".format("Jack", "Jill")
4 'Hello Jack and Jill!'
5 >>> "Float: {}, List: {}".format(1 / 7, [1, 2, 3])
6 'Float: 0.14285714285714285, List: [1, 2, 3]'
```

The string in line 1 contains braces and the `format()` method is invoked on this string. Thus, by definition, this is a *format string*. Within this string there is single replacement field between `Hello` and the exclamation point. The one argument of the `format()` method, i.e., the string `Jack`, is substituted into this placeholder. The resulting string is shown in line 2. In line 3, there are two replacement fields: one after `Hello` and one immediately before the exclamation mark. Correspondingly, `format()` has two arguments. The first argument is substituted into the first replacement field and the second argument is substituted into the second replacement field. The resulting string is shown in line 4.

Line 5 illustrates that replacement fields can serve as generic placeholders for any object, i.e., any object we provide as an argument to the `print()` function can be "mapped" to a replacement field. In line 5 `format()`'s first argument is an expression that evaluates to a `float` and the second is a three-element `list`. In the resulting string, shown in line 6, the braces are replaced by the `float` and `list` values.

Replacement fields can be much more flexible, powerful, and, well, complicated than these simple placeholders. Rather than using merely empty braces, information can be provided between the braces. We will consider two pieces of information: the field "name" and the format specifier.

A replacement field can contain a *field name*. Although the field name can be constructed in various ways, we will consider only a numeric "name." If an integer value is given as the field name, it specifies the argument of `format()` to which the replacement field corresponds. You can think of the "name" as an index and the arguments of `format()` as having indices starting from zero. Thus, a field name of `{0}` corresponds to the first argument of the `format()` method, a field name of `{1}` corresponds to the second argument, and so on. In this way `format()` doesn't have to have a separate argument for each replacement field. Listing 9.31 demonstrates this:

Listing 9.31 Demonstration of the use of field names in the replacement fields of a format string. Only numeric "names" are considered. The field name specifies the argument of the `format()` method to which the replacement field corresponds.

```
1 >>> "Hello {0} and {1}.  Or, is it {1} and {0}?".format("Jack", "Jill")
2 'Hello Jack and Jill.  Or, is it Jill and Jack?'
3 >>> "Hello {0} {0} {0} {0}.".format("Major")
4 'Hello Major Major Major Major.'
5 >>> "Hello {} {} {} {}.".format("Major")
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 IndexError: tuple index out of range
```

In line 1 there are four replacement fields in the format string, but the `format()` method has only two arguments. Note that these replacement fields are still simply placeholders—they do

not provide any formatting information beyond location. Since these replacement fields now have field names, they specify the arguments of the `format()` method for which they are placeholders. Again, a replacement field of `{0}` specifies that the first argument of `format()` should appear in this location, while `{1}` specifies the second argument should replace the replacement field. Note the resulting string in line 2 and how `Jack` and `Jill` appear twice (and in different order). In the format string in line 3 there are four replacement fields but there is only one argument in the `format()` method. Because each of the replacement fields is `{0}`, the first (and only) argument of the `format()` method appears four times in the resulting string in line 4.[9]

The expression in line 5 also has a format string with four replacement fields and the `format()` method again has a single argument. Here, however, the replacement fields do not provide field names and, as indicated by the resulting exception shown in lines 6 through 8, Python cannot determine how to match the replacement fields to `format()`'s argument. (In this particular case the error message itself is not especially useful in describing the root of the problem.)

Listing 9.32 gives another example in which the format string simply provides placeholder information (and the replacement fields are unnamed). This code further demonstrates that objects with different types can be used with format strings. Additionally, this code illustrates that the `format()` method returns a string (and it does not differ from any other string—it can be used in any way one would normally use a string).

Listing 9.32 The arguments of the `format()` method can be variables, literals, or other expressions. This code also demonstrates that the return value of the `format()` method is simply a string.

```
1  >>> opponent = "Eagles"
2  >>> score_us = 17
3  >>> score_them = 22
4  >>> "Seahawks {}, {} {}".format(score_us, opponent, score_them)
5  'Seahawks 17, Eagles 22'
6  >>> final = "Seahawks {}, {} {}".format(score_us + 7, opponent,
7  ... score_them)
8  >>> type(final)
9  <class 'str'>
10 >>> print(final)
11 Seahawks 24, Eagles 22
```

In line 4 `format()`'s arguments are all variables—the first is an integer, the second is a string, and the third is also an integer. As you would expect, these arguments must all be separated by commas. When it comes to replacement fields within the format string, there is no restriction placed on what comes between the fields. In line 4 there is a comma between the first and second replacement field, but this is a literal comma—this indicates a comma should appear in the resulting string at this location. Line 5 shows the resulting string. In line 6, `format()`'s first argument is an integer expression and the other two arguments remain unchanged. The resulting string is assigned to the variable `final`. After confirming `final`'s type is `str` (lines 8 and 9), the `print()` statement in line 10 displays the `final` string.

[9] If you haven't read Joseph Heller's *Catch-22*, you should.

9.7.2 Format Specifier: Width

The contents of a replacement field can also provide instructions about the formatting of output. We will consider several of the myriad ways in which the output can be formatted. Section 9.7.1 explains that replacement fields can be named. To provide formatting information, we must provide a *format specifier* within the replacement field. The format specifier must be preceded by a colon. The field name, if present, appears to the left of this colon. Thus, we can think of a general replacement field as consisting of

```
<replacement_field> = {<field_name>:<format_specifier>}
```

As both the field name and format specifier are optional, alternate forms of replacement fields are

```
<replacement_field> = {}
```

and

```
<replacement_field> = {<field_name>}
```

and

```
<replacement_field> = {:<format_specifier>}
```

Perhaps the simplest formatting information is the (minimum) width of the resulting field. Thus, for a particular replacement field, this is the number of spaces provided to hold the string representation of the corresponding object given as an argument to `format()`. If the number of spaces is insufficient to display the given object, Python will use as many additional spaces as necessary. If the object requires fewer characters than the given width, the "unused" spaces are, by default, filled with blank spaces. The code in Listing 9.33 demonstrates various aspects of the width specifier.

Listing 9.33 Demonstration of the width specifier. If the string representation of the corresponding object is longer than the width, the output will exceed the width as necessary. If the object is shorter than the width, blank spaces are used to fill the field.

```
1  >>> "{0:5}, {1:5}, and {2:5}!".format("Red", "White", "Periwinkle")
2  'Red  , White, and Periwinkle!'
3  >>> "{0:5}, {1:5}, and {2:5}!".format(5, 10, 15)
4  '    5,    10, and    15!'
5  >>> # "Name" portion of replacement field is unnecessary when there is
6  >>> # a one-to-one and in-order correspondence between the fields and
7  >>> # arguments of format().
8  >>> "{:5}, {:5}, and {:5}!".format(5, 10, 15)
9  '    5,    10, and    15!'
10 >>> # Colon in replacement field is not an operator.  Cannot have
11 >>> # spaces between the field name and colon.
12 >>> "{0 : 5}, {1 : 5}, and {2 : 7}!".format(5, 10, 15)
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in <module>
15 KeyError: '0 '
```

In line 1 there are three replacement fields, each of which has a width of 5. The argument corresponding to the first field (Red) has a length of 3, i.e., two characters less than the width of the field. Thus, in line 2, there are two spaces between Red and the comma that follows the field. Note that Red has been left-justified in the field. The argument corresponding to the second replacement field (White) has a length of 5 and hence in line 2 there is no space between White and the comma that follows the field. The argument corresponding to the third replacement field has a length of 9, i.e., four characters longer than the width of the field. In cases such as this the field simply grows to whatever length is necessary to accommodate the string. (When no width is given, the field is always as wide as the string representation of the object.)

In line 3 of Listing 9.33 the same format string is used as in line 1, but now the arguments of the format() method are integers. The output in line 4 shows that numbers are right justified within a field. Line 8 is similar to line 3 except the field names have been dropped (i.e., the portion of the replacement field to the left of the colon). A field name is unnecessary when there is a one-to-one and in-order correspondence between the replacement fields and the arguments of the format() method.

To enhance readability in this textbook, we typically surround an operator by spaces. However, the colon in a replacement field is not an operator and, as of Python 3.2.2, there cannot be space between the field name and the colon.[10] This is illustrated in lines 12 through 15 where a space preceding the colon results in an error. (Again, the error message is not especially useful in pointing out the cause of the error.)

9.7.3 Format Specifier: Alignment

Listing 9.33 shows that, by default, strings are left-justified while numbers are right-justified within their fields, but we can override this default behavior. We can explicitly control the alignment by providing one of four characters prior to the width: < for left justification, ^ for centering, > for right justification, and = to left-justify the sign of a number while right-justifying its magnitude. This is demonstrated in Listing 9.34. The same format string is used in both lines 1 and 3. This string has three replacement fields. The first specifies left justification, the second specifies centering, and the third specifies right justification. The output in lines 2 and 4 demonstrates the desired behavior. The expressions in lines 5 and 7 use the = alignment character. By default the plus sign is not displayed for positive numbers.

Listing 9.34 Alignment of output within a field can be controlled with the aligned characters < (left), ^ (centering), and > (right).

```
1 >>> print("|{:<7}||{:^7}||{:>7}|".format("bat", "cat", "dat"))
2 |bat    ||  cat  ||    dat|
3 >>> print("|{:<7}||{:^7}||{:>7}|".format(123, 234, 345))
4 |123    ||  234  ||    345|
5 >>> "{:=7}".format(-123)
6 '-   123'
7 >>> "{:=7}".format(123)
```

[10] A space is permitted following the colon but, as described in 9.7.4, is interpreted as the "fill character" and can affect formatting of the output.

```
8 '    123'
```

9.7.4 Format Specifier: Fill and Zero Padding

One can also specify a "fill character." If an object does not fill the allocated space of the replacement field, the default "fill character" is a blank space. However, this can be overridden by providing an additional character between the colon and the alignment character, i.e., at the start of the format specifier.[11] This is demonstrated in Listing 9.35. In line 1 the fill characters for the three fields are <, *, and =. In line 3 the fill characters are >, ^, and 0. The last fill character is of particular interest for reasons that will become clear shortly. The discussion of this code continues following the listing.

Listing 9.35 A "fill character" can be specified between the colon and alignment character. This character will be repeated as necessary to fill any spaces that would otherwise be blank.

```
1  >>> print("|{:<<7}||{:*^7}||{:=>7}|".format("bat", "cat", "dat"))
2  |bat<<<<||**cat**||====dat|
3  >>> print("|{:><7}||{:^^7}||{:0>7}|".format(123, 234, 345))
4  |123>>>>||^^234^^||0000345|
5  >>> "{:07} {:07} {:07}".format(345, 2.5, -123)
6  '0000345 00002.5 -000123'
7  >>> "{:0=7} {:0=7} {:0>7}".format(2.5, -123, -123)
8  '00002.5 -000123 000-123'
9  >>> "{:07}".format("mat")
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 ValueError: '=' alignment not allowed in string format specifier
13 >>> "{:0>7}".format("mat")
14 '0000mat'
```

When the argument of the `format()` method is a numeric value, a zero (`0`) can precede the width specifier. This indicates the field should be "zero padded," i.e., any unused spaces should be filled with zeroes, but the padding should be done between the sign of the number and its magnitude. In fact, a zero before the width specifier is translated to a fill and alignment specifier of `0=`. This is demonstrated in lines 5 through 8 of Listing 9.35. In line 5 the arguments of `format()` are an integer, a `float`, and a negative integer. For each of these the format specifier is simply `07` and the output on line 6 shows the resulting zero-padding. In line 7 the arguments of `format()` are a `float` and two negative integers. The first two replacement fields use `0=7` as the format specifier. The resulting output on line 8 indicates the equivalence of `07` and `0=7`. However, the third field in line 7 uses `0>7` as the format specifier. This is *not* equivalent to `07` (or `0=7`) in that the sign of the number is now adjacent to the magnitude rather than on the left side of the field.

[11] One restriction is that the fill character cannot be a closing brace (}).

Lines 9 through 12 of Listing 9.35 indicate that zero-padding cannot be used with a string argument. The error message may appear slightly cryptic until one realizes that zero-padding is translated into alignment with the = alignment character. Note that, as shown in lines 13 and 14, a format specifier of `:0>7` does not produce an error with a string argument and does provide a type of zero-padding. But, really this is just right justification of the string with a fill character of `0`.

9.7.5 Format Specifier: Precision (Maximum Width)

Section 9.7.2 describes how an integer is used to specify the (minimum) width of a field. Somewhat related to this is the *precision* specifier. This is also an integer, but unlike the width specifier, the precision specifier is preceded by a dot. Precision has different meanings depending on the type of the object being formatted. If the object is a `float`, the precision dictates the number of digits to display. If the object is a string, the precision indicates the maximum number of characters allowed to represent the string. (One cannot specify a precision for an integer argument.) As will be shown in a moment, when a width is specified, it can be less than, equal to, or greater than the precision. Keep in mind that the width specifies the size of the field while the precision more closely governs the characters used to format a given object.

Listing 9.36 demonstrate how the precision specifier affects the output. Line 1 defines a `float` with several non-zero digits while line 2 defines a `float` with only three non-zero digits. Line 3 defines a string with seven characters. Lines 5 and 6 show the default formatting of these variables. The output in line 6 corresponds to the output obtained when we write `print(x, y, s)` (apart from the leading and trailing quotation marks identifying this as a string). The discussion continues following the listing.

Listing 9.36 Use of the precision specifier to control the number of digits or characters displayed.

```
1  >>> x = 1 / 7        # float with lots of digits.
2  >>> y = 12.5         # float with only three digits.
3  >>> s = "stringy"    # String with seven characters.
4  >>> # Use default formatting.
5  >>> "{}, {}, {}".format(x, y, s)
6  '0.14285714285714285, 12.5, stringy'
7  >>> # Specify a precision of 5.
8  >>> "{:.5}, {:.5}, {:.5}".format(x, y, s)
9  '0.14286, 12.5, strin'
10 >>> # Specify a width of 10 and a precision of 5.
11 >>> "{:10.5}, {:10.5}, {:10.5}".format(x, y, s)
12 '   0.14286,       12.5, strin     '
13 >>> # Use alternate numeric formatting for second term.
14 >>> "{:10.5}, {:#10.5}, {:10.5}".format(x + 100, y, s)
15 '    100.14,     12.500, strin     '
```

In line 8 each of the format specifiers simply specifies a precision of `5`. Thus, in line 9, we see five digits of precision for the first `float` (the zero to the left of the decimal point is not considered a

significant digit). The second `float` is displayed with all its digits (i.e., all three of them). The string is now truncated to five characters. In line 11 both a width and a precision are given. The width is `10` while the precision is again `5`. In this case the "visible" output in line 12 is no different from that of line 9. However, these characters now appear in fields of width `10` (that are filled with spaces as appropriate). In line 14 the value of the first argument is increased by `100` while the format specifier of the second argument now has the hash symbol (`#`) preceding the width. In this context the hash symbol means to use an "alternate" form of numeric output. For a `float` this translates to showing any trailing zeros for the given precision. Notice that in line 15 the first term still has five digits of precision: three before the decimal point and two following it.

As we will see in the next section, the meaning of precision changes slightly for `floats` depending on the *type specifier*.

9.7.6 Format Specifier: Type

Finally, the format specifier may be terminated by a character that is the *type specifier*. The type specifier is primarily used to control the appearance of integers or `floats`. There are some type specifiers that can only be applied to integers. One of these is `b` which dictates using the binary representation of the number rather than the usual decimal representation. Two other type specifiers that pertain only to integers are `d` and `c`. A decimal representation is obtained with the `d` type specifier, but this is the default for integers and thus can be omitted if decimal output is desired. The type specifier `c` indicates that the value should be displayed as a character (as if using `chr()`).

The type specifiers that are nominally for `floats` also work for integer values. These type specifiers include `f` for fixed-point representation, `e` for exponential representation (with one digit to the left of the decimal point), and `g` for a "general format" that typically tries to format the number in the "nicest" way. If the value being formatted is a `float` and no type specifier is provided, then the output is similar to `g` but with at least one digit beyond the decimal point (by default `g` will not print the decimal point or a trailing digit if the value is a whole number).

The use of type specifiers is demonstrated in Listing 9.37. The code is discussed following the listing.

Listing 9.37 Use of various type specifiers applied to an integer and two `floats`.

```
1 >>> # b, c, d, f, e, and g type specifiers with an integer value.
2 >>> "{0:b}, {0:c}, {0:d}, {0:f}, {0:e}, {0:g}".format(65)
3 '1000001, A, 65, 65.000000, 6.500000e+01, 65'
4 >>> # f, e, and g with a float value with zero fractional part.
5 >>> "{0:f}, {0:e}, {0:g}".format(65.0)
6 '65.000000, 6.500000e+01, 65'
7 >>> # f, e, and g with a float value with non-zero fractional part.
8 >>> "{0:f}, {0:e}, {0:g}".format(65.12345)
9 '65.123450, 6.512345e+01, 65.1235'
```

In line 2 the argument of `format()` is the integer `65`. The subsequent output in line 3 starts with `1000001` which is the binary equivalent of `65`. We next see `A`, the ASCII character corresponding

to 65. The remainder of the line gives the number 65 formatted in accordance with the d, f, e, and g type specifiers.

In line 5 the f, e, and g specifiers are used with the float value 65.0, i.e., a float with a fractional part of zero. Note that the output produced with the g type specifier lacks the trailing decimal point and 0 that we typically expect to see for a whole-number float value. (It is an error to use any of the integer type specifiers with a float value, e.g., b, c, and d cannot be used with a float value.) Line 8 shows the result of using these type specifiers with a float that has a non-zero fractional part.

We can combine type specifiers with any of the previous format specifiers we have discussed. Listing 9.38 is an example where the alignment, width, and precision are provided together with a type specifier. In the first format specifier the alignment character indicates right alignment, but since this is the default for numeric values, this character can be omitted. This code demonstrates the changing interpretation of the precision caused by use of different type specifiers. The first two values in line 2 show that the precision specifies the number of digits to the right of the decimal sign for f and e specifiers, but the third values shows that it specifies the total number of digits for the g specifier.

Listing 9.38 The intepretation of precision depends on the type specifier. For e and f the precision is the number of digits to the right of the decimal sign. For g the precision corresponds to the total number of digits.

```
1 >>> "{0:>10.3f}, {0:<10.3e}, {0:^10.3g}".format(65.12345)
2 '    65.123, 6.512e+01 ,    65.1   '
```

9.7.7 Format Specifier: Summary

To summarize, a format specifier starts with a colon and then may contain any of the terms shown in brackets in the following (each of the terms is optional):

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

A brief description of the terms is provided below. We note that not all of these terms have been discussed in the preceding material. The interested reader is encouraged either to read the formatting information available online[12] or simply to enter these in a format string and observe the resulting output.

fill: Fill character (may be any character other than }). When given, the fill character must be followed by an alignment character.

align: < (left), ^ (center), > (right), or = (for numeric values left-justify sign and right-justify magnitude).

[12] docs.python.org/py3k/library/string.html#formatstrings.

sign: + (explicitly show positive and negative signs), − (show only negative signs; default), or <*space*> (leading space for positive numbers, negative sign for negative numbers).

#: Use alternate form of numeric output.

0: Use zero padding (equivalent to fill and alignment of `0=`).

width: Minimum width of the field (integer).

,: Show numeric values in groups of three, e.g., `1,000,000`.

.precision: Maximum number of characters for strings (integer); number of digits of precision for `floats`. For `f`, `F`, `e`, and `E` type specifiers this is the number of digits to the right of the decimal point. For `g`, `G`, and <*none*> type specifiers the precision is the total number of digits.

type: Integers: `b` (binary), `c` (convert to character; equivalent to using `chr()`), `d` (decimal; default), `o` (octal), `x` or `X` (hexadecimal with lowercase or uppercase letters), `n` (same as `d` but with "local" version of the `,` grouping symbol). `floats`: `e` or `E` (exponential notation), `f` or `F` (fixed point), `g` or `G` (general format), `n` (same as `g` but with "local" version of `,` grouping symbol), `%` (multiply value by 100 and display using `f`), <*none*> (similar to `g` but with at least one digit beyond the decimal point). Strings: `s` (default).

9.7.8 A Formatting Example

Assume we must display a time in terms of minutes and seconds down to a tenth of a second. (Here we are thinking of "time" as a duration, as in the time it took a person to complete a race. We are not thinking in terms of time of day.) In general, times are displayed with the minutes and seconds separated by a colon. For example, 19:34.7 would be 19 minutes and 34.7 seconds.

Assume we have a variable `mm` that represents the number of minutes and a variable `ss` that represents the number of seconds. The question now is: How can we put these together so that the result has the "standard" form of MM:SS.s? The code in Listing 9.39 shows an attempt in line 3 to construct suitable output for the given minutes and seconds, but this fails because of the spaces surrounding the colon. In line 5 a seemingly successful attempt is realized using a statement similar to the one in line 3 but setting `sep` to the empty string. Line 7 also appears to yield the desired output by converting the minutes and seconds to strings and then concatenating these together with a colon. But, are the statements in lines 5 and 7 suitable for the general case?

Listing 9.39 Attempt to construct a "standard" representation of a time to the nearest tenth of a second.

```
1 >>> mm = 19       # Minutes.
```

```
2 >>> ss = 34.7    # Seconds.
3 >>> print(mm, ":", ss)           # Fails because of spurious space.
4 19 : 34.7
5 >>> print(mm, ":", ss, sep="")    # Output appears as desired.
6 19:34.7
7 >>> print(str(mm) + ":" + str(ss)) # Output appears as desired.
8 19:34.7
```

Let's consider some other values for the seconds but continue to use the `print()` statement from line 7 of Listing 9.39. In line 2 of Listing 9.40 the seconds are set to an integer value. The subsequent time in line 4 is flawed in that it does not display the desired tenths of a second. In line 5 the seconds are set to a value that is less than ten. In this case the output, shown in line 7 is again flawed in that two digits should be used to display the seconds (i.e., the desired format is MM:SS.s and MM:S.s is not considered acceptable). Finally, in line 8 the seconds are set to a value with many digits of precision. The output in line 10 is once again not what is desired in that too many digits are displayed—we only want the seconds to the nearest tenth.

Listing 9.40 Demonstration that the desired format of MM:SS.s is not obtained for the given values of seconds when the `print()` statement from line 7 of Listing 9.39 is used.

```
1 >>> mm = 19
2 >>> ss = 14
3 >>> print(str(mm) + ":" + str(ss))  # Output lacks tenths of second.
4 19:14
5 >>> ss = 7.3
6 >>> print(str(mm) + ":" + str(ss))  # Output missing a digit.
7 19:7.3
8 >>> ss = 34.7654321
9 >>> print(str(mm) + ":" + str(ss))  # Too many digits in output.
10 19:34.7654321
```

At this point you probably realize that a general solution for these different values for seconds is achieved using a format string with suitable format specifiers. We assume that the minutes are given as an integer value and set the width for the minutes to `2`, but we know the output will increase appropriately if the number of minutes exceeds two digits. For the seconds, we want a fixed-point (`float`) representation with the width of the field equal to `4` and one digit of precision (one digit to the right of the decimal point), and zero-padding is used if the seconds value is less than ten. Listing 9.41 demonstrates that the desired output is obtained for all the values of seconds used in Listing 9.40.

Listing 9.41 Use of a format string to obtain the desired "standard" representation of time.

```
1 >>> mm = 19
2 >>> ss = 14
3 >>> "{0:2d}:{1:04.1f}".format(mm, ss)
```

```
'19:14.0'
>>> ss = 7.3
>>> "{0:2d}:{1:04.1f}".format(mm, ss)
'19:07.3'
>>> ss = 34.7654321
>>> "{0:2d}:{1:04.1f}".format(mm, ss)
'19:34.8'
```

9.8 Chapter Summary

Strings are *immutable*, i.e., they cannot be changed (although an identifier assigned to a string variable can be assigned to a new string).

Strings can be concatenated using the plus operator. With operator overloading, a string can be repeated by multiplying it by an integer.

Indexing and slicing of strings is the same as for `lists` and `tuples` (but working with characters rather than elements).

The **`len()`** function returns the number of characters in its string argument.

The **`str()`** function returns the string representation of its argument.

ASCII provides a mapping of characters to numeric values. There are a total of 128 ASCII characters, 95 of which are printable (or graphic) characters.

The **`ord()`** function returns the numeric value of its character argument. The **`chr()`** function returns the character for its numeric argument. `ord()` and `chr()` are inverses.

An *escape sequence* within a string begins with the backslash character which alters the usual meaning of the adjacent character or characters. For example, `'\n'` is the escape sequence for the newline character.

Strings have many methods including the following, where "a given string" means the string on which the method is invoked:

- **`split()`**: Returns a `list` of elements obtained by splitting the string apart at the specified substring argument. Default is to split on whitespace.
- **`join()`**: Concatenates the (string) elements of the `list` argument with a given string inserted between the elements. The insertion may be any string including an empty string.
- **`capitalize()`**, **`title()`**, **`lower()`**, **`upper()`**, **`swapcase()`**: Case methods that return a new string with the case set appropriately.
- **`count()`**: Returns the number of times the substring argument occurs in a given string.
- **`find()`** and **`index()`**: Return the index at which the substring argument occurs within a given string. Optional arguments can be used to specify the range of the search. `find()` returns `-1` if the substring is not found while `index()` raises an exception if the substring is not found.
- **`lstrip()`**, **`rstrip()`**, **`strip()`**: Strip whitespace from a given string (from the left, right, or from both ends, respectively).

- **replace()**: Replaces an "old" substring with a "new" substring in a given string.
- **__repr__()**: Returns a string that shows the "official" representation of a given string (useful for debugging purposes when not in an interactive environment).
- **format()**: Allows fine-grain control over the appearance of an object within a string. Used for formatting output.

9.9 Review Questions

1. What is printed by the following Python fragment?

```
s = "Jane Doe"
print(s[1])
```

   (a) J
   (b) e
   (c) Jane
   (d) a

2. What is printed by the following Python fragment?

```
s = "Jane Doe"
print(s[-1])
```

   (a) J
   (b) e
   (c) Jane
   (d) a

3. What is printed by the following Python fragment?

```
s = "Jane Doe"
print(s[1:3])
```

   (a) Ja
   (b) Jan
   (c) an
   (d) ane

4. What is the output from the following program, if the input is `Spam And Eggs`?

```
def main():
    msg = input("Enter a phrase: ")
    for w in msg.split():
        print(w[0], end="")

main()
```

(a) `SAE`

(b) `S A E`

(c) `S S S`

(d) `Spam And Eggs`

(e) None of the above.

5. What is the output of this program fragment?

```
for x in "Mississippi".split("i"):
    print(x, end="")
```

(a) `Msssspp`

(b) `M ssissippi`

(c) `Mi ssi ssi ppi`

(d) `M ss ss pp`

6. ASCII is

(a) a standardized encoding of written characters as numeric codes.

(b) an encryption system for keeping information private.

(c) a way of representing numbers using binary.

(d) computer language used in natural language processing.

7. What function can be used to get the ASCII value of a given character?

(a) `str()`

(b) `ord()`

(c) `chr()`

(d) `ascii()`

(e) None of the above.

8. What is output produced by the following?

```
s = "absense makes the brain shrink"
x = s.find("s")
y = s.find("s", x + 1)
print(s[x : y])
```

(a) `sens`

(b) `ens`

(c) `en`

(d) `sen`

9. One difference between strings and lists in Python is that

   (a) strings are sequences, but lists aren't.

   (b) lists can be indexed and sliced, but strings can't.

   (c) lists are mutable (changeable), but strings immutable (unchangeable).

   (d) strings can be concatenated, but lists can't.

10. What is an appropriate `for`-loop for writing the characters of the string `s`, one character per line?

    (a)
    ```
    for ch in s:
        print(ch)
    ```

    (b)
    ```
    for i in range(len(s)):
        print(s[i])
    ```

    (c) Neither of the above.

    (d) Both of the above.

11. The following program fragment is meant to be used to find the sum of the ASCII values for all the characters in a string that the user enters. What is the missing line in this code?

```
phrase = input("Enter a phrase: ")
ascii_sum = 0        # accumulator for the sum
for ch in phrase:
    ##### missing line here
print(ascii_sum)
```

   (a) `ascii_sum = ascii_sum + ch`

   (b) `ascii_sum = chr(ch)`

   (c) `ascii_sum = ascii_sum + chr(ch)`

   (d) `ascii_sum = ascii_sum + ord(ch)`

12. What is the result of evaluating the expression `chr(ord('A') + 2)`?

    (a) `'A2'`
    (b) `'C'`
    (c) `67`
    (d) An error.
    (e) None of the above.

13. What is the output of the following code?

```
s0 = "A Toyota"
s1 = ""
for ch in s0:
    s1 = ch + s1

print(s1)
```

    (a) `A Toyota`
    (b) `atoyoT A`
    (c) None of the above.

14. What is the output of the following code?

```
s0 = "A Toyota"
s1 = ""
for ch in s0[ : : -1]:
    s1 = ch + s1

print(s1)
```

    (a) `A Toyota`
    (b) `atoyoT A`
    (c) None of the above.

15. What is the output of the following code?

```
s0 = "A Toyota"
s1 = ""
for ch in s0[-1 : 0 : -1]:
    s1 = s1 + ch

print(s1)
```

(a) `A Toyota`

(b) `atoyoT A`

(c) None of the above.

16. What is the value of `z` after the following has been executed:

```
s = ''
for i in range(-1, 2):
    s = s + str(i)

z = int(s)
```

(a) `0`

(b) `2`

(c) `-1012`

(d) `-101`

(e) This code produces an error.

17. What is the value of `ch` after the following has been executed?

```
ch = 'A'
ch_ascii = ord(ch)
ch = chr(ch_ascii + 2)
```

(a) `'A'`

(b) `67`

(c) `'C'`

(d) This code produces an error.

18. What is the output produced by the `print()` statement in the following code?

```
s1 = "I'd rather a bottle in front of me than a frontal lobotomy."
s2 = s1.split()
print(s2[2])
```

(a) `'`

(b) `d`

(c) `rather`

(d) `a`

(e) `bottle`

19. What is the output produced by the `print()` statement in the following code?

```
s1 = "I'd\nrather a bottle in front of me than a frontal lobotomy."
s2 = s1.split()
print(s2[2])
```

(a) `'`

(b) `d`

(c) `rather`

(d) `a`

(e) `bottle`

(f) None of the above.

20. The variable `s` contains the string `'cougars'`. A programmer wants to change this variable so that it is assigned the string `'Cougars'`. Which of the following will accomplish this?

(a)
```
s.upper()
```

(b)
```
s[0] = 'C'
```

(c)
```
s = 'C' + s[1 : len(s)]
```

(d)
```
s.capitalize()
```

(e) All of the above.

(f) None of the above.

21. What output is produced by the following code?

```
1 s = "Jane Doe"
2 print(s[1 : 3: -1])
```

(a) `aJ`

(b) `naJ`

(c) `na`

(d) `en`

(e) None of the above.

22. After the following commands have been executed, what is the value of `x`?

```
s = "this is a test"
x = s.split()
```

23. After the following commands have been executed, what is the value of `y`?

```
s = "this is a test"
y = s.split("s")
```

24. Recall that the `str()` function returns the string equivalent of its argument. What is the output produced by the following:

```
a = 123456
s = str(a)
print(s[5] + s[4] + s[3] + s[2])
```

25. What is the value of `count` after the following code has been executed?

```
s = "He said he saw Henry."
count = s.count("he")
```

    (a) `0`
    (b) `1`
    (c) `2`
    (d) `3`
    (e) None of the above.

26. What is the value of `s2` after the following has been executed?

```
s1 = "Grok!"
s2 = s1[ : -2] + "w."
```

    (a) `Grow.`
    (b) `kw.`
    (c) `k!w`
    (d) None of the above.

27. What is the value of `s2` after the following has been executed?

```
s1 = "Grok!"
s2 = s1[-2] + "w."
```

    (a) `Grow.`
    (b) `kw.`
    (c) `k!w`
    (d) None of the above.

28. What is the value of `s2` after the following has been executed?

```
s1 = "Grok!"
s2 = s1[-2 : ] + "w."
```

   (a) `kw.`
   (b) `Grow.`
   (c) `k!w`
   (d) None of the above.

ANSWERS: 1) d; 2) b; 3) c; 4) a; 5) a; 6) a; 7) b; 8) d; 9) c; 10) d; 11) d; 12) b; 13) b; 14) a; 15) c; 16) d; 17) c; 18) d; 19) d; 20) c; 21) e; 22) `['this', 'is', 'a', 'test']`; 23) `['thi', ' i', ' a te', 't']`; 24) `6543`; 25) b; 26) a; 27) b; 28) d.

Chapter 10

Reading and Writing Files

Up to this point we have entered data into our programs either literally (i.e., hardwired into the code when the code is written) or obtained it from the user via an `input()` statement. However, we often need to work with data stored in a file. Furthermore, rather than displaying results on the screen, sometimes the desired action is to store the results in a file. In this chapter we explore the reading and writing of files. In general, the content of a file is in the form of (readable) text or in the form of binary or "raw" data. In this chapter we will only consider the reading and writing of text, i.e., files that consist of characters from the ASCII character set.

10.1 Reading a File

To open a file for reading we use the built-in function `open()`. The first argument is the name of the file and the second argument is the *mode*. When we want to read from an existing file, the mode is set to the character `'r'` (as opposed to the character `'w'` which indicates we want to write to a new file).[1] For the file to be opened successfully, it must be in one of the directories where Python searches, i.e., the file must be somewhere in Python's *path*. Controlling Python's path for the reading and writing of files is no different than controlling the path for importing modules. Thus, the discussion in Sec. 8.6 is directly relevant to the material in this chapter. As a reminder, perhaps the simplest way to ensure Python will find an existing file is to set the *current working directory* to the directory where the file resides. For example, assume there is a file called `info.txt`[2] in the `Documents` directory of a Macintosh or Linux machine or in the `My Documents` folder of a Windows machine.[3] The statements shown in Listing 10.1 are appropriate for opening this file for the user `guido`. The first two lines of both sets of instructions serve to set the current working directory to the desired location. Even if more than one file in this directory is opened, the first two statements are issued just once (however, there must be one `open()` statement for each file).

From the file: `files.tex`

[1] Actually `open()` can be used with a single argument—the file name—in which case it is understood that the file should be opened for reading, i.e., `'r'` is the default mode.

[2] There is no restriction on a file name. Often files will have an extension of `.txt` or `.dat`, but this is not necessary.

[3] We use the terms *folder* and *directory* interchangeably.

Listing 10.1 Demonstration of setting the current working directory and opening a file within this directory. It is assumed the user's account name is `guido` and the file system has been configured in a "typical" manner.

The following are appropriate for a Macintosh or Linux machine:

```
import os     # Import the "operating system" module os.
os.chdir("/Users/guido/Documents")
file = open("info.txt", "r")
```

Analogous statements on a Windows machine are:

```
import os     # Import the "operating system" module os.
os.chdir("C:/Users/guido/My Documents")
file = open("info.txt", "r")
```

The `open()` function returns a *file object*.[4] We will typically assign the file object to an identifier although, as will be shown, this isn't strictly necessary to access the contents of a file.

A list of a file object's methods is obtained by giving the object as the argument to `dir()`. This is done in Listing 10.2. In the subsequent output we observe the methods `read()`, `readline()`, and `readlines()`. Of some interest is the fact that, even though we have opened the file for reading, the object has methods `write()` and `writelines()`. We will consider the write-related methods in Sec. 10.2.

Listing 10.2 Methods for a file object. The methods discussed in this chapter are shown in slanted bold type.

```
>>> file = open("info.txt", "r")
>>> dir(file)
['_CHUNK_SIZE', '__class__', '__delattr__', '__doc__', '__enter__',
 '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__',
 '__getstate__', '__gt__', '__hash__', '__init__', '__iter__',
 '__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '_checkClosed', '_checkReadable',
 '_checkSeekable', '_checkWritable', 'buffer', 'close', 'closed',
 'detach', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
 'line_buffering', 'name', 'newlines', 'read', 'readable',
 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate',
 'writable', 'write', 'writelines']
```

Throughout this section we assume the existence of the file `info.txt` located in a directory in Python's path. The file is assumed to contain the following text:

```
This file contains some numbers (and text).
4,   5,   6,    12
```

[4]More technically, the object is a *stream* or a *file-like* object. However, if you issue the command `help(open)` the documentation states that `open()` returns a file object. On the other hand, if you use `type()` to check the type of one of these file objects, you get something that looks rather mysterious.

```
12.3 37.2 -15.7

This is the last line.
```

After opening a file, the contents can be obtained by invoking the `read()`, `readline()`, or `readlines()` methods (or a combination of these methods). `read()` and `readline()` both return a string. `readlines()` returns a list. Alternatively, we can also simply use the file object directly as the iterable in a `for`-loop. In the following sections we consider each of these ways of reading a file.

10.1.1 `read()`, `close()`, and `tell()`

When one uses the `read()` method, the entire file is read and its contents are returned as a single string. This is demonstrated in Listing 10.3. In line 1 the file is opened. In line 2 the `read()` method is called and the result is stored to the identifier `all`. Line 3 serves to echo the contents of `all` which we see in lines 4 and 5.[5] The output in lines 4 and 5 is not formatted, e.g., newlines are indicated by `\n`. To obtain output that mirrors the contents of the file in the way it is typically displayed, we can simply print this string as is done in line 6. Note that the `print()` statement has the optional argument `end` set to the empty string. This is done because the string `all` already contains all the newline characters that were contained in the file itself. Thus, because `all` ends with a newline character, if we do not set `end` to the empty string, there will be an additional blank line at the end of the output.

Listing 10.3 Use of the `read()` method to read an entire file as one string.

```
>>> file = open("info.txt", "r")   # Open file for reading.
>>> all = file.read()              # Read the entire file.
>>> all                            # Show the resulting string.
'This file contains some numbers (and text).\n4,   5,   6,    12\n
12.3 37.2 -15.7\n\nThis is the last line.\n'
>>> print(all, end="")             # Print the contents of the file.
This file contains some numbers (and text).
4,   5,   6,    12
12.3 37.2 -15.7

This is the last line.
```

After reading the contents of a file with `read()`, we can call `read()` again. *However,* rather than obtaining the entire contents of the file, we get an empty string. Python maintains a *stream position* that indicates where it is in terms of reading a file, i.e., the index of the next character to be read. When a file is first opened, this position is at the very start of the file. After invoking the `read()` method the position is at the end of the file, i.e., there are no more characters to read. If you invoke the `read()` method with the stream position at the end of the file, you merely obtain

[5] An explicit line break has been added for display purposes—Python tries to display the string on a single line but the output is "wrapped" to the following line at the edge of the window.

an empty string—`read()` does not automatically return to the start of the file. If you must read a file multiple times, you can *close* the file, using the `close()` method, and then open it again using `open()`. In fact, there is no need to `close()` the file before issuing the second `open()` because if a file is already open, Python will close it before opening it again. However, it is good practice to close files when you are done with them. When you `close()` a file that has been opened for reading, you free some resources and you ensure that accidental reads of a file do not occur. Listing 10.4 demonstrates reading from a file multiple times.

Listing 10.4 If the `read()` method is invoked more than once on a file, the method returns an empty string for all but the first invocation. To reposition the stream position back to the start of the file the simplest approach is to close the file and reopen it. It is an error to try to read from a closed file.

```
1  >>> file = open("info.txt", "r")  # Open file for reading.
2  >>> all = file.read()             # Read the entire file.
3  >>> everything = file.read()      # Attempt to read entire file again.
4  >>> everything                    # There is nothing in everything.
5  ''
6  >>> file.close()                  # Close file.
7  >>> everything = file.read()      # Attempt to read a closed file.
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 ValueError: I/O operation on closed file.
11 >>> file = open("info.txt", "r")  # Open file for reading (again).
12 >>> everything = file.read()      # Now we obtain everything.
13 >>> everything
14 'This file contains some numbers (and text).\n4,   5,   6,    12\n
15 12.3 37.2 -15.7\n\nThis is the last line.\n'
```

The first two lines of Listing 10.4 are the same as those of Listing 10.3, i.e., the file is opened and the entire file is read. In line 3 the `read()` method is again called. This time the return value is assigned to the identifier `everything`. Lines 4 and 5 show that, rather than containing a copy of the entire file, `everything` is the empty string. In line 6 the `close()` method is used to close the file. In line 7 we again attempt to use `read()` to read the file. However, it is an error to read from a closed file and hence a `ValueError` exception is raised as shown in lines 8 through 10. In line 11 the file is opened again. In line 12 the `read()` method is used to read the entire file. As shown in lines 14 through 15, `everything` now corresponds to the contents of the entire file.

Although it is generally something we ignore, it can be instructive to observe the stream position, i.e., the value that indicates the index of the next character to be read. As mentioned, when a file is first opened, the stream position is zero, i.e., the next character to be read is the first one in the file. Or, thought of another way, this zero indicates that we are offset zero from the start of the file. We can obtain the value of the stream position with the `tell()` method. This is demonstrated in Listing 10.5. The code is discussed following the listing.

Listing 10.5 The `tell()` method returns the current stream position, i.e., the index of the next character to be read.

```
1  >>> file = open("info.txt", "r")   # Open file for reading.
2  >>> file.tell()                    # Stream position at start of file.
3  0
4  >>> all = file.read()              # Read entire file.
5  >>> file.tell()                    # Stream position at end of file.
6  103
7  >>> # The following shows that, after reading the entire file, the
8  >>> # length of all the characters in the file and the value of the
9  >>> # stream position are the same.
10 >>> len(all)
11 103
```

In line 1 the file is opened for reading. In lines 2 and 3 `tell()` reports the stream position is `0`, i.e., we are at the start of the file. In line 4 the entire file is read using `read()` and stored as the string `all`. In lines 5 and 6 we see the stream position is now `103`. Checking the length of `all` in lines 10 and 11 we see that it is the same as the current stream position. (Recall that the last valid index of a sequence is one less than its length, i.e., the last valid index for `all` is `102`.)

10.1.2 `readline()`

Unlike `read()`, which reads the entire contents of a file in one shot, the `readline()` method reads a single line and returns this line as a string. This is illustrated in Listing 10.6 where the file `info.txt` is read one line at a time. To illustrate how Python keeps track of where it is within the file, the `tell()` method is also called to show the stream position. Typically there is no reason to display this information—it is sufficient that Python is keeping track of it. The code is discussed following the listing.

Listing 10.6 Demonstration of the use of `readline()` to read the contents of a file one line at a time. The `tell()` method is used merely to show how the stream position advances with each invocation of `readline()`.

```
1  >>> file = open("info.txt", "r")
2  >>> file.readline()        # Read one line.
3  'This file contains some numbers (and text).\n'
4  >>> file.tell()            # Check on stream position.
5  44
6  >>> file.readline()        # Read second line.
7  '4,    5,    6,     12\n'
8  >>> file.tell()
9  63
10 >>> file.readline()        # Read third line.
11 '12.3 37.2 -15.7\n'
12 >>> file.tell()
```

```
79
>>> file.readline()       # Fourth line is "blank" but not empty.
'\n'
>>> file.tell()
80
>>> file.readline()       # Read fifth line.
'This is the last line.\n'
>>> file.tell()
103
>>> # It is not an error to call readline() when at the end of the file.
>>> # Instead, readline merely returns an empty string.
>>> file.readline()
''
```

In line 1 the file is opened for reading. In line 2 the `readline()` method is invoked. This returns, as a string, the first line of the file. Note that the newline character is part of this string (i.e., the last character of the string). In line 4 the `tell()` method is called to check the current value of the stream position. Line 5 reports it is `44`. This corresponds to the number of characters in the first line of the file (the newline character counts as a single character). The next several lines repeat the process of calling `readline()` and then showing the stream position.

In line 15 of Listing 10.6, note the result of reading the fourth line of `info.txt` (i.e., note the result of the statement in line 14). The fourth line of `info.txt` is a "blank" line but, in fact, that does not mean it is empty. Within the file this line consists solely of the newline character. We see this in the output on line 15.

In line 24, we attempt to read a sixth line. Although there isn't a sixth line, this does not produce an error. Instead, the return value is the empty string. Thus, we know (and Python knows) that the end of the file has been reached when `readline()` returns an empty string.

Obviously it can be awkward if one has to explicitly call the `readline()` method for each individual line of a long file. Fortunately there are a couple other constructs that facilitate reading the entire file on a line-by-line basis.

10.1.3 readlines()

An alternative approach for conveniently reading the entire contents of a file is offered by the `readlines()` method. The `readlines()` method is somewhat like `read()` in that it reads the entire file.[6] However, rather than returning a single string, `readlines()` returns a `list` in which each element of the `list` is a line from the file. This is demonstrated in Listing 10.7 which is discussed following the listing.

Listing 10.7 Demonstration of the use of the `readlines()` method .

```
>>> file = open("info.txt", "r")
>>> lines = file.readlines()       # Read all lines into a list.
```

[6] Actually these methods read from the current stream position to the end of the file as discussed in more detail later.

```
3  >>> lines                       # Display list.
4  ['This file contains some numbers (and text).\n', '4,    5,    6,    12\n',
5   '12.3 37.2 -15.7\n', '\n', 'This is the last line.\n']
6  >>> count = 0
7  >>> for line in lines:     # Show lines together with a count.
8  ...     count = count + 1
9  ...     print(count, ": ", line, sep="", end="")
10 ...
11 1: This file contains some numbers (and text).
12 2: 4,    5,    6,    12
13 3: 12.3 37.2 -15.7
14 4:
15 5: This is the last line.
```

In line 1 the file is opened for reading. In line 2 the `readlines()` method is invoked. This returns a `list`, which is assigned to the identifier `lines`, in which each element corresponds to a line of the file. In this particular case the list `lines` contains all the lines of the file. The entire file is stored within the `list` because the stream position was `0` when the method was invoked, i.e., `readlines()` starts from the current location of the stream position and reads until the end of the file. This type of behavior (with the reading starting from the current stream position and continuing to the end of the file) also pertains to `read()` and is discussed further in Sec. 10.1.5. If the `readlines()` method is invoked when the stream position is at the end of a file, the method returns an empty `list`.

Line 3 echoes `lines`. The output on lines 4 and 5 shows that this `list` does indeed contain all the lines of the file. Note that each string in the `list` (i.e., each line) is terminated by a newline character. The code in lines 6 through 9 displays each line of the file with the line preceded by a line number (starting from one). The output is shown in lines 11 through 15. Note that the `print()` statement in line 9 has the optional argument `end` set to the empty string. If this were not done, each numbered line would be followed by a blank line since one newline is produced by the newline character contained within the string for the line and another newline generated by the `print()` function, thus resulting in a blank line. The optional argument `sep` is also set to the empty string so that there is no space between the count at the start of the line and the colon.

10.1.4 File Object Used as an Iterable

Rather than using `readlines()` to read the lines of file into a `list` and *then* using a `for`-loop to cycle through the elements of this `list`, one can use the file object itself as the iterable in the `for`-loop header. In this case the file will be read line-by-line, i.e., for each iteration of the `for`-loop the loop variable will be assigned a string corresponding to a line of the file. Lines are read, in order, from the current stream position until the end of the file.

In some applications input files are huge. It thus requires significant memory to read and store the entirety of these files. However, by using the file object as the `for`-loop iterable, only one line is read (and processed) at a time. In this way it is possible to process files that are tremendously large without using much memory.

The use of a file object as the iterable of a `for`-loop is illustrated in Listing 10.8.

Listing 10.8 Demonstration that a file object can be used as the iterable in the header of a `for`-loop. In this case the loop variable takes on the value of each line of the file.

```
>>> file = open("info.txt", "r")
>>> count = 0
>>> for line in file:
...     count = count + 1
...     print(count, ": ", line, sep="", end="")
...
1: This file contains some numbers (and text).
2: 4,   5,   6,    12
3: 12.3 37.2 -15.7
4:
5: This is the last line.
```

In this code the integer variable `count` is used to keep track of the line number. The `print()` statement in the `for`-loop prints both the line number and the corresponding line from the file. `end` is set to the empty string to suppress the newline generated by the `print()` function. Alternatively, rather than setting `end` to the empty string, to avoid repeating the newline character, we can strip the newline character from the `line` variable using the `rstrip()` method.

As a slight twist to the implementation in Listing 10.8, one may use the open statement directly as the iterable in the `for`-loop header. Thus, an alternate implementation of Listing 10.8 is given in Listing 10.9. Here, however, we made the further modification of enclosing the `open()` function in the `enumerate()` function. This allows us, in line 1 of Listing 10.9, to use simultaneous assignment to obtain both a "count" and a line from the file. However, the count starts from zero. Because we want the line number to start from one, we add `1` to `count` in line 2 in the first argument of the `print()` function.

Listing 10.9 An alternate implementation of 10.8 where the `open()` function is used as the iterable of the `for`-loop. In this implementation the `enumerate()` function is used to obtain the "count" directly, although we have to add one to this value to obtain the desired line number.

```
>>> for count, line in enumerate(open("info.txt", "r")):
...     print(count + 1, ": ", line, sep="", end="")
...
1: This file contains some numbers (and text).
2: 4,   5,   6,    12
3: 12.3 37.2 -15.7
4:
5: This is the last line.
```

Both Listing 10.8 and Listing 10.9 accomplish the same thing but five lines of code were required in Listing 10.8 whereas only two lines were needed in Listing 10.9. Reducing the number of lines of code is, in itself, neither a good thing nor a bad thing. Code should strive for both

efficiency and readability. In the case of Listings 10.8 and 10.9, efficiency is not an issue and the "readability" will partly be a function of the experience of the programmer. Listing 10.9 is rather "dense" and hence may be difficult for beginning Python programmers to understand. On the other hand, experienced Python programmers may prefer this density to the comparatively "verbose" code in Listing 10.8. Recognizing that readability is partially in the eye of the reader, we try to use common Python idioms in the listings, but often tend toward a verbose implementation when it is easier to understand than a more terse or dense implementation.

10.1.5 Using More than One Read Method

As mentioned in Sec. 10.1.3, when the `read()` or `readlines()` methods are called, the reading starts from the current stream position. To help illustrate this, consider a situation in which the first two lines of a file are comment lines or provide header information or simply need to be handled differently than the rest of the file. In this case these two lines could be read using the `readline()` method and then the remainder of the file could be conveniently read using either `readlines()` or `read()`.

The "mixing" of read methods is demonstrated in Listing 10.10. The file is opened in line 1. The first and second lines are read in lines 2 and 3, respectively. Then, in line 4, the `readlines()` method is invoked as the iterable of the `for`-loop. The strings returned by `readlines()` are assigned to the loop variable `line` and printed using the `print()` statement in line 5. The fact that there are only three lines of output (shown in lines 7 through 9) shows that `readlines()` starts reading the file from its third line.

Listing 10.10 The first two lines of a file are read using `readline()` and the remainder of the file is read using `readlines()`.

```
1 >>> file = open("info.txt", "r")
2 >>> line_one = file.readline()
3 >>> line_two = file.readline()
4 >>> for line in file.readlines():
5 ...     print(line, end="")
6 ...
7 12.3 37.2 -15.7
8
9 This is the last line.
```

10.2 Writing to a File

The converse of reading from a file is writing to a file. To do this, we again open the file with the `open()` function. As with opening a file for reading, the first argument is the file name. Now, however, the second argument (the mode) must be `'w'`. **Caution:** *When you open a file for writing, you destroy any previous file that existed with this file name in the current working directory.* Because `open()` can destroy existing files, any time you plan to open a file for writing

or reading, make doubly sure you have set the mode and the file name correctly. After opening the file we can use the `write()` or `writelines()` methods to write to it. Additionally, there is an optional argument for the `print()` function that allows us to specify a file to which the output can be written (i.e., rather than the output appearing on the screen, it is written to the file).

10.2.1 `write()` and `print()`

The `write()` method takes a single argument which is the string to be written to the file. The `write()` method is much less flexible than the `print()` function. With the `print()` function we can use any number of arguments and the arguments can be of any type. Furthermore, using the optional `sep` and `end` parameters, we can specify the separator to appear between objects and how the line should be terminated. In contrast, the `write()` method only accepts a single *string* as its argument (or, of course, any expression that returns a string). However, this is not quite as restrictive as it might first appear. Using string formatting, we can easily represent many objects as a single string. As an example of such a construction, first consider the code in Listing 10.11 which uses the `print()` function to display three variables.

Listing 10.11 Use of the `print()` function to display three values.

```
1 >>> a = 1.0
2 >>> b = "this"
3 >>> c = [12, -4]
4 >>> print(a, b, c)
5 1.0 this [12, -4]
```

The code in Listing 10.12 produces the same output as Listing 10.11, but in this case the output goes to the file `foo.dat`. In line 1 the file is opened and assigned to the identifier `file_out`. In line 5 the `write()` method is called with a single string argument. However, this string is the one produced by the `format()` method acting on the given format string which contains three replacement fields (see the discussion of replacement fields in Sec. 9.7). The resulting string is the same as representation of the three objects displayed in line 5 of Listing 10.11, i.e., after this code has been run, the file `foo.dat` will contain the same output as shown in line 5 of Listing 10.11. Line 7 of Listing 10.12 invokes the `close()` method on `file_out`. If you look at the contents of the file `foo.dat` before calling the `close()` method, you will probably *not* see the output generated by the `write()` method. The reason is that, in the interest of speed and efficiency, the output is *buffered*. Writing from internal computer memory to external memory (such as a disk drive) can be slow. The fewer times such an operation occurs, the better. When output is buffered, it is stored internally and not written to the file until there are "many" characters to write or until the file is closed.[7]

Listing 10.12 Use of the `write()` method to write a single string. Use of a format string and the `format()` method gives the same representation of the three objects as produced by the `print()` statement in Listing 10.11.

[7]Alternatively, there is a `flush()` method that forces the "flushing" of output from the buffer to the output file.

```
1 >>> file_out = open("foo.dat", "w")
2 >>> a = 1.0
3 >>> b = "this"
4 >>> c = [12, -4]
5 >>> file_out.write("{} {} {}\n".format(a, b, c))
6 18
7 >>> file_out.close()
```

The 18 that appears in line 6 is the return value of the write() method and corresponds to the number of characters written to the file. This number is shown in the interactive environment, but if the statement on line 5 appeared within a program (i.e., within a .py file), the return value would not be visible. A programmer is not obligated to use a return value and when a return value is not assigned to an identifier or printed, it simply "disappears."

Now, having discussed the write() method, we actually *can* use the print() function to write to a file! By adding an optional file=<*file_object*> argument to print()'s argument list, the output will be written to the file rather than to the screen.[8] Thus, the code in Listing 10.13 is completely equivalent to the code in Listing 10.12.

Listing 10.13 The print() function's optional file argument is used to produce the same result as in Listing 10.12. This code and the code of Listing 10.12 differ only in line 5.

```
1 >>> file_out = open("foo.dat", "w")
2 >>> a = 1.0
3 >>> b = "this"
4 >>> c = [12, -4]
5 >>> print(a, b, c, file=file_out)
6 >>> file_out.close()
```

As we are already familiar with the print() function, we will not consider it further in the remainder of this chapter.

As an example of the use of the write() method, let's read from one file and write its contents to another. Specifically, let's copy the contents of the file info.txt to a new file called info_lined.txt and insert the line number at the start of each line. This can be accomplished with the code shown in Listing 10.14. Lines 1 and 2 open the files. In line 3 the variable count is initialized to 0. The for-loop in lines 4 through 6 cycles through each line of the input file. Pay close attention to the argument of the write() method in line 6. The first replacement field in the format string specifies that two spaces should be allocated to the integer corresponding to the first argument of the format() method. The second replacement field merely serves as a placeholder for the second argument of the format() method, i.e., the string corresponding to a line of the input file. Note that this format string does not end with a newline character and the write method does not add a newline character. *However*, the line read from the input file does end with a newline character. Thus, one should not add another newline. The numerical values shown in lines 8 through 12 are the numbers of characters written on each line. (We can prevent these

[8] Or, more technically, the default output is directed to sys.stdout which is usually the screen.

values from being displayed simply by assigning the return value of the `write()` method to a variable. However, when this code is run from a `.py` file, these values are not displayed unless we explicitly do something to display them.)

Listing 10.14 Code to copy the contents of the file `info.txt` to the (new) file `info_lined.txt`.

```
>>> file_in = open("info.txt", "r")
>>> file_out = open("info_lined.txt", "w")
>>> count = 0
>>> for line in file_in:
...     count = count + 1
...     file_out.write("{:2d}: {}".format(count, line))
...
48
23
20
5
27
>>> file_out.close()
```

After running the code in Listing 10.14, the file `info_lined.txt` contains the following:

```
 1: This file contains some numbers (and text).
 2: 4,   5,   6,    12
 3: 12.3 37.2 -15.7
 4:
 5: This is the last line.
```

10.2.2 `writelines()`

The `writelines()` method takes a sequence as an argument, e.g., a tuple or a `list`. Each element of the sequence must be a string. In some sense `writelines()` is misnamed in that it doesn't necessarily write *lines*. Instead, it writes elements of a sequence (but a method name of `writeelementsofsequence()` isn't very appealing). If all the elements of the sequence end with the newline character, then the output will indeed be as if `writelines()` writes lines.

Consider the code shown in Listing 10.15 which creates two files, `out_1.txt` and `out_2.txt`. The `writelines()` method is used to write the `list values_1` to `out_1.txt` and the `tuple values_2` to `out_2.txt`. The significant difference between these two statements is not that one argument is a `tuple` and the other is a `list` (this distinction is of no concern to `writelines()`). Rather, it is that the strings in `values_2` are terminated with newline characters but the strings in `values_1` are not. The discussion continues following the listing.

Listing 10.15 Demonstration of the use of the `writelines()` method.

```
1 >>> out_1 = open("out_1.txt", "w")
2 >>> out_2 = open("out_2.txt", "w")
3 >>> values_1 = ["one", "two", "three"]
4 >>> values_2 = ("one\n", "two\n", "three\n")
5 >>> out_1.writelines(values_1)
6 >>> out_2.writelines(values_2)
7 >>> out_1.close()
8 >>> out_2.close()
```

After the code in Listing 10.15 is run, the file `out_1.txt` contains the following:

```
onetwothree
```

This text is not terminated by a newline character. On the other hand the file `out_2.txt` contains:

```
one
two
three
```

This text is terminated by a newline character.

10.3 Chapter Summary

Files are opened for reading or writing using the `open()` function. The first argument is the file name and the second is the mode (`'r'` for read, `'w'` for write). Returns a *file object*.

The `stream position` indicates the next character to be read from a file. When the file is first opened, the stream position is zero.

Contents of a file can be obtained using the following file-object methods:

- **`read()`**: Returns a string corresponding to the contents of a file from the current stream position to the end of the file.
- **`readline()`**: Returns, as a string, a single line from a file.
- **`readlines()`**: Returns a `list` of strings, one for each line of a file, from the current stream position to the end of the file (newline characters are not removed).

If the stream position is at the end of a file, `read()` and `readline()` return empty strings while `readlines()` returns an empty `list`.

A file object can be used as the iterable in a `for`-loop.

The **`close()`** method is used to close a file object. It is an error to read from a closed file.

The **`write()`** method can be used to write a string to a file.

A `print()` statement can also be used to print to a file (i.e., write a string to a file) using the optional `file` argument. A file object must be provided with this argument.

The **`writelines()`** method takes a sequence of strings as its argument and writes them to the given file as a continuous string.

10.4 Review Questions

1. Assume the file `input.txt` is opened successfully in the following. What is the type of the variable `z` after the following code is executed:

```
file = open("input.txt", "r")
z = file.readlines()
```

   (a) `list`
   (b) `str`
   (c) `file object`
   (d) None of the above.
   (e) This code produces an error.

For problems 2 through 9, assume the file `foo.txt` contains the following:

```
This is
a test.
Isn't it?  I think so.
```

2. What output is produced by the following?

```
file = open("foo.txt", "r")
s = file.read()
print(s[2])
```

3. What output is produced by the following?

```
file = open("foo.txt", "r")
file.readline()
print(file.readline(), end="")
```

4. What output is produced by the following?

```
file = open("foo.txt", "r")
for line in file:
    print(line[0], end="")
```

5. What output is produced by the following?

```
file = open("foo.txt", "r")
s = file.read()
xlist = s.split()
print(xlist[1])
```

6. What output is produced by the following?

```
file = open("foo.txt")
s = file.read()
xlist = s.split('\n')
print(xlist[1])
```

7. What output is produced by the following?

```
file = open("foo.txt")
xlist = file.readlines()
ylist = file.readlines()
print(len(xlist), len(ylist))
```

8. What output is produced by the following?

```
file = open("foo.txt")
xlist = file.readlines()
file.seek(0)
ylist = file.readlines()
print(len(ylist), len(ylist))
```

9. What output is produced by the following?

```
file = open("foo.txt")
for x in file.readline():
   print(x, end=":")
```

10. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
print(s, "is a test", file=file)
file.close()
```

11. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
file.write(s, "is a test")
file.close()
```

12. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
file.write(s + "is a test")
file.close()
```

13. What is contained in the file `out.txt` after executing the following?

```
file = open("out.txt", "w")
s = "this"
file.write("{} {}\n".format(s, "is a test"))
file.close()
```

ANSWERS: 1) a; 2) `i`; 3) `a test.`; 4) `TaI`; 5) `is`; 6) `a test.`; 7) `3 0`; 8) `3 3`; 9) `T:h:i:s:  :i:s:`; 10) `this is a test` (terminated with a newline character); 11) The file will be empty since there is an error that prevents the `write()` method from being used—`write()` takes a single string argument and here is is given two arguments.; 12) `thisis a test` (this is *not* terminated with a newline character); 13) `this is a test` (terminated with a newline character).

Chapter 11

Conditional Statements

So far we have written code that is executed sequentially, i.e., statements are executed in the order in which they are written. However there are times when we need to alter a program's flow of execution so that certain statements are only executed when a condition is met. To accomplish this we use *conditional statements*. For example, a conditional statement can be used to check whether a student's score on an exam is above a given value. If it is, perhaps a particular message is printed; if not, a different message is printed. As a second example, assume some operation must be performed on lines read from a file, but lines that start with a certain character should be ignored (perhaps this character indicates the line is a comment). In this case we use a conditional statement to check the first character of the line and have the program act appropriately.

The simplest form of a conditional statement is an `if` statement. As you will see, in an `if` statement a *test expression* is checked to see if it is "true." If it is, the body of the `if` statement is executed. Conversely, if the test expression is "false," the body of the `if` statement is not executed.

In this chapter we define what we mean by true and false in the context of conditional statements. We also explore many other programming elements related to conditional statements, including Boolean variables, comparison operators, logical operators, and `while`-loops.

11.1 `if` Statements, Boolean Variables, and `bool()`

George Boole was an English mathematician and logician who lived in the 1800's and formalized much of the math underlying digital computers. A *Boolean expression* is an expression in which the terms have one of only two states which we conveniently think of as true or false. Furthermore, a Boolean expression evaluates to only one of two states: true or false. Although you probably aren't acquainted with all the rigors of Boolean expressions, you are already quite familiar with them at some level because you frequently encounter them in your day-to-day life.

As an example of a Boolean expression, assume you are considering whether or not to go to a Saturday matinee. You might boil your decision down to the weather (you rather be outside when the weather is nice) *and* the movie's rating at **rottentomatoes.com** (perhaps you only go to movies that have a rating over 80 percent). We can think of your decision as being governed by a Boolean expression. The terms in the expression are "nice weather" and "rating over 80 percent" (again, in a Boolean expression these terms are either true or false). Ultimately you will see a

From the file: `conditionals.tex`

movie if the following Boolean expression evaluates to true and you will not see it if it evaluates to false: *NOT* nice weather *AND* rating over 80 percent. We will return to expressions involving operators such as *NOT* and *AND* in Sec. 11.4. First, however, we want to consider how the outcome of such expressions can control the flow of a program and consider what “true” and “false” are in terms of a Python program.

In honor of George Boole’s rigorous study of logical expressions, i.e., expressions in which terms can only be true or false, in many computer languages, including Python, there is a *Boolean data type*. In Python this type is identified as `bool` and there are two `bool` literals: `True` and `False`.[1]

We will return to Boolean variables shortly, but let us first consider `if` statements. The template for an `if` statement is shown in Listing 11.1. The statement consists of a header, shown in line 1, and a body that can contain any number of statements. The statements that constitute the body must be indented to the same level (similar to the bodies of `for`-loops and function definitions). Within the header is a `test_expression`. If the `test_expression` evaluates to `True`, the body is executed. Conversely, if the `test_expression` evaluates to `False`, the body is not executed and program execution continues with the statement following the body.

Listing 11.1 Template for an `if` statement.

```
1 if <test_expression>:
2     <body>
```

The `test_expression` in an `if` statement always evaluates to an object that is considered either `True` or `False`. There is, for instance, no possibility of something being considered “partially true.” It might seem that this places rather tight restrictions on what can serve as a valid `test_expression`. However, this is not the case. In fact, we can use any expression as the `test_expression`! But, let’s hold this thought and return to Boolean variables.

Listing 11.2 demonstrates the behavior of `if` statements. The `if` statement in lines 1 and 2 has a test expression that is simply the literal `True`. Since this expression simply evaluates to `True`, the body of the `if` statement is executed, resulting in the output shown in line 4. On the other hand, the `if` statement in lines 5 and 6 has a test expression consisting of the literal `False`. Since this evaluates to `False`, the body of the `if` statement, i.e., line 6 is not executed. Thus, in the interactive environment we simply get the interactive prompt back. The discussion of the code continues following the listing.

Listing 11.2 Demonstration of `if` statements using Boolean literals (`True` and `False`) and a Boolean variable.

```
1 >>> if True:
2 ...     print("It's true!")
3 ...
```

[1] Contrast this to, say, `int`s or `str`s where there are effectively an infinite number of possible literals! The number of digits in an integer literal or the number of characters in a string literal are only limited by the computer’s memory and our patience.

```
4  It's true!
5  >>> if False:
6  ...     print("It's true!")
7  ...
8  >>> bt = True
9  >>> type(bt)
10 <class 'bool'>
11 >>> if bt:
12 ...     print("bt is a true variable.  Or should I say True?")
13 ...
14 bt is a true variable.  Or should I say True?
```

In line 8 the variable `bt` is set equal to `True`. The `type()` function is used in line 9 to show that `bt` has a type of `bool`. The `if` statement in lines 11 and 12 and the subsequent output in line 14 demonstrate that a variable can be used as the test expression.

`if` statements that employ only literals, such as those in Listing 11.2, are of no practical use: if we already know a test expression is `True`, we simply write the code in the body (i.e., discard the header). On the other hand, if the test expression is known to be `False`, we can do away with the `if` statement all together. There are several ways to obtain practical and meaningful test expressions. In the following section we discuss *comparison operators* and the important role they play in many conditional statements. Before doing this, however, we want to consider a simpler and remarkably powerful way to obtain meaningful test expressions.

Python is able to map every object to either `True` or `False`. Since the result of any expression is an object, this means that Python is able to use *any* expression as the test expression in an `if` statement header!

Python considers the following to be equivalent to `False`: the numeric value zero (whether integer, `float`, or complex), empty objects (such as empty strings, `lists`, or `tuples`), `None`, and `False`. Everything else is considered to be equivalent to `True`. If you are ever unsure whether a particular object (or expression) is considered `True` or `False`, you can use the `bool()` function to obtain the Boolean representation of this object (or the Boolean representation of whatever object the expression evaluates to). The mapping of objects to Boolean values is illustrated in Listing 11.3.

Listing 11.3 Demonstration that all objects can be treated as Boolean values and hence any expression can be used as the "test expression" in an `if` statement's header.

```
1  >>> x = 0; y = 0.0; z = 0 + 0j
2  >>> bool(x), bool(y), bool(z)
3  (False, False, False)
4  >>> x = -1; y = 1.e-10; z = 0 + 1j
5  >>> bool(x), bool(y), bool(z)
6  (True, True, True)
7  >>> x = []; y = [0]; z = "0"
8  >>> bool(x), bool(y), bool(z)
9  (False, True, True)
10 >>> if 1 + 1:
```

```
11 ...     print("Test expression considered True.")
12 ...
13 Test expression considered True.
14 >>> if 6 * 3 - 18:
15 ...     print("Test expression considered True.")
16 ...
17 >>> s = ""
18 >>> if s:
19 ...     print("Nothing to say.")
20 ...
21 >>>
```

In line 1, identifiers are assigned the integer, `float`, and complex representations of zero. In line 2 `bool()` is used to determine the Boolean equivalents of these values. All are considered `False` as reported in line 3. In line 4 these same identifiers are set to non-zero values. Lines 5 and 6 show that the variables are now all considered to be `True`. In line 7 the variables `x`, `y`, and `z` are assigned an empty `list`, a `list` containing the integer `0`, and a string containing the character `'0'`, respectively. In this case, as shown in lines 8 and 9, `x` is considered `False` while `y` and `z` are considered `True`.

The `if` statement headers in lines 10, 14, and 18 serve to demonstrate that *any* expression can be used as the "test expression." In line 10 the test expression evaluates to `2` which is considered `True` (because it is non-zero) and hence the body of the statement is executed. In line 14, `6 * 3 - 18` evaluates to `0` which is considered `False` and hence the body in line 15 is not executed (note that the string argument of the `print()` function in line 15 does *not* reflect the actual state of the test expression: were the test expression `True`, this string would have been printed). In line 18 the test expression consists simply of the variable `s`. Since `s` was initialized to the empty string in line 17, this is considered `False` and the body is not executed.

To further illustrate what is `True` and what is `False`, the code in Listing 11.4 creates a `list` of objects and then sifts through this `list` to identify the objects considered to be `True`. A `list` of objects is created in line 1 and assigned to the variable `items` while in line 2 the variable `truth` is assigned the empty `list`. The `for`-loop starting in line 3 cycles through each item of `items`. The `if` statement in lines 4 and 5 appends an item to the `truth` list if the item is considered `True` (nothing is done with items considered `False`). Lines 7 and 8 show the objects that are considered `True`. (Items in `items` that do not subsequently appear in `truth` are considered `False`.)

Listing 11.4 Further demonstration of the distinction between objects considered `True` and `False`.

```
1 >>> items = [0, 1, 7 / 2, None, False, "hi", (), 7.3, (0, 0)]
2 >>> truth = []
3 >>> for item in items:
4 ...     if item:
5 ...         truth.append(item)
6 ...
7 >>> truth
8 [1, 3.5, 'hi', 7.3, (0, 0)]
```

Now let us consider two examples that exploit the fact that all expressions in Python evaluate to a value that is considered either to be True or False. These two examples are somewhat challenging: until you become comfortable with what is and isn't considered to be True, the following code may seem slightly mysterious. This code actually can be written in a more readable way using the comparison operators described in the next section. Nevertheless, it is worth taking the time to understand it because doing so will put you well on your way to understanding logical constructs in Python.

Assume we are asked to write a function called remove_repeats() that takes a list of numbers as its only argument. The function returns a new list that is a copy of the original list but with any consecutive repeats removed. We are told the original list will always have at least one element. As examples of the function's behavior, when it is passed [1, 2, 2, 3], it returns [1, 2, 3], i.e., the second 2 is removed. However, when the function is passed [1, 2, 3, 2], it returns [1, 2, 3, 2], i.e., although 2 appears twice in this list, it does not appear consecutively.

How do we implement this function? We know we must cycle over the elements of the original list; thus we will need to use a for-loop. But, do we need to cycle over all the elements? After some thought, we realize the first element in the list has no preceding value, so there is nothing to check regarding it (as the first element, it can't be a repeat!). Thus the for-loop can start with the second element. Because we are building a new list, we need an accumulator into which we append the non-repeated elements. Rather than initializing this accumulator to an empty list, we can initialize it to contain the first value from the original list. But, which other elements should be appended to the accumulator? If an element is not the same as the preceding one, then it should be appended. This indicates we need an if statement. The "trick," it seems, is to come up with the appropriate test expression for the if statement. Listing 11.5, which is discussed below, provides the code to implement this function. (Before looking at this, you may want to try to implement your own solution!)

Listing 11.5 A function that returns a new list that is a copy of the list the function is passed but consecutively repeated numeric values are removed.

```
1  >>> def remove_repeats(xlist):
2  ...     clean = [xlist[0]]
3  ...     for i in range(1, len(xlist)):
4  ...         if xlist[i] - xlist[i - 1]:
5  ...             clean.append(xlist[i])
6  ...     return clean
7  ...
8  >>> remove_repeats([1, 2, 2, 3])
9  [1, 2, 3]
10 >>> remove_repeats([1, 2, 3, 2])
11 [1, 2, 3, 2]
12 >>> ylist = [2, 7, 7, 7, 8, 7, 7, 9, 1.2, 1.2]
13 >>> remove_repeats(ylist)
14 [2, 7, 8, 7, 9, 1.2]
```

The function is defined in lines 1 through 6. The header uses the identifier xlist for the sole formal parameter. In line 2 the accumulator clean is created with a single element corresponding to the first element of xlist. The loop variable i in the header of the for-loop in line 3 takes on values ranging from 1 to the last valid index for xlist. Thus, thinking in terms of indices, for the iterations of the loop, i takes on the index of the second element of xlist, then the third element, and so on, until reaching the end of xlist. Now, consider the if statement in lines 4 and 5. The test expression is simply the difference of xlist[i] and xlist[i - 1], i.e., the preceding element is subtracted from the "current" element. This difference will be zero when the two elements are equal. Because zero is considered to be False, the body of the if statement will not be executed if the two values are equal! Lines 8 through 14 show the function works properly.

As another challenging problem, assume we must write a function called find_averages() that takes a single argument which is a list. Each element of this list is itself a list. The inner lists consist of zero or more numerical values, i.e., the number of elements in each of the inner lists is not known in advance. The function find_averages() must calculate and print the average of each inner list that has at least one element. If an inner list has no elements, it is simply ignored. The code to accomplish this is shown in Listing 11.6. The code is discussed following the listing.

Listing 11.6 A function to calculate the average of non-empty lists that are contained within an outer list.

```
1  >>> def find_averages(xlist):
2  ...     for inner in xlist:
3  ...         if inner:
4  ...             total = 0
5  ...             for num in inner:
6  ...                 total = total + num
7  ...             print(total / len(inner))
8  ...
9  >>> zlist = [[20, 15, 40], [], [5, 8, 10, 15, 100], [3.14]]
10 >>> find_averages(zlist)
11 25.0
12 27.6
13 3.14
```

The find_averages() function is defined in lines 1 through 7. The for-loop starting in line 2 cycles through each element of the list the function is passed. These elements are themselves lists. Recall that an empty list is considered to be False. Thus, only a non-empty list will make it into the body of the if statement in lines 3 through 7. Empty lists are effectively ignored. The code in lines 4 through 7 calculates and prints the average. An integer accumulator is initialized to zero in line 4. The for-loop in lines 5 and 6 adds all the values to the accumulator. Finally, the print() statement in line 7 displays the average. The remaining lines of the listing demonstrate the function works properly.

Let's introduce another built-in function and consider an alternate implementation find_averages(). The built-in function sum() returns the sum of an iterable consisting of numeric values. This is

demonstrated in Listing 11.7 where, in line 1, `sum()` is passed the `list` consisting of the integers `1`, `2`, and `3`. The subsequent output, on line 2, is the sum of these values. In line 3, the identifier `ylist` is assigned a `list` of five numbers. This list is passed to the `sum()` function in line 4 and the output on line 5 is the sum of these values. Using the `sum()` function the average of a `list` can be obtained with a single expression and this fact is used in line 9 of Listing 11.7 to simplify the implementation of the `find_averages()` function. The remainder of Listing 11.7 shows that this new implementation of `find_averages()` yields the same results as the implementation in 11.6.

Listing 11.7 Another function to calculate the average of non-empty `lists` that are contained within an outer `list`. This function is similar to the one presented in Listing 11.6, but this function uses the built-in `sum()` function.

```
>>> sum([1, 2, 3])
6
>>> ylist = [7.3, 8.4, 9.5, 100, 1000]
>>> sum(ylist)
1125.2
>>> def find_averages(xlist):
...     for inner in xlist:
...         if inner:
...             print(sum(inner) / len(inner))
...
>>> zlist = [[20, 15, 40], [], [5, 8, 10, 15, 100], [3.14]]
>>> find_averages(zlist)
25.0
27.6
3.14
```

The code in Listing 11.5 uses the fact that zero is considered to be `False` to affect the flow of execution. Alternatively, we can think of the code in Listing 11.5 as exploiting the fact that only non-zero values are considered to be `True`. The code in Listings 11.6 and 11.7 uses the fact that an empty `list` is considered to be `False` to affect the flow of exectution. An alternate way of thinking about this code is that only non-empty `lists` are considered to be `True`.

11.2 Comparison Operators

The fact that Python is able to treat every object (and hence every expression) as either `True` or `False` provides a convenient way to construct parts of many programs. However, we are often interested in making a decision based on a comparison. For example, when it comes to determining whether the weather is "nice" we may want to base our decision on the temperature and wind speed, for example, is the temperature greater than 60 but less than 80 and is the wind speed less than 15 mph? In order to accomplish such comparisons we use *comparison operators*.[2]

[2] In other languages these are often called relational operators.

Like algebraic operators, comparison operators take two operands. You are already familiar with many of the symbols used for comparison operators. For example, the greater-than sign, >, is used to ask the question: Is the operand on the left greater than the operand on the right? If so, the expression evaluates to `True`. If not, the expression evaluates to `False`. In your math classes the greater than sign was typically used to *establish* a relationship. For example, in a math class $x > 5$ often establishes that the value of x must be greater than 5. However, in Python, `x > 5` does not establish that the variable `x` is greater than `5`. Instead, this expression evaluates whether or not `x` is greater than `5`.

The code in Listing 11.8 illustrates the use of the greater-than comparison operator. In line 1 we ask if `5` is greater than `7`. The `False` in line 2 says this is not so. The simultaneous assignment in line 3 sets `x` and `y` to `45` and `-3.0`, respectively. The comparison in line 4 evaluates to `True` because `x` is greater than `y`. The statement in line 6 assigns to the identifier `result` the result of the comparison on the right side of the assignment operator. The expression on the right side produces the Boolean answer to the question: Is `x` greater than the value of `y` plus `50`? Note that all arithmetic operators have higher precedence than the comparison operators, and all comparison operators have equal precedence. Because `y` plus `50` is `47`, which is greater than the value of `x`, the right side of 6 evaluates to `False`. Indeed, we see, in line 8, that `result` has been assigned `False`. The discussion continues following the listing.

Listing 11.8 Demonstration of the greater-than comparison operator.

```
1  >>> 5 > 7                  # Is 5 greater than 7?
2  False
3  >>> x, y = 45, -3.0
4  >>> x > y                  # Is 45 greater than -3.0?
5  True
6  >>> result = x > y + 50 # Is 45 greater than -3.0 + 50?
7  >>> result
8  False
9  >>> if 1 + 1 > 1:
10 ...     print("I think this should print.")
11 ...
12 I think this should print.
13 >>> "hello" > "Bye"        # Comparison of strings.
14 True
15 >>> "AAB" > "AAC"
16 False
```

The header of the `if` statement in line 9 shows that comparison operators can be used as part of the test expression. Here we are evaluating the question: Is one plus one greater than one? Because this evaluates to `True`, the body of the `if` statement is executed which produces the output shown in line 12.

Python can compare more than just numbers. We can, for example, compare strings. When comparing strings, the comparison is based on ASCII values. Since letters in ASCII are in alphabetic order, comparing ASCII values is effectively the same as comparing alphabetic ordering.

For strings comparison, the comparison starts with the first characters and continues until there is a mismatch in the characters or until reaching the end of the shorter string. Keep in mind that uppercase letters come before lowercase letters. So, for example, `'a'` is greater than `'Z'` while `'z'` is greater than `'A'` (in fact, `'z'` is greater than every letter other than itself!). As lines 13 through 16 of Listing 11.8 show, `hello` is greater than `Bye` but `AAB` is not greater than `AAC`. If one wants to ensure case is ignored in a comparison, the string methods `upper()` or `lower()` can be used to create strings that have all the letters in a single case.

Listing 11.9 lists most of the comparison operators. (Two comparison operators are omitted, one of which is the `is` operator introduced in Sec. 7.3. We return to the other omitted operator in Sec. 11.8.)

Listing 11.9 Comparison operators. These operators are *binary operators* in the sense that they require two operands.

`x < y`	Is `x` less than `y`?
`x <= y`	Is `x` less than or equal to `y`?
`x == y`	Is `x` equal to `y`?
`x >= y`	Is `x` greater than or equal to `y`?
`x > y`	Is `x` greater than `y`?
`x != y`	Is `x` not equal to `y`?

Because of keyboard constraints, we must write >= for "greater than or equal to" rather than the symbol you would typically use in a math class, i.e., $\geq$. Similarly, we write <= for "less than or equal to" instead of $\leq$. Because a single equal sign is the symbol for the assignment operator, we must write two equal signs to test for equality of two operands.[3] To test for inequality we must write != instead of the more familiar symbol $\neq$.

Listing 11.10 demonstrates the use of several comparison operators. In line 1 the integer `7` is compared with the `float 7.0`. The result on line 2 indicates these two are equal. *However*, generally one should *avoid* using the equality comparison with `floats`. Keep in mind that `floats` are often approximations to what we *think* are the values. This is illustrated in lines 3 through 9. In line 3 `x` is initialized to `0.1`, i.e., one-tenth. In line 4 the integer `1` is compared to `10` times `x`. The result on line 5 shows, as we would expect, that one and ten times one-tenth are equal. However, line 6 asks if the integer `1` is equal to the sum of ten copies of `x`. Line 7 reports `False`, i.e., as far as the computer is concerned, one and the sum of ten one-tenth's are not equal! Lines 8 and 9 indicate why this is so. This result is a consequence of the binary representation of one-tenth requiring an infinite number of binary digits. However, only 64 bits are used to store the number in the computer. This truncation results in a slight round-off error. Thus, when we sum `0.1` ten times, we obtain a value that is ever so slightly smaller than `1.0`. Nevertheless, this difference is enough to make the comparison `False`. The discussion continues following the listing.

[3] If you never confuse == and = in conditional statements, you will probably be the first programmer in history to accomplish this feat. In some languages, such as C, this error can lead to insidious bugs. Python, however, does catch this error because Python doesn't allow assignment operations in the header of a conditional statement.

Listing 11.10 Demonstration of the use of various comparison operators.

```
>>> 7 == 7.0
True
>>> x = 0.1
>>> 1 == 10 * x
True
>>> 1 == x + x + x + x + x + x + x + x + x + x
False
>>> x + x + x + x + x + x + x + x + x + x
0.9999999999999999
>>> 7 != "7"
True
>>> 'A' == 65
False
>>> threshold = 5
>>> xlist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> at_or_above = []
>>> for item in xlist:
...     if item >= threshold:
...         at_or_above.append(item)
...
>>> at_or_above
[5, 6, 7, 8, 9]
```

In line 10 we ask if the integer 7 is *not* equal to the string '7'. Integers are never equal to strings, so this is True, i.e., they are not equal. To emphasize this point, in line 12 the string 'A' is compared to the integer 65. Recall that the ASCII value for the letter 'A' is 65, so we might guess that these would be equal. However, as reported in line 13, these objects are not equal.

The code in lines 14 through 19 appends copies of the values from the list xlist onto the list at_or_above if the value is greater than or equal to a specified threshold. The threshold is set to 5 in line 14. The at_or_above list is initialized to the empty list in line 16. The for-loop in lines 17 through 19 cycles through each item of xlist. If the item is greater than or equal to the threshold, the value is appended to at_or_above. We see, in lines 21 and 22, that the at_or_above list contains the values from xlist that were equal to or greater than the threshold.[4]

if statements can be nested inside other if statements. Listing 11.11 demonstrates this. In lines 1 and 2 the identifier creatures is assigned a list that has four elements. Each of these elements is a three-element list that contains values representing a species, a name, and an age. In this particular list there are two dogs and two humans. The for-loop in lines 3

[4] list comprehension, which is discussed in Sec. 7.7, allows one to rewrite lines 16 through 19 as a single statement. It was mentioned in Sec. 7.7 that list comprehensions could include conditional statements. The appropriate list comprehension would be:

```
at_or_above = [item for item in xlist if item >= threshold]
```

through 10 cycles through each element of the `creatures list`. In the header of the `for`-loop, simultaneous assignment is used to assign values to the variables `species`, `name`, and `age`. Thus, in the first iteration of the loop when we are working with the first element of the `creatures` list, `species` is assigned `dog`, `name` is assigned `Rover`, and `age` is assigned `7`. The discussion of the code continues following the listing.

Listing 11.11 Demonstration of nested `if` statements.

```
1  >>> creatures = [['dog', 'Rover', 7], ['human', 'Fred', 2],
2  ...              ['dog', 'Fido', 1], ['human', 'Sally', 32]]
3  >>> for species, name, age in creatures:
4  ...     if species == 'dog':
5  ...         if age <= 2:
6  ...             print(name, "is just a pup.")
7  ...         if age > 2:
8  ...             print(name, "is all grown up.")
9  ...     if species == 'human':
10 ...         print("Hi " + name + ".")
11 ...
12 Rover is all grown up.
13 Hi Fred.
14 Fido is just a pup.
15 Hi Sally.
```

The `if` statements in lines 4 and 9 allow us to handle dogs and humans differently (if there were other species in the `creatures` list, nothing would be done with them). Within the body of the `if` statement for dogs are two more `if` statements. The `if` statement in lines 5 and 6 serves to recognize a younger dog while the one in lines 7 and 8 announces a mature dog. The `if` statement for humans, in lines 9 and 10, simply greets the person. The output from the loop is shown in lines 12 through 15.

As an example that ties together many of the things we have learned, let's write a function called `add_day()` that takes a two-element `list` as its single argument. Both elements are integers. The first element represents a month and the second represents a day within that month. The month can be between `1` and `12` while the day is between `1` and the maximum number of days for that month (for this example we ignore leap years and assume February has 28 days). The function `add_day()` adds one day to the date it is passed as an argument and returns the resulting date in a new two-element list. The code for this function is given in Listing 11.12 between lines 1 and 18. The first "real" line in the body of the function, line 8, initializes the `tuple days_in_month` to the number of days in each month. Line 9 simply uses simultaneous assignment to give convenient names to the month and day that were passed to the function. The `if` statement in line 12 checks whether the day is less than the number of days in the given month (note that we must subtract `1` from `month` to obtain the correct index, e.g., the number of days in January is given by `days_in_month[0]`). If the day is indeed less than the number of days in the month, the statement in line 13 returns a two-element `list` in which the month is unchanged and the day is incremented by one. Thus, if and when the body of this `if` statement is executed,

the `return` statement in line 13 ensures that control is returned to the point in the program where this function was called and no other statements in this function are executed. However, if this condition is not `True`, i.e., the day is not less than the number of days in the month, then the body of the `if` statement is skipped and execution of statements continues with line 17. The discussion of this code is continued following the listing.

Listing 11.12 Function to calculate the new month and day when the date is advanced by one day (leap years are ignored).

```
>>> def add_day(date):
...     """
...     Add a day to the given date.  The date is a two-element list
...     containing the month and the day of the month.  The function
...     returns a new list containing the new date.
...     """
...     # Number of days in each month.
...     days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
...     month, day = date       # Extract month and day from given date.
...     # If there is another day left in month, simply increment the
...     # day.  Place the month and new day in a list and return it.
...     if day < days_in_month[month - 1]:
...         return [month, day + 1]
...     # If we reached this point, we are at the last day of the month.
...     # So, increment the month (ensuring we return to January if
...     # we're at December), and set the day to 1.  Return new date.
...     month = month % 12 + 1 # Increment month.
...     return [month, 1]
...
>>> today = [3, 31]     # March 31st.
>>> add_day(today)
[4, 1]
>>> add_day([7, 3])     # July 3rd.
[7, 4]
>>> add_day([12, 31])   # New Year's Eve, i.e., December 31st.
[1, 1]
```

In line 17 the `month` is taken modulo `12`. Thus, if it is December, `month % 12` evaluates to `0`. This value is incremented by `1` and assigned back to `month`. Given that we just "rolled over" to a new month, the day is set to `1`. The `list` returned in line 18 has the `month` that was calculated in line `17` and the day "hardwired" to `1`.

The remaining lines of Listing 11.12 demonstrate that the function works properly. In line 20 the `list today` is defined with the date corresponding to March 31st. When passed this date, `add_day()` returns the date corresponding to April 1st as shown in line 22. In the following two calls to `add_day()` the date is directly specified as a literal argument. We see that July 3rd is incremented to July 4th and December 31st is incremented to January 1st.

11.3 Compound Conditional Statements

In the `add_day()` function in Listing 11.12 there are some statements that should be executed only if the given day is not the last day of the month. On the other hand, there are other statements that should only be executed if the day *is* the last day of the month. In this function an `if` statement coupled with a `return` statement ensures that only one set of statements is executed. There are many instances like this one in which we want either one block of code or another to be executed. However, we generally can't use (or don't want to use) the `return` statement as is done in the `add_day()` function. Instead, we should use an `if-else` statement. Extending this idea, there are times when there are several different blocks of code that we want to select from, i.e., only one of multiple possible blocks should be executed. In such cases we should use an `if-elif-else` statement. We explore both `if-else` and `if-elif-else` statements in this section.

11.3.1 `if-else` Statements

The template for an `if-else` statement is shown in Listing 11.13.

Listing 11.13 Template for an `if-else` statement.

```
1 if <test_expression>:
2     <body_1>
3 else:
4     <body_2>
```

As with an `if` statement, if the `test_expression` in the header evaluates to `True`, then the code immediately following the header is executed, i.e., the code identified as `body_1`. However, if the `test_expression` evaluates to `False`, then the code following the `else` is executed, i.e., `body_2` is executed but `body_1` is skipped. Thus, one body of code is executed but not the other. Listing 11.14 demonstrates the use of `if-else` statements.

Listing 11.14 Demonstration of an `if-else` statement.

```
1 >>> grade = 95
2 >>> if grade > 80:
3 ...     print("Keep up the good work.")
4 ... else:
5 ...     print("Try working harder.")
6 ...
7 Keep up the good work.
```

In line 1 `grade` is assigned the value `95`. In line 2 we ask if the grade is greater than `80`. Since it is, the first body is executed (and the second body is skipped). This results in the output shown in line 7.

As another example, let's consider a function called `test_square()` that tests whether a number is a "perfect square," i.e., if a given number can be formed by the square of an integer.

If a number is a perfect square, the function announces this fact as well as the integer that forms the perfect square. If the number is not a perfect square, this fact is announced. A function to accomplish this is shown in lines 1 through 7 of Listing 11.15. The code is discussed following the listing.

Listing 11.15 Function to test whether a number is a perfect square. An appropriate message is printed that depends on whether or not the number is a perfect square.

```
1  >>> def test_square(num):
2  ...     root = int(num ** 0.5)
3  ...     if num == root * root:
4  ...         print(num, " is a perfect square (",
5  ...             root, " * ", root, ").", sep="")
6  ...     else:
7  ...         print(num, "is not a perfect square.")
8  ...
9  >>> test_square(49)
10 49 is a perfect square (7 * 7).
11 >>> test_square(50)
12 50 is not a perfect square.
13 >>> test_square(622521)
14 622521 is a perfect square (789 * 789).
```

The function header in line 1 has the single parameter of `num`. In line 2 the square root of `num` is calculated (by raising the value to the power `0.5`). This square root is converted to an integer (i.e., any fractional part is discarded) and assigned to the variable `root`. In line 3 we check if the square of `root` is equal to `num`. If it is, the number must be a perfect square and hence the `print()` statement that spans lines 4 and 5 is executed. If the number is not a perfect square, the statement following the `else` in line 6 is executed, i.e., the `print()` statement in line 7.

In line 9 the function is called with an argument of `49`. Because this is a perfect square, the message in line 10 announces the value as a perfect square. Lines 11 and 12 show that `50` is not a perfect square while lines 13 and 14 show that `622521` is.

In many programs we need to write or use functions that test whether something *is* true. These functions are often identified by names that start with `is`. We have actually seen the names of several of these functions (but we have not used them). Looking closely at Listing 5.5, we see that strings have methods with names such as `isalnum()`, `isupper()`, and `isdigit()`. `isalnum()` returns `True` if the string *is* one that consists of only alphanumeric characters (i.e., letters or digits but not punctuation or whitespace) and has at least one alphanumeric character; `isupper()` returns `True` if the string *is* one where all the "cased" characters are uppercase letters and there is at least one cased letter; and `isdigit` returns `True` if the string *is* one that contains only characters corresponding to digits.

Listing 11.16 demonstrates the use of `isalnum()`, `isupper()`, and `isdigit()` (although here we use these methods on strings consisting of a single character). In line 1 a string is assigned to the variable `s` that has a mix of letters, digits, and punctuation. In lines 2 through 4 a `for`-loop cycles over all the characters of `s`. The `isupper()` method is used to test whether the character

is uppercase. If it is, the `print()` statement in line 4 prints the character. Since only the first character in `s` is uppercase, the output of this loop is merely the letter `T` shown in line 6. The loop in lines 7 to 9 is similar to the previous loop except now the `isalnum()` method is used. In this case the output, shown in line 11, consists of all the letters and digits in the string. The final loop, in lines 12 through 14, uses the `isdigit()` method. In this case the output, shown in lines 16 through 18, is the digits in the string.

Listing 11.16 Demonstration of various "is" methods for strings.

```
>>> s = "Test 123!!!"
>>> for ch in s:          # Find all uppercase letters.
...     if ch.isupper():
...         print(ch)
...
T
>>> for ch in s:          # Find all alphanumeric characters.
...     if ch.isalnum():
...         print(ch, end=" ")
...
T e s t 1 2 3 >>>
>>> for ch in s:          # Find all digits.
...     if ch.isdigit():
...         print(ch)
...
1
2
3
```

Now let's return to the testing of perfect squares. Assume we want to write a function called `is_square()` that returns `True` if the argument is a perfect square and returns `False` otherwise. As a first attempt, we can base this new function on the `test_square()` function in Listing 11.15. Thus, we might implement the function as follows:

```
def is_square(num):
    root = int(num ** 0.5)
    if num == root * root:
        return True
    else:
        return False
```

This implementation is, in fact, correct in that it does exactly what we want. *However*, this implementation is not considered the best one in that there is really no reason for the `if-else` statement! Keep in mind that expressions with comparison operators evaluate to `True` or `False`. Hence a better implementation of this function is the one shown in lines 1 through 3 in Listing 11.17. Note especially line 3 where the `return` statement says to return the value to which the comparison evaluates. The remaining code in the listing demonstrates that this function works properly.

Listing 11.17 A function to test whether a number is a perfect square. The function directly returns the value to which the comparison evaluates.

```
>>> def is_square(num):
...     root = int(num ** 0.5)
...     return num == root * root
...
>>> is_square(49)
True
>>> is_square(50)
False
>>> if is_square(1234321):
...     print("Perfect square.")
... else:
...     print("Not a perfect square.")
...
Perfect square.
```

11.3.2 `if-elif-else` Statements

Using an `if-elif-else` statement we can write programs that execute at most one among multiple bodies of code. The `elif` should be thought of as "else if." The template for an `if-elif-else` statement is shown in Listing 11.18. The statement must start with an `if` header (and its corresponding body). This is followed by any number of `elif` headers (and their bodies). Each `elif` header has its own test expression. The final `else` is optional. When this statement is executed, the first test expression is evaluated. If it is `True`, the first body is executed and the rest of the `if-elif-else` statement is skipped. If the first test expression evaluates to `False`, the first body is skipped and the second test expression is evaluated. If it is `True`, the second body is executed and the rest of the `if-elif-else` statement is skipped. If the second test expression evaluates to `False`, the second body is skipped and the third test expression is evaluated. And so on. If none of the test expressions are `True`, then the body accompanying the `else` part of the statement is executed if it is present. However, since the `else` part is optional, it may be that none of the bodies are executed.

Listing 11.18 Template for an `if-elif-else` statement.

```
if <test_expression_1>:
    <body1>
elif <test_expression_2>:
    <body2>
elif <test_expression_3>:
    <body3>
      .
```

```
8          .
9          .
10 else:
11     <bodyN>
```

As an example, let's consider the grading of a course in which scores are on a 100-point scale. For this particular course scores are mapped to letter grades as follows:

A	$100 \geq$ score ≥ 90
B	$90 >$ score ≥ 80
C	$80 >$ score ≥ 70
D	$70 >$ score ≥ 60
F	$60 >$ score

A function to implement this grading is shown in Listing 11.19. The function is defined in lines 1 through 11 while the `if-elif-else` statement spans lines 2 through 11. In line 2 a score is checked to see whether it is greater than or equal to `90`. (Although the upper limit on the score is nominally `100`, we do not check this.) If the score is greater than or equal to `90`, the `print()` statement in line 3 is executed. Then the flow of execution drops to the end of the `if-elif-else` statement. As there are no further statements in this function, Python will return to the point in the program where the function was called (as a reminder, this is a void function because it does not return anything). If the score is less than `90`, the comparison in line 4 is made. Note that we only check if the number is greater than or equal to `80`. From the scoring described above, it seems that to establish whether a score is a B we have to establish that the score is both greater than or equal to `80` *and* less than `90`. This is true, but the first part of the statement takes care of any scores that are at or above `90`. Hence, if we have reached line 4 we already know the score is less than `90`. The rest of the function continues similarly. The remainder of the listing, in lines 13 through 20, consists of calls to `show_grade()` that demonstrate that it works properly.

Listing 11.19 Use of an `if-elif-else` statement to map a numeric score to a letter grade.

```
1  >>> def show_grade(score):
2  ...     if score >= 90:
3  ...         print("A")
4  ...     elif score >= 80:
5  ...         print("B")
6  ...     elif score >= 70:
7  ...         print("C")
8  ...     elif score >= 60:
9  ...         print("D")
10 ...     else:
11 ...         print("F")
12 ...
13 >>> show_grade(91)
14 A
15 >>> show_grade(80)
```

```
B
>>> show_grade(69)
D
>>> show_grade(20)
F
```

The following is a flawed version of the `show_grade()` function. See whether you can anticipate what this function produces when passed an argument of `91`. (The answer is provided below.)

```
def show_flawed_grade(score):
    if score >= 90:
        print("A")
    if score >= 80:
        print("B")
    if score >= 70:
        print("C")
    if score >= 60:
        print("D")
    else:
        print("F")
```

Calling this function with an argument of `91` results in the following:

```
>>> show_flawed_grade(91)
A
B
C
D
```

The problem with this function is that there are three standalone `if` statements and one `if-else` statement. A score of `91` results in each of the comparisons evaluating to `True`. Hence each body is executed except the one associated with the final `else`. There actually is a way to implement the grading function using standalone `if` statements, but we must ensure that a score is between the upper and lower limits for the particular letter grade. We defer further discussion of this until we have presented the operators described in the next section.

11.4 Logical Operators

We started Sec. 11.1 with a description of a Boolean expression taken from daily life, namely the decision to go to the matinee if the following evaluates to true: *not* nice weather *and* rating over 80 percent. Deciding what constitutes "nice" weather may be a complicated one, but let us assume we are thinking Pythonically and have a Boolean variable `nice_weather` that has been appropriately set. Further assume we have a variable `rating` that stores the rating for a particular movie (and perhaps we'll need a `for`-loop to cycle through the ratings of multiple

movies). We can test if the rating is over 80 using an expression such as `rating > 80`. But how do we combine these things to realize our complete Boolean expression? We actually have learned enough that we can construct somewhat awkward code involving nested `if` statements that will, say, call a `go_to_movie()` function if the Boolean expression is true. However, there is a much more elegant way to implement this if we use *logical operators* which are often called *Boolean operators*.

Logical operators are also something with which you are quite familiar. They are simply the operators `and`, `or`, and `not`. Although you probably don't think of it in these terms, `not` is a unary operator that changes its operand from `True` to `False` or, conversely, from `False` to `True`. Both `and` and `or` are binary operators. An expression involving `and` is considered `True` only if both operands evaluate to `True`. An `or` expression is considered `True` if either or both of the operands evaluate to `True`. Of these three logical operators, `not` has the highest precedence, followed by `and`, and finally `or` has the lowest precedence. All these operators have lower precedence than the comparison operators (and, hence, by extension, lower precedence than the arithmetic operators). As always, if you are ever in doubt about operator precedence, use parentheses to ensure the order of operation you want.

Returning to the movie-going Boolean expression, this could be realized with something as simple as the following:

```
if not nice_weather and rating > 80:
  go_to_movie()
```

This is syntactically correct Python code. If you substitute the word "then" for the colon, this almost reads like English! However, before we dive into more details concerning these operators, we must mention that a straight translation from something said in English to a logical expression isn't always so simple. Without some careful thought, such a translation can lead to subtle bugs and we consider one such example in Sec. 11.7.

Listing 11.20 demonstrates the behavior of the logical operators. Here we use Boolean literals as the operands, but keep in mind that in practice the operands can be any expression! Because `not` has the highest precedence, it modifies the value to its right before the value is used by any other operator. This fact is illustrated in lines 5 and 11. Lines 13 and 15 show that we can have multiple `and`'s and `or`'s in an expression. Line 15 and the subsequent result in line 16 demonstrate that `and` has higher precedence than `or`.

Listing 11.20 Demonstration of the `and`, `or`, and `not` logical operators.

```
1  >>> not True
2  False
3  >>> False and True
4  False
5  >>> not False and True
6  True
7  >>> (not False) and True      # Same as previous statement.
8  True
9  >>> True or False
10 True
```

```
11 >>> not False or True          # Same as: (not False) or True.
12 True
13 >>> not (False or True)
14 False
15 >>> False and False or True   # Same as: (False and False) or True.
16 True
17 >>> False and (False or True)
18 False
```

Python allows the conversion of Boolean values to integers using `int()`. The integer equivalent of `False` is `0` while the integer equivalent of `True` is `1`. This conversion is not something we generally care about, but we can exploit it to create easily readable *truth tables* that show the outcomes of a logical expression for all possible combinations of the terms in this expression. This is demonstrated in Listing 11.21 which is discussed following the listing.[5]

Listing 11.21 Truth tables can be constructed by cycling over all the possible inputs to a logical function. Here `0` represents `False` and `1` represents `True`.

```
1  >>> booleans = [False, True]
2  >>> for x in booleans:
3  ...     for y in booleans:
4  ...         print(int(x), int(y), "=>", int(x or y))
5  ...
6  0 0 => 0
7  0 1 => 1
8  1 0 => 1
9  1 1 => 1
10 >>> for x in booleans:
11 ...     for y in booleans:
12 ...         print(int(x), int(y), "=>", int(not x and y or x and not y))
13 ...
14 0 0 => 0
15 0 1 => 1
16 1 0 => 1
17 1 1 => 0
```

In line 1 the `list booleans` is initialized to the two possible Boolean values. The header of the `for`-loop in line 2 causes the variable `x` to take on these values while the header of the `for`-loop in line 3 causes the variable `y` also to take on these values. The second loop is nested inside the first. The `print()` statement in line 4 displays (in integer form) the value of `x`, the value of `y`, and the result of the logical expression `x or y`. The output is shown in lines 6 through 9, i.e., the first column corresponds to `x`, the second column corresponds to `y`, and the final column corresponds to the result of the logical expression using these values. The nested `for`-loops in lines 10 through

[5]Because `0` and `1` are functionally equivalent to `False` and `True`, we can use these integer values as the elements of the `list booleans` in line 1 with identical results.

12 are set up similarly. The only difference is the logical expression that is being displayed. The expression in line 12 appears much more complicated than the previous one, but the result has a rather simple description in English: the result is `True` if either `x` or `y` is `True` but not if both `x` and `y` are `True`.[6]

11.5 Multiple Comparisons

In the `show_grade()` function in Listing 11.19 an `if-elif-else` statement was used to determine the range within which a given score fell. This construct relied upon the fact that we progressively tested to see if a value exceeded a certain threshold. Once the value exceeded that threshold, the appropriate block of code was executed. But, what if we want to directly check if a value falls within certain limits? Say, for example, we want to test if a score is greater than or equal to 80 but less than 90. If a value falls in this range, assume we want to print a message, for example, `This score corresponds to a B.` How should you implement this? Prior to learning the logical operators in the previous section, we would have used nested `if` statements such as:

```
if score >= 80:
    if score < 90:
        print("This score corresponds to a B.")
```

Having learned about the logical operators, we can write this more succinctly as

```
if score >= 80 and score < 90:
    print("This score corresponds to a B.")
```

In the majority of computer languages this is how you would implement this conditional statement. However, Python provides another way to implement this that is aligned with how ranges are often expressed in mathematics. We can directly "chain" comparison operators. So, the code above can be implemented as

```
if 80 <= score < 90:
    print("This score corresponds to a B.")
```

or

```
if 90 > score >= 80:
    print("This score corresponds to a B.")
```

This can be generalized to any number of operators. If `cmp` is a comparison operator and `op` is an operand, Python translates expressions of the form

```
op1 cmp1 op2 cmp2 op3 ... opN cmpN op{N+1}
```

to

[6]This expression is known as the *exclusive or* of `x` and `y`.

```
(op1 cmp1 op2) and (op2 cmp2 op3) ... and (opN cmpN op{N+1})
```

Even though some operands appear twice in this translation, at most, each operand is evaluated once.

Listing 11.22 demonstrates some of the ways that chained comparison can be used. In line 1 the variables `x`, `y`, and `z` are assigned the values `10`, `20`, and `30`, respectively. In line 2 we ask if `x` is less than `y` and `y` is less than `z`. The result of `True` in line 3 shows both these conditions are met. In line 4 we ask if `10` is equal to `x` and if `x` is less than `z`. The answer is again `True`. In line 6 we ask if `99` is greater than `x` but less than `y`. The `False` in line 7 shows `99` falls outside this range. Finally, the expressions in lines 8 and 10 show that we can write expressions that are not acceptable in mathematics but are valid in Python. In line 8 we are asking if `x` is less than `y` and if `x` is less than `z`. Despite the chaining of operators, Python partially decouples the chain and links the individual expressions with the logical `and` (as mentioned above). So, the expression in line 8 is not making a comparison between `y` and `z`. A similar interpretation should be used to understand the expression in line 10.

Listing 11.22 Demonstration of the use of chained comparisions.

```
1  >>> x, y, z = 10, 20, 30
2  >>> x < y < z
3  True
4  >>> 10 == x <= z
5  True
6  >>> x < 99 < y
7  False
8  >>> y > x < z
9  True
10 >>> x < z > y
11 True
```

11.6 `while`-Loops

By now, we are quite familiar with `for`-loops. `for`-loops are definite loops in that we can typically determine in advance how many times the loop will execute. (A possible exception to this is when a `for`-loop is in a function and there is a `return` statement in the body of the loop. Once the `return` statement is encountered, the loop is terminated as well as the function that contained the loop.) However, often we need to have looping structures in which the number of iterations of the loop cannot be determined in advance. For example, perhaps we want to prompt a user for data and allow the user to keep entering data until the user provides some signal that they are done. Rather than signaling when they are done, the user can potentially be asked to start by specifying the amount of data to be entered. In this case we can stick to using a `for`-loop, but this can be quite inconvenient for the user. Rather than using a definite loop, we want to use an *indefinite loop*

in which the number of times it iterates is not known in advance. In Python (and in many other languages) we implement this using a `while`-loop.

The template for a `while`-loop is shown in Listing 11.23. The header in line 1 consists of the keyword `while` followed by a `test_expression` (which can be any valid expression), and a colon. Following the header is an indented body. The `test_expression` is evaluated. If it evaluates to `True`, then the body of the loop is executed. After executing the body, the `test_expression` is evaluated again. While `test_expression` evaluates to `True`, the body of the loop is executed. When the `test_expression` evaluates to `False`, the loop is terminated and execution continues with the statement following the body.

Listing 11.23 Template for a `while`-loop.

```
1 while <test_expression>:
2     <body>
```

As an example of the use of a `while`-loop, let's write code that prompts the user for names (using the `input()` function). When the user is done entering names, they should signal this by merely hitting return. The code to accomplish this is shown in Listing 11.24. In line 1 the variable `prompt` is assigned the string that is used as the prompt (we create this variable since the prompt is used in two different statements). In line 2 `names` is initialized to the empty `list`. In line 3 the user is prompted to enter a name. The user is told to hit the return key when done. We see the user entered the name `Uma` on line 4 which is assigned to the variable `name`. The `while`-loop starts with the header on line 5 that says the body of the loop should be executed provided the variable `name` is considered `True`. Recall that an empty string is considered `False` but everything else is considered `True`. This header is equivalent to

```
while name != "":
```

However, the statement in line 5 is the more idiomatic way of expressing the desired behavior. The body of the `while`-loop consists of two statements: the first appends `name` to the `names list` while the second prompts the user for another name. In lines 9 through 11 we see the other names the user entered. In line 12 the user responded to the prompt by hitting the return key. Thus `name` was set to the empty string, the `test_expression` for the loop evaluated to `False`, and the loop terminated. Back at the interactive prompt in line 13 we echo the `names list` and see that it contains the four names the user entered.

Listing 11.24 Use of a `while`-loop to records names until the user is done.

```
1 >>> prompt = "Enter name [<ret> when done]: "
2 >>> names = []
3 >>> name = input(prompt)
4 Enter name [<ret> when done]: Uma
5 >>> while name:
6 ...     names.append(name)
7 ...     name = input(prompt)
```

```
...
Enter name [<ret> when done]: Ulrike
Enter name [<ret> when done]: Ursula
Enter name [<ret> when done]: Uta
Enter name [<ret> when done]:
>>> names
['Uma', 'Ulrike', 'Ursula', 'Uta']
```

11.6.1 Infinite Loops and `break`

There is something slightly awkward about the code in Listing 11.24. Note that there is one call to `input()` outside the loop (line 3) and another inside the loop (line 7). Both these calls use the same prompt and assign `input()`'s return value to the same variable, so there appears to be a needless duplication of code, but there doesn't seem to be a simple way around this duplication. The first call to `input()` starts the process. If a user enters a name at the first prompt, then the `while`-loop is executed to obtain a `list` with as many additional names as the user cares to enter.

However, there is an alternative construction that is arguably "cleaner." The new implementation relies on the use of a `break` statement. `break` statements are placed inside loops and are almost always embodied within an `if` statement. When a `break` statement is executed, the surrounding loop is terminated immediately (i.e., regardless of the value of the test expression in the header) and execution continues with the statement following the loop. `break` statements can be used with `for`-loops and `while`-loops.

Before showing examples of `break` statements, let us consider *infinite loops*. Infinite loops are loops that theoretically run forever because the text expression in the header of the loop never evaluates to `False`. Such a situation may arise either because the loop was intentionally coded in such a way or because of a coding error. For example, consider the following code where perhaps the programmer intended to print the numbers 0.1, 0.2, $\cdots$, 0.9:

```
x = 0.0
while x != 1.0:
    x = x + 0.1
    print(x)
```

In line 1 `x` is initialized to `0.0`. The header of the `while`-loop in line 2 dictates that the loop should be executed if `x` is not equal to `1.0`. Looking at line 3 in the body of the loop we see that `x` is incremented by `0.1` for each iteration of the loop. Thus it seems the loop should execute 10 times and then stop. However this is not the case. Recalling the discussion of Listing 11.10, we know that, because of the round-off error in `float`s, summing `0.1` ten times does not equal `1.0`. Thus the value of `x` is never equal to `1.0` and hence the loop does not stop of its own accord (we have to terminate the loop either by hitting control-C on a Mac or Linux machine or by typing control-Z followed by a return on a Windows machine).

If a `while`-loop's test expression is initially `True` and there is nothing done in the body of the loop to affect the test expression, the loop is an infinite loop. It is not uncommon to forget to write the code that affects the test expression. For example, assume we want to implement a

while-loop that displays a countdown from a given value to zero and then prints Blast off! A first attempt to implement this will often be written as

```
count = 10
while count >= 0:
    print(count, "...", sep="")
print("Blast off!")
```

The problem with this code is that count is never changed. A correct implementation is

```
count = 10
while count >= 0:
    print(count, "...", sep="")
    count = count - 1
print("Blast off!")
```

An infinite loop is often created intentionally and written as

```
while True:
    <body>
```

The test expression in the header clearly evaluates to True and there is nothing in the body of the loop that can affect this. However, the body will typically contain a break statement that is within the body of an if statement.[7] The existence of the break statement is used to ensure the loop is not truly infinite. And, in fact, no loop is truly infinite as a computer will run out of memory or the power will eventually be turned off. Nevertheless, when the test expression in the header of a while-loop will not cause the loop to terminate, we refer to the loop as an infinite loop.

Listing 11.25 demonstrates a function that can be used to implement the countdown described above using an infinite loop and a break statement. The function countdown() is defined in lines 1 through 7. It takes the single argument n which is the starting value for the count. The body of the function contains an infinite loop in lines 2 through 6. The loop starts by printing the value of n and then decrementing n. The if statement in line 5 checks if n is equal to -1. If it is, the break statement in line 6 is executed, terminating the loop. Note that a break only terminates execution of the loop. It is not a return statement. Thus, when we break out of the loop, the next statement to be executed is the print() statement in line 7. (Also, if one loop is nested within another and a break statement is executed within the inner loop, it will not break out of the outer loop.)

Listing 11.25 Use of an infinite loop to realize a finite "countdown" function.

```
>>> def countdown(n):
...     while True:
...         print(n, "...", sep="")
...         n = n - 1
```

[7] On the other hand, there are some programs that are designed to run continuously whenever your computer is on. For example, your system may continuously run a program which periodically queries a server to see if you have new email. We will not explicitly consider these types of functions.

```
5  ...          if n == -1:
6  ...              break
7  ...      print("Blast off!")
8  ...
9  >>> countdown(2)
10 2...
11 1...
12 0...
13 Blast off!
14 >>> countdown(5)
15 5...
16 4...
17 3...
18 2...
19 1...
20 0...
21 Blast off!
```

Now let's implement a new version of the code in Listing 11.24 that reads a list of names. In this new version, shown in Listing 11.26, we incorporate an infinite loop and a `break` statement. This new version eliminates the duplication of code that was present in Listing 11.24. In line 1 `names` is assigned to the empty `list`. The loop in lines 2 through 6 is an infinite loop in that the test expression will always evaluate to `True`. In line 3 the user is prompted for a name. If the user does not enter a name, i.e., if `input()` returns an empty string, the test expression of the `if` statement in line 4 will be `True` and hence the `break` in line 5 will be executed, thus terminating the loop. However, if the user does enter a name, the name is appended to `names` and the body of the loop is executed again. In lines 8 through 11 the user enters four names. In line 12 the user does not provide a name which terminates the loop. The `print()` statement in line 13 and the subsequent output on line 14 show the names have been placed in the `names` list.

Listing 11.26 Use of an infinite loop to obtain a `list` of names.

```
1  >>> names = []
2  >>> while True:
3  ...      name = input("Enter name [<ret> when done]: ")
4  ...      if not name:
5  ...          break
6  ...      names.append(name)
7  ...
8  Enter name [<ret> when done]: Laura
9  Enter name [<ret> when done]: Libby
10 Enter name [<ret> when done]: Linda
11 Enter name [<ret> when done]: Loni
12 Enter name [<ret> when done]:
13 >>> print(names)
14 ['Laura', 'Libby', 'Linda', 'Loni']
```

11.6.2 continue

There is one more useful statement for controlling the flow of loops. The `continue` statement dictates termination of the current iteration and a return to execution at the top of the loop, i.e., the test expression should be rechecked and if it evaluates to `True`, the body of the loop should be executed again.

For example, assume we again want to obtain a `list` of names, but with the names capitalized. If the user enters a name that doesn't start with an uppercase letter, we can potentially convert the string to a capitalized string ourselves. However, perhaps the user made a typo in the entry. So, rather than trying to fix the name ourselves, let's start the loop over and prompt for another name. The code in Listing 11.27 implements this function and is similar to the code in Listing 11.26. The difference between the two implementations appears in lines 6 through 8 of Listing 11.27. In line 6 we use the `islower()` method to check if the first character of the name is lowercase. If it is, the `print()` in line 7 is executed to inform the user of the problem. Then the `continue` statement in line 8 is executed to start the loop over. This ensures uncapitalized names are not appended to the `names` list. The remainder of the listing demonstrates that the code works properly.

Listing 11.27 A while loop that uses a `continue` statement to ensure all entries in the `names` list are capitalized

```
1  >>> names = []
2  >>> while True:
3  ...     name = input("Enter name [<ret> when done]: ")
4  ...     if not name:
5  ...         break
6  ...     if name[0].islower():
7  ...         print("The name must be capitalized.  Try again...")
8  ...         continue
9  ...     names.append(name)
10 ...
11 Enter name [<ret> when done]: Miya
12 Enter name [<ret> when done]: maude
13 The name must be capitalized.  Try again...
14 Enter name [<ret> when done]: Maude
15 Enter name [<ret> when done]: Mary
16 Enter name [<ret> when done]: mabel
17 The name must be capitalized.  Try again...
18 Enter name [<ret> when done]: Mabel
19 Enter name [<ret> when done]:
20 >>> print(names)
21 ['Miya', 'Maude', 'Mary', 'Mabel']
```

The `continue` statement can be used with `for`-loops as well. Let's consider one more example that again ties together many things we have learned in this chapter and in previous ones. Assume we want to write code that will read lines from a file. If a line starts with the hash symbol (#), it is taken to be a comment line. Comment lines are printed to the output and the rest of the

loop is skipped. Other lines are assumed to consist of numeric values separated by whitespace. There can be one or more numbers per line. For these lines the average is calculated and printed to the output.

To make the example more concrete, assume the following is in the file data.txt:

```
# Population (in millions) of China, US, Brazil, Mexico, Iceland.
1338 312 194 113 0.317
# Salary (in thousands) of President, Senator, Representative.
400 174 174
```

This file has two comment lines and two lines of numeric values. Line 2 has 5 values (giving the 2011 population in millions for five countries). Line 4 has three values giving the annual salaries (in thousands of dollars) for the President of the United States and members of the Senate and House of Representatives. Keep in mind that our code can make no assumptions about the number of values in a line (except that there must be at least one).

The code to process the file data.txt (or any file similar to data.txt) is shown in Listing 11.28. The file is opened in line 1. The for-loop starting in line 2 cycles through every line of the file. In line 3 the first character of the line is checked to see if it is a hash. If it is, the line is printed and the continue statement is used to start the loop over again, i.e., to get the next line of the file. If the line doesn't start with a hash, the split() method is used in line 6 to split the values in the line into individual strings which are stored in the list numbers. Recall that although the line contains numeric data, at this point in the code the data is in the form of strings. The for-loop in lines 8 and 9 cycles through all the numbers in the line, adding them to total which was initialized to zero in line 7. The float() function is used in line 9 to covert the strings to floats. After the total has been obtained, the print() statement in line 10 shows the average, i.e., the total divided by the number of values in the line. The result of processing the file is shown in lines 12 through 15.[8,9]

Listing 11.28 Code to process a file with comment lines and "data lines" where the number of items in a data line is not fixed. The numeric values in data lines are averaged.

```
>>> file = open("data.txt")
>>> for line in file:
...     if line[0] == '#':    # Comment? Echo line and skip rest of loop.
...         print(line, end="")
...         continue
...     numbers = line.split()   # Split line.
...     total = 0                # Initialize accumulator to zero.
...     for number in numbers:   # Sum all values.
...         total = total + float(number)
...     print("Average =", total / len(numbers)) # Print average.
...
```

[8]The numeric averages are a little "messy" and could be tidied using a format string, but we won't bother to do this.

[9]The built-in function sum() cannot be used to directly sum the elements in the list numbers since this list contains strings and sum() requires an iterable of numeric values.

```
12 # Population (in millions) of China, US, Brazil, Mexico, Iceland.
13 Average = 391.4634
14 # Salary (in thousands) of President, Senator, Representative.
15 Average = 249.33333333333334
```

11.7 Short-Circuit Behavior

The logical operators `and` and `or` are sometimes referred to as *short-circuit operators*. This has to do with the fact that Python will not necessarily evaluate both operands in a logical expression involving `and` and `or`. Python only evaluates as much as needed to determine if the overall expression is equivalent to `True` or `False`. Furthermore, these logical expressions don't necessarily *evaluate* to the literal Booleans `True` or `False`. Instead, they evaluate to the value of one operand or the other, depending on which operand ultimately determines whether the expression should be considered `True` or `False`. Exploiting the short-circuit behavior of the logical operators is a somewhat advanced programming technique. It is described here for three reasons: (1) short-circuit behavior can lead to bugs that can be extremely difficult to detect if you don't understand short-circuit behavior, (2) the sake of completeness, and (3) as you progress in your programming you are likely to encounter code that uses short-circuit behavior.

Let us first consider the short-circuit behavior of `and`. If the first operand evaluates to `False`, there is no need to evaluate the second operand because `False` `and`'ed with anything will still be `False`. To help illustrate this, consider the code in Listing 11.29. In line 1 `False` is `and`'ed with a call to the `print()` function. If `print()` is called, we see an output of `Hi`. However, Python doesn't call `print()` because it can determine this expression evaluates to False regardless of what the second operand returns. In line 3 there is another `and` expression but this time the first operand is `True`. So, Python must evaluate the second operand to determine if this expression should be considered `True` or `False`. The output of `Hi` on line 4 shows that the `print()` function is indeed called. But, what does the logical expression in line 3 evaluate to? The output in line 4 is rather confusing. Does this expression evaluate to the string `Hi`? The answer is no, and the subsequent lines of code, discussed below the listing, help explain what is going on.

Listing 11.29 Demonstration of the use of `and` as a short-circuit operator.

```
1  >>> False and print("Hi")
2  False
3  >>> True and print("Hi")
4  Hi
5  >>> x = True and print("Hi")
6  Hi
7  >>> print(x)
8  None
9  >>> if True and print("Hi"):
10 ...     print("The text expression evaluated to True.")
11 ... else:
12 ...     print("The text expression evaluated to False.")
```

```
13 ...
14 Hi
15 The text expression evaluated to False.
```

In line 5 the result of the same logical expression is assigned to the variable x. This assignment doesn't change the fact that the print() function is called which produces the output shown in line 6. In line 7 we print x and see that it is None. Where does this come from? Recall that print() is a void function and hence evaluates to None. For this logical expression Python effectively says, "I can't determine if this logical expression should be considered True or False based on just the first operand, so I will evaluate the second operand. I will use whatever the second operand returns to represent the value to which this logical expression evaluates." Hence, the None that print() returns is ultimately the value to which this logical expression evaluates. Recall that None is treated as False. So, if this (rather odd) logical expression is used in a conditional statement, as done in line 9, the test expression is considered to be False and hence only the else portion of this if-else statement is executed as shown by the output in line 15. The Hi that appears in line 14 is the result of the print() function being called in the evaluation of the header in line 9.

To further illustrate the short-circuit behavior of and, consider the code in Listing 11.30. The short-circuit behavior of and boils down to this: If the first operand evaluates to something that is equivalent to False, then this is the value to which the overall expression evaluates. If the first operand evaluates to something considered to be True, then the overall expression evaluates to whatever value the second operand evaluates. In line 1 of Listing 11.30 the first operand is an empty list. Since this is considered to be False, it is assigned to x (as shown in lines 2 and 3). In line 4 the first operand is considered to be True. Thus the second operand is evaluated. In this case the second operand consists of the expression 2 + 2 which evaluates to 4. This is assigned to x (as shown in lines 5 and 6).

Listing 11.30 Using the short-circuit behavior of the and operator to assign one value or the other to a variable.

```
1 >>> x = [] and 2 + 2
2 >>> x
3 []
4 >>> x = [0] and 2 + 2
5 >>> x
6 4
```

The short-circuit behavior of the or operator is similar to, but essentially the converse of, the short-circuit behavior of and. In the case of or, if the first operand is effectively True, there is no need to evaluate the second operand (since the overall expression will be considered to be True regardless of the second operand). On the other hand, if the first operand is considered to be False, the second operand must be evaluated. The value to which an or expression evaluates is the same as the operand which ultimately determines the value of the expression. Thus, when used in assignment statements, the value of the first operand is assigned to the variable if this first operand is effectively True. If it is not, the value of the second operand is used. This is

illustrated in Listing 11.31. In line 1 the two operands of the or operator are 5 + 9 and the string hello. Both these operands are effectively True (since the first operand evaluates to 14 which is non-zero). However, Python never "sees" the second operand because it knows the outcome of the logical expression simply from the first operand. The logical expression thus evaluates to 14 which is assigned to x (as shown in lines 2 and 3). Line 4 differs from line 1 in that the first operand is now the expression 5 + 9 - 14. This evaluates to zero and thus the output of the logical expression hinges on the value to which the second operand evaluates. Thus the value to which the entire logical expression evaluates is the string hello.

Listing 11.31 Demonstration of the short-circuit behavior of the or operator.

```
1 >>> x = 5 + 9 or "hello"
2 >>> x
3 14
4 >>> x = 5 + 9 - 14 or "hello"
5 >>> x
6 'hello'
```

It may not be obvious where one would want to use the short-circuit behavior of the logical operators. As an example of their utility, assume we want to prompt a user for a name (using the input() function), but we also want to allow the user to remain anonymous by simply hitting return at the input prompt. When the user doesn't enter a name, input() will return the empty string. Recall that the empty string is equivalent to False. Given this, the code in Listing 11.32 demonstrates how we can prompt for a name and provide a default of Jane Doe if the user wishes to remain anonymous. In line 1 the or operator's first operand is a call to the input() function. If the user enters anything, this will be the value to which the logical expression evaluates. However, if the user does not provide a name (i.e., input() returns the empty string), then the logical expression evaluates to the default value Jane Doe given by the second operand. When line 1 is executed, the prompt is produced as shown in line 2. In line 2 we also see the user input of Mickey Mouse. Lines 3 and 4 show that this name is assigned to name. Line 5 is the same as the statement in line 1. Here, however, the user merely types return at the prompt. In this case name is assigned Jane Doe as shown in lines 7 and 8.

Listing 11.32 Use of the short-circuit behavior of the or operator to create a default for user input.

```
1 >>> name = input("Enter name: ") or "Jane Doe"
2 Enter name: Mickey Mouse
3 >>> name
4 'Mickey Mouse'
5 >>> name = input("Enter name: ") or "Jane Doe"
6 Enter name:
7 >>> name
8 'Jane Doe'
```

Of course, we can provide a default without the use of a short-circuit operator. For example, the following is equivalent to lines 1 and 5 of Listing 11.32:

```
name = input("Enter name: ")
if not name:
    name = "Jane Doe"
```

This code is arguably easier to understand than the implementation in Listing 11.32. However, since it requires three lines as opposed to the single statement in Listing 11.32, many experienced programmers opt to use the short-circuit approach.

Let's assume you have decided not to exploit the short-circuit behavior of the logical operators. However, there is a chance that this behavior can inadvertently sneak into your code. For example, assume you have a looping structure and for each iteration of the loop you "ask" the user if the program should execute the loop again. If the user wants to continue, they should enter a string such as `Yes`, `yep`, or simply `y`, i.e., anything that starts with an uppercase or lowercase y. Any other response means the user does not want to continue. An attempt to implement such a loop is shown in Listing 11.33. The code is discussed after the listing.

Listing 11.33 A flawed attempt to allow the user to specify whether or not to continue executing a `while`-loop. Because of the short-circuit behavior of the `or` operator this is an infinite loop.

```
>>> response = input("Continue? [y/n] ")
Continue? [y/n] y
>>> while response[0] == 'y' or 'Y':    # Flawed test expression.
...     print("Continuing...")
...     response = input("Continue? [y/n] ")
...
Continuing...
Continue? [y/n] Yes
Continuing...
Continue? [y/n] No
Continuing...
Continue? [y/n] Stop!
Continuing...
Continue? [y/n] Quit!
Continuing...
    .
    .
    .
```

In line 1 the user is prompted as to whether the loop should continue (at this point, the prompt is actually asking whether to enter the loop in the first place). Note that the user is prompted to enter `y` or `n`. However, in the interest of accommodating other likely replies, we want to "secretly" allow any reply and will treat replies that resemble yes to be treated in the same way as a reply of `y`. Thus, the test expression in line 3 uses the first character of the response and appears to ask if this letter is `y` or `Y`. *However*, this is not actually what the code does. Recall that logical operators

have lower precedence than comparison operators. Thus, response[0] is compared to y. If this evaluates to False, the or operator will return the second operand, i.e., the test expression evaluates to Y. As the character Y is a non-empty string, it is considered to be True. Therefore the test expression is always considered to be True! We have implemented an infinite loop. This is demonstrated starting in line 7 and continuing through the end of the listing. We see that even when the user enters No or Stop! or anything else, the loop continues.

There are various ways to correctly implement the header of the while-loop in Listing 11.33. Here is one approach that explicitly compares the first character of the response to y and Y:

```
while response[0] == 'y' or response[0] == 'Y':
```

Another approach is to use the lower() method to ensure we have the lowercase of the first character and then compare this to y:

```
while response[0].lower() == 'y':
```

11.8 The in Operator

Two comparison operators were omitted from the listing in Listing 11.9: in and is. The is operator is discussed in Sec. 7.3. It returns True if its operands refer to the same memory and returns False otherwise. The is operator can be a useful debugging or instructional tool, but otherwise it is not frequently used. In contrast to this, the in operator provides a great deal of utility and is used in a wide range of programs. The in operator answers the question: is the left operand contained in the right operand? The right operand must be a "collection," i.e., an iterable such as a list or tuple.

We can understand the operation of the in operator by writing a function that mimics its behavior. Listing 11.34 defines a function called my_in() that duplicates the behavior of the in operator. The only real difference between my_in() and in is that the function takes two arguments whereas the operator is written between two operands. The function is defined in lines 1 through 5. It has two parameters called target and container. The goal is to return True if target matches one of the (outer) elements of container. (If container has other containers nested inside it, these are not searched.) If target is not found, the function returns False. The body of the function starts with a for-loop which cycles over the elements of container. The if statement in line 3 tests whether the element matches the target. (Note that the test uses the "double equal" comparison operator.) If the target and element are equal, the body of the if statement is executed and returns True, i.e., the function is terminated at this point and control returns to the point of the program at which the function was called. If the for-loop cylces through all the elements of the container without finding a match, we reach the last statement in the body of the function, line 5, which simply returns False. Discussion of this code continues following the listing.

Listing 11.34 The function my_in() duplicates the functionality provided by the in operator.

```
1 >>> def my_in(target, container):
```

```
2  ...     for item in container:
3  ...         if item == target:
4  ...             return True
5  ...     return False
6  ...
7  >>> xlist = [7, 10, 'hello', 3.0, [6, 11]]
8  >>> my_in(6, xlist)          # 6 is not in list.
9  False
10 >>> my_in('hello', xlist)  # String is found in list.
11 True
12 >>> my_in('Hello', xlist)  # Match is case sensitive.
13 False
14 >>> my_in(10, xlist)         # Second item in list.
15 True
16 >>> my_in(3, xlist)          # Integer 3 matches float 3.0.
17 True
```

In line 7 the `list` `xlist` is created with two integers, a string, a `float`, and an embedded `list` that contains two integers. In line 8 `my_in()` is used to test whether `6` is in `xlist`. The integer `6` is contained within the `list` embedded in `xlist`. However, since the search is only performed on the outer elements of the container, line 9 reports that `6` is not in `xlist`. In line 10 we check whether `hello` is in `xlist`. Line 11 reports that it is. Matching of strings is case sensitive as lines 12 and 13 demonstrate. In line 16 we ask whether the integer `3` is in `xlist`. Note that `xlist` contains the `float` `3.0`. Nevertheless, these are considered equivalent (i.e., the == operator considers `3` and `3.0` to be equivalent).

Listing 11.35 defines the same values for `xlist` as used in Listing 11.34. The statements in lines 2 through 11 perform the same tests as performed in lines 8 through 17 of Listing 11.34 except here the built-in `in` operator is used. Line 12 shows how we can check whether a target is *not* in the container. Note that we can write `not in` which is similar to how we express this query in English. However, we can also write this expression as shown in line 14. (The statement in line 12 is the preferred idiom.)

Listing 11.35 Demonstration of the use of the `in` operator.

```
1  >>> xlist = [7, 10, 'hello', 3.0, [6, 11]]
2  >>> 6 in xlist
3  False
4  >>> 'hello' in xlist
5  True
6  >>> 'Hello' in xlist
7  False
8  >>> 10 in xlist
9  True
10 >>> 3 in xlist
11 True
12 >>> 22 not in xlist     # Check whether target is not in container.
```

```
13 True
14 >>> not 22 in xlist     # Alternate way to write previous expression.
15 True
```

To demonstrate one use of `in`, let's write a function called `unique()` that accepts any iterable as an argument. The function returns a `list` with any duplicates removed from the items within the argument. Listing 11.36 defines the function in lines 1 through 6. The argument to the function is the "container" `dups` (which can be any iterable). In line 2 the `list` `no_dups` is initialized to the empty `list`. The `for`-loop in lines 3 through 5 cycles through all the elements of `dups`. The `if` statement in lines 4 and 5 checks whether the element is *not* currently in the `no_dups` `list`. If it is not, the item is appended to `no_dups`. After cycling through all the elements in `dups`, the `no_dups` `list` is returned in line 6. The discussion continues following the listing.

Listing 11.36 Use the `in` operator to remove duplicates from a container.

```
1  >>> def unique(dups):
2  ...     no_dups = []
3  ...     for item in dups:
4  ...         if item not in no_dups:
5  ...             no_dups.append(item)
6  ...     return no_dups
7  ...
8  >>> xlist = [1, 2, 2, 2, 3, 5, 2]
9  >>> unique(xlist)
10 [1, 2, 3, 5]
11 >>> unique(["Sue", "Joe", "Jose", "Jorge", "Joe", "Sue"])
12 ['Sue', 'Joe', 'Jose', 'Jorge']
```

In line 8 the `list` `xlist` is defined with several copies of the integer `2`. When this is passed to `unique()` in line 9, the `list` that is returned has no duplicates. In line 11 the `unique()` function is passed a `list` of strings with two duplicate entries. The result in line 12 shows these duplicates have been removed.

11.9 Chapter Summary

Python provides the Boolean literals **`True`** and **`False`**.

When used in a conditional statement, all objects are equivalent to either `True` or `False`. Numeric values of zero, empty containers (for example, empty strings and empty `lists`), `None`, and `False` itself are considered to be `False`. All other objects are considered to be `True`.

The **`bool()`** function returns either `True` or `False` depending on whether its argument is equivalent to `True` or `False`.

The template for an **`if`** statement is

```
if <test_expression>:
    <body>
```

The body is executed only if the object returned by the test expression is equivalent to `True`.

`if` statements may have **`elif`** clauses and an **`else`** clause. The template for a general conditional statement is

```
if <test_expression1>:
    <body1>
elif <test_expression2>:
    <body2>
... # Arbitrary number
... # of elif clauses.
else:
    <bodyN>
```

The body associated with the first test expression to return an object equivalent to `True` is executed. No other body is executed. If none of the test expressions returns an object equivalent to `True`, the body associated with the `else` clause, when present, is executed. The `else` clause is optional.

The comparison, or relational, operators compare the values of two operands and return `True` if the implied relationship is true. The comparison operators are: less than (**`<`**), less than or equal to (**`<=`**), greater than (**`>`**), greater than or equal to (**`>=`**), equal to (**`==`**), and not equal to (**`!=`**).

The logical operators **`and`** and **`or`** take two operands. `and` produces a `True` value only if both its operands are equivalent to `True`. `or` produces a `True` value if either or both its operands are equivalent to `True`. The logical operator **`not`** is a unary operator that negates the value of its operand.

All comparison operators have higher precedence than logical operators. `not` has higher precedence than `and`, and `and` has higher precedence than `or`. Parentheses can be used to change the order of precedence of these operators. All math operators have higher precedence than both comparison and logical operators.

`and` and `or` use "shortcircuit" behavior. In expressions involving `and` or `or`, Python only evaluates as much as needed to determine the final outcome. The return value is the object that determines the outcome.

The template for a **`while`**-loop is:

```
while <test_expression>:
    <body>
```

The test expression is checked. If it is equivalent to `True`, the body is executed. The test expression is checked again and the process is repeated.

A **`break`** statement terminates the current loop.

A **`continue`** statement causes the remainder of a loop's body to be skipped in the current iteration of the loop.

Both `break` and `continue` statements can be used with either `for`-loops or `while`-loops.

The **`in`** operator returns `True` if the left operand is contained in the right operand and returns `False` otherwise.

11.10 Review Questions

1. Consider the following code. When prompted for input, the user enters the string `SATURDAY`. What is the output?

```
day = input("What day is it? ")
day = day.lower()
```

```
if day == 'saturday' or day == 'sunday':
   print("Play!")
else:
   print("Work.")
```

2. Consider the following code. When prompted for input, the user enters the string `monday`. What is the output?

```
day = input("What day is it? ")
day = day.lower()
if day != 'saturday' and day != 'sunday':
   print("Yep.")
else:
   print("Nope.")
```

3. Consider the following code. What is the output?

```
values = [-3, 4, 7, 10, 2, 6, 15, -300]
wanted = []
for value in values:
    if value > 3 and value < 10:
        wanted.append(value)

print(wanted)
```

4. What is the output generated by the following code?

```
a = 5
b = 10
if a < b or a < 0 and b < 0:
  print("Yes, it's true.")
else:
  print("No, it's false.")
```

5. What is the value of `x` after the following code executes?

```
x = 2 * 4 - 8 == 0
```

   (a) `True`
   (b) `False`
   (c) None of the above.
   (d) This code produces an error.

6. What is the output generated by the following code?

```
a = 5
b = -10
if a < b or a < 0 and b < 0:
  print("Yes, it's true.")
else:
  print("No, it's false.")
```

7. What is the output generated by the following code?

```
a = -5
b = -10
if (a < b or a < 0) and b < 0:
  print("Yes, it's true.")
else:
  print("No, it's false.")
```

8. What is the output produced by the following code?

```
a = [1, 'hi', False, '', -1, [], 0]
for element in a:
   if element:
      print('T', end=" ")
   else:
      print('F', end=" ")
```

9. Consider the following conditional expression:

```
x > 10 and x < 30
```

Which of the following is equivalent to this?

(a) `x > 10 and < 30`

(b) `10 < x and 30 > x`

(c) `10 > x and x > 30`

(d) `x <= 10 or x >= 30`

10. To what value is `c` set by the following code?

```
a = -3
b = 5
c = a <= (b - 8)
```

(a) `True`

(b) `False`

(c) This code produces an error.

11. What is the output produced by the following code?

```
def is_lower(ch):
   return 'a' <= ch and ch <= 'z'

print(is_lower("t"))
```

(a) `True`
(b) `False`
(c) `None`
(d) This code produces an error

12. What is the output produced by the following code?

```
def is_there(names, query):
   for name in names:
      if query == name:
         return True

print(is_there(['Jake', 'Jane', 'Alice'], 'Tom'))
```

(a) `True`
(b) `False`
(c) `None`
(d) This code produces an error.

13. What output is produced by the following code?

```
def monotonic(xlist):
    for i in range(len(xlist) - 1):
        if xlist[i] < xlist[i + 1]:
            return False
    return True

data1 = [5, 3, 2, 2, 0]
data2 = [5, 2, 3, 2, 0]
print(monotonic(data1), monotonic(data2))
```

(a) `True True`
(b) `True False`
(c) `False True`

(d) `False False`

(e) None of the above.

14. What output is produced by the following code?

```
def swapper(xlist):
    for i in range(len(xlist) - 1):
        if xlist[i] > xlist[i + 1]:
            # Swap values.
            xlist[i], xlist[i + 1] = xlist[i + 1], xlist[i]

data = [5, 3, 2, 2, 0]
swapper(data)
print(data)
```

15. What is the value of `x` after the following code executes?

```
y = 10
x = 2 * 4 - 8 or y
```

(a) `True`

(b) `False`

(c) None of the above.

(d) This code produces an error.

16. What is the value of `x` after the following code executes?

```
y = 10
if 2 * 4 - 8:
    x = 2 * 4 - 8
else:
    x = y
```

(a) `True`

(b) `False`

(c) `0`

(d) `10`

17. What is the value of `x` after the following code executes?

```
x = 4
while x > 0:
    print(x)
    x = x - 1
```

(a) 4
(b) 1
(c) 0
(d) -1
(e) None of the above.

18. What is the value of x after the following code executes?

```
x = 4
while x == 0:
    print(x)
    x = x - 1
```

(a) 4
(b) 1
(c) 0
(d) -1
(e) None of the above.

19. What is the value returned by the function func1() when it is called in the following code?

```
def func1(xlist):
    for x in xlist:
        if x < 0:
            return False
    return True

func1([5, 2, -7, 7])
```

(a) True
(b) False
(c) None of the above.
(d) This code produces an error.

20. What is the value returned by the function func2() when it is called in the following code?

```
def func2(xlist):
    for i in range(len(xlist) - 1):
        if xlist[i] + xlist[i + 1] == 0:
            return True
    return False

func2([5, 2, -7, 7])
```

(a) `True`

(b) `False`

(c) None of the above.

(d) This code produces an error.

ANSWERS: 1) `Play!`; 2) `Yep.`; 3) `[4, 7, 6]`; 4) `Yes, it's true.`; 5) a; 6) `No, it's false.`; 7) `Yes, it's true.`; 8) `T T F F T F F`; 9) b; 10) a; 11) a; 12) c; 13) b; 14) `[3, 2, 2, 0, 5]`; 15) c (it is `10`); 16) d; 17) c; 18) a; 19) b; 20) a.

Chapter 12

Recursion

12.1 Background

Recursion is a powerful programming construct that provides an elegant way to solve large problems by breaking them down into simpler subproblems. In this way it is often possible to solve seemingly complex problems via repeated application of very simple solutions to the subproblems.

In programming languages, when we talk about recursion we talk in terms of recursive functions.[1] A recursive function is a function that calls itself somewhere in its definition, i.e., in some sense, the function is defined in terms of itself. At first this may seem confusing. After all, if we use a word to define itself, for example, if we say, "a tautology is a tautology," then we haven't helped to clarify what a tautology is.[2] So, clearly there must be something more to a useful recursive function than the mere fact that it uses itself within its own definition.

12.2 Flawed Recursion

We will consider the proper implementation of a recursive function in a moment, but let's start by considering some flawed implementations because it is important to recognize how you must *not* implement a recursive function. You could easily write a function that uses itself in its own definition and hence is arguably a recursive function. Consider the function `r1()` defined in Listing 12.1 which is the simplest possible (albeit flawed) implementation of a recursive function. This function has no arguments and the entire body of the function is merely a call to the function itself. Thus, when we call `r1()`, the body of the function says to call the function `r1()`. So, `r1()` will be called again only to find that `r1()` should be called again, and so on. In theory, these successive calls would go on forever, never doing anything useful. However, in practice, these successive calls can't go on forever; something must ultimately break if `r1()` is called (for example, it requires some memory to keep track of function calls so perhaps your computer ultimately runs out of memory).

From the file: `recursion.tex`

[1]Not all computer languages support recursive functions, but nearly all modern languages do.

[2]A tautology is, according to Webster's Collegiate Dictionary, 5th ed., "needless repetition of meaning in other words; also, an instance of this as 'audible to the ear'."

Listing 12.1 Implementation of the simplest (albeit flawed) recursive function. The body of the function is merely a call to the function itself.

```
1 >>> def r1():
2 ...     r1()
```

Listing 12.2 demonstrates what happens when we call `r1()`. In line 1 `r1()` is called. This produces the `Traceback` message which contains hundreds of repeated lines (most of which have been removed from the listing). This is followed by a statement telling us there was a `RuntimeError` because the "maximum recursion depth [was] exceeded." Essentially Python kept track of how many times `r1()` called `r1()`. Once the number of calls exceeded a certain value, Python stepped in, thinking that something must be wrong, and halted the entire process.

Listing 12.2 Calling the recursive function `r1()` produces an error because the maximum recursion depth is exceeded.

```
1 >>> r1()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 2, in r1
5   File "<stdin>", line 2, in r1
6                     .
7                     .   <<OUTPUT DELETED>>
8                     .
9 RuntimeError: maximum recursion depth exceeded
```

Let's implement another (flawed) recursive function but add a bit of code to the function's body to help shed light on what is happening. Listing 12.3 starts by defining the function `r2()` that takes a single argument (which is assumed to be an integer). In line 2, the first statement in the body of the function prints the value of the argument together with the word `Start`. Then, in the second statement of the body (line 3), `r2()` is called *but* the argument passed to `r2()` is incremented by one from the value that was originally passed to the function. The last statement in the body of the function (line 4) again prints the argument but now it is paired with the word `End`. In line 6 `r2()` is called with an argument of `0`. The subsequent output is discussed following the listing.

Listing 12.3 Another flawed recursion function. This function accepts an argument and increments the value of the argument with each successive recursive call.

```
1 >>> def r2(n):
2 ...     print(n, "Start")
3 ...     r2(n + 1)
4 ...     print(n, "End")
5 ...
```

```
6  >>> r2(0)
7  0 Start
8  1 Start
9  2 Start
10  .
11  .  <<OUTPUT DELETED>>
12  .
13 994 Start
14 995 Start
15 996 Start
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18   File "<stdin>", line 3, in r2
19   File "<stdin>", line 3, in r2
20                   .
21                   .  <<OUTPUT DELETED>>
22                   .
23   File "<stdin>", line 2, in r2
24 RuntimeError: maximum recursion depth exceeded while calling a Python
25 object
```

In line 7 of Listing 12.3, we see that when `r2()` is called with an argument of `0`, the first output that is produced is `0 Start`, i.e., the first statement in the body of the function (line 2) merely prints whatever argument was passed to the function. Then, in line 3, `r2()` is called again (i.e., `r2()` is called from within `r2()`, before the flow of execution returned from the first call to `r2()`). But, the argument is incremented by one. Thus, for this second call to `r2()`, the argument is `1` which results in the output shown on line 8 (`1 Start`. After printing this argument, `r2()` is called again, but the argument is again incremented by one. This leads to the output of `2 Start` shown on line 9. This cycle repeats until, ultimately, Python generates the `RuntimeError` shown in lines 16 to 22 (portions of the output have been deleted).[3] Importantly, notice that we never got to the second `print()` statement in the body of the function, i.e., the statement in line 4 that prints the argument followed by the word `End`. That is because we never *returned* from any of the calls to `r2()` before Python halted the process.

12.3 Proper Recursion

Now we can ask: What causes the errors shown in Listings 12.1 and 12.3? Thinking about how `r1()` and `r2()` are defined, we see that they "never" stop calling themselves, i.e., they keep calling themselves until the recursion limit is reach. When this limit is reached one obtains the `RuntimeError` error messages shown in the listings. To prevent this error and to make a useful recursive function, *a recursive function must possess two vital features*:

[3]On the system on which this particular example was run, the maximum recursion depth was set to 1000. This is true despite the fact that there are 997 integer values displayed in the output. You can obtain the recursion limit with the following two statements: `import sys; print(sys.getrecursionlimit())`. The function `getrecursionlimit()` in the `sys` module is used to set the recursion limit to a different value.

1. There must be a reachable *base case* where the function stops calling itself.

2. The argument of the function must be modified with each call.

Note that `r1()` in Listing 12.1 possessed neither of these features. On the other hand, `r2()` in Listing 12.3 possessed the second feature (the argument was modified with each call), but it did not possess a base case.

The base case specifies when to stop making recursive calls. In order to do this, we need to use what we've learned about conditionals to write the base case and the general case. As will become more clear in a little while, you can think of a base case as the simplest subproblem you have, for which the answer is easily determined while the general case is where we modify the argument to ensure that the base case is reachable. But, before doing anything particularly useful in terms of solving a problem, let's create a modified version of `r2()` so that it has both features necessary to realize a useful recursive function.

Listing 12.4 defines the function `r3()` which is similar to `r2()` except that, as shown in lines 3 and 4, it will only make a recursive call to `r3()` if `n` is less than 4. For this function the "base case" is when the function is passed an argument of 4 (or greater). When this occurs, instead of calling `r3()` again, the function merely prints the start and end messages and then returns. The "general case" results in the printing of the start message, the recursive call to `r3()`, and then the printing of the end message. The discussion continues following the listing.

Listing 12.4 Recursive function that has a (reachable) base case and modifies its argument with each successive call.

```
1  >>> def r3(n):
2  ...     if n < 4:      # General case.
3  ...         print(n, "Start")
4  ...         r3(n + 1)
5  ...         print(n, "End")
6  ...     else:          # Base case.
7  ...         print(n, "Start")
8  ...         print(n, "End")
9  ...
10 >>> r3(0)
11 0 Start
12 1 Start
13 2 Start
14 3 Start
15 4 Start
16 4 End
17 3 End
18 2 End
19 1 End
20 0 End
```

In line 10 `r3()` is called with an argument of 0. Because this argument is less than 4, this leads to the "general-case behavior" of the function. Thus, in accordance with the statement in line

3, the start message is printed (as shown in line 11). Then, in line 4, a recursive call is made to `r3()` but the argument is incremented so that it is now 1. Because an argument of 1 is less than 4, this call to `r3()` again results in the general-case behavior. This ultimately produces all the start messages shown in lines 11 through 14.

When `r3()` is called with an argument of 4, we have reached the base case (which corresponding to the statements in lines 7 and 8). The start message is again printed, as shown in line 15, i.e., `4 Start`. But then, rather than calling `r3()` again, the function prints the end message (shown in line 16) and then returns. The flow of execution returns to the point where this function was called. This happens to be from `r3()` but where the arguments was 3. So, now that function prints the end message (i.e., `3 End`) and returns. Execution now returns to prior invocation of `r3()` where the argument was 2, and so on, until we ultimately get back to where `r3()` was called with an argument of 0 (and thus back to the interactive prompt).

Again, `r3()` is a proper recursive function in that it has a reachable base case (where the function stops making calls to itself) and the argument is modified with each call. You should spend a bit of time looking at the output in Listing 12.4. Note that both the *first* line of output and the *last* line of output are associated with the call to `r3()` with an argument of zero. This call to `r3()` executes a `print()` statement, then `r3()` is called so that nothing else is executed from this original call to `r3()` until all the other calls to `r3()` are complete. Once they are complete and control is returned to the first call to `r3()`, the final `print()` statement is executed.

The implementation of the `r3()` function in Listing 12.4 separates the general case and the base case into two distinct blocks of code. However, because there is an overlap in what the general case and the base case actually do, there is an needless duplication of code. A cleaner implementation is shown in Listing 12.5.

Listing 12.5 A alternate implementation of the `r3()` function of 12.4 that removes the unnecessary duplication of code.

```
1 def r3(n):
2     print(n, "Start")
3     if n < 4:
4         r3(n + 1)
5     print(n, "End")
```

In mathematics you will often see the Fibonacci sequence defined recursively, so let's consider an implementation of that. Assume $F(n)$ yields the nth Fibonacci number. $F(n)$ can be defined as follows[4][5]

$$\begin{aligned} F(1) &= 1 \\ F(2) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

[4]The implementation of the Fibonacci sequence given in Sec. 6.9.3 is actually much more efficient than the recursive implementation we will present here. We are defining it recursively here for purposes of illustration.

[5]Sometimes the sequence is defined with the first two numbers being 0 and 1 instead of 1 and 1.

The Fibonacci function $F(n)$ is defined in terms of itself. The third line above is the general case and says that any number in the sequence is the sum of the previous two numbers in the sequence. There are actually two possible base cases. Both $F(1)$ and $F(2)$ simply return the number 1. Using this definition let's write, as shown in Listing 12.6, a function that uses recursion to generate numbers in the Fibonacci sequence.

Listing 12.6 Recursive implementation of a function to generate numbers in the Fibonacci sequence.

```
1 >>> def fib(n):
2 ...     if n == 1 or n == 2:          # Base case.
3 ...         return 1
4 ...     return fib(n - 1) + fib(n - 2) # General case.
```

Using the mathematical definition above, we can easily determine the base case: if n is 1 or 2, then the function merely returns 1. For the general case we want to return the sum of the previous two numbers in the sequence, i.e., the sum of the $(n-1)$th number and the $(n-2)$th number. Since `fib(n)` generates the nth number in the sequence, we can call `fib()` with arguments of `n - 1` and `n - 2` and return the sum of the results.

These separate calls to `fib()` will in turn make more calls to `fib()` until the base case is reached. Once this happens, the results will build from the base case to find the $(n-1)$th and $(n-2)$th Fibonacci number, and then, from these, the nth number. Note that the body of the function definition in Listing 12.6 looks almost identical to the mathematical definition! Note that our implementation does meet the requirements for a useful recursive function: there is a reachable base case and the argument is modified for each recursive call.[6] Listing 12.7 demonstrates the behavior of this recursive implementation of a Fibonacci number generator.

Listing 12.7 Demonstration of the behavior of the `fib()` function as implemented in Listing 12.6.

```
1 >>> fib(1)
2 1
3 >>> fib(2)
4 1
5 >>> fib(3)
6 2
7 >>> fib(10)
8 55
```

In lines 1 and 3 of Listing 12.7, `fib()` is called with arguments that result in base-case behavior where the function immediately returns 1. When `fib()` is called with an argument of 3,

[6]However, it may also be worth pointing out that the base case is only reachable if the function is called with a positive argument. You may wish to confirm for yourself that if the function is called with a non-positive argument, the subsequent recursive calls have arguments that progressively become more negative until eventually the recursion limit is reached.

as is done in line 5, the return value is the sum of `fib(1)` and `fib(2)`, i.e., the two base-case values, which is simply 2. In line 7 `fib()` is called with an argument of 10. Behind the scenes this actually triggers a rather large cascade of recursive calls. In fact, the argument does not have to be large before the computer takes a noticeably long time to return a value. Thus, as mentioned in Footnote 4 on page 301, a purely recursive implementation of a Fibonacci number generator is not efficient. It is much more efficient to start from the base values and then, in an iterative loop (such as a `for`-loop), directly build up to the number n rather than starting from n and recursively working down to the base values.

Now let's consider a recursive implementation of the factorial function (in this case the recursive implementation is nearly as efficient as an iterative [non-recursive] implementation). Recall that $n!$, read as n factorial, is defined as $n \times (n-1) \times (n-2) \cdots \times 1$. Listing 12.8 shows both a recursive and a non-recursive implementation of the factorial function.

Listing 12.8 Recursive and non-recursive implementation of the factorial function.

```
1  >>> def fact_r(n):   # Recursive implementation.
2  ...     if n == 1:                # Base case.
3  ...         return 1
4  ...     return n * fact_r(n - 1) # General case.
5  ...
6  >>> fact_r(1)
7  1
8  >>> fact_r(5)
9  120
10 >>> fact_r(40)
11 815915283247897734345611269596115894272000000000
12 >>> def fact_nr(n): # Non-recursive implementation.
13 ...     fact = 1
14 ...     for i in range(1, n + 1):
15 ...         fact = fact * i
16 ...     return fact
17 ...
18 >>> fact_nr(40)
19 815915283247897734345611269596115894272000000000
```

The recursive function is defined in lines 1 through 4. The base case for this function is when the argument is 1, in which case the function returns 1. This is handled by the first two lines in the body of the function, i.e., lines 2 and 3. Thus, if 1 is supplied as an argument then we can simply return 1 as the answer. The fact that the function does this is demonstrated in lines 6 and 7. Recalling the definition of $n!$ we recognize that $n!$ is equal to $n \times (n-1)!$. We can use this fact to define our general case by multiplying the argument n by a recursive call to `fact_r` with an argument of $(n-1)$ as is done in the last statement in the body of the function. Lines 8 through 11 demonstrate that the function works properly for arguments that do not immediately trigger the base case.

A non-recursive implementation of the factorial function is given in lines 12 through 16 of Listing 12.8. In the body of function the identifier `fact` is assigned the initial value of 1. Then, in

lines 14 and 15, fact is used as an accumulator in a for-loop to accumulate all the multiplication needed to build up from 1 to n. The last two lines of the listing demonstrate the function works properly. The non-recursive implementation is actually slightly more efficient than the recursive implementation because calls to a function are typically more computationally expensive than iterating a for-loop.

Recursion is applicable to more than just mathematical problems. For example, consider the function in Listing 12.9 that returns the reverse of the string it was passed.

Listing 12.9 Recursive function to reverse a string.

```
1 >>> def reverse(s):
2 ...     if s == "":
3 ...         return ""
4 ...     return s[-1] + reverse(s[ : -1])
```

Inspect this code and then think about how you might describe, in words, the solution. How can we use recursion to reverse the characters of a string and what is the base case? We know that in order to reverse a string recursively we're going to have to break it down into subproblems of the same form, but when should we stop trying to reverse the string? The answer is: when there isn't anything left to reverse, i.e., when the string we want to reverse is empty. If the string is empty then we can simply return an empty string because there is nothing to reverse.[7]

The more challenging part of the string-reversal problem is coming up with the general case. Think of this in terms of adding something to the answer of a smaller subproblem that has already been solved. We know that the first character in a reversed string is the last character of the given string (e.g., the first character in the reverse of cat is t because t is the last character of the given string [and will ultimately be the first character of tac]). The first character of the reversed string can be easily obtained using negative indexing on the given string, i.e., using s[-1]. In order to get a complete reversed string, we concatenate the last character of the string with the reverse of the rest of the string, i.e., the reverse of the string after removing the last character. This suggests a recursive solution. We are defining a function that gives us the reverse of the string so we can call this function to find the reverse of the rest of the string! We just have to ensure that the argument we pass to the function is the string with the final character removed, i.e., s[: -1]. So, the recursive implementation could be described as, "Write the last character and then write the remaining characters in reverse order." Listing 12.10 demonstrates that the function works properly.

Listing 12.10 Demonstration that the recursive function reverse() defined in Listing 12.9 works properly.

```
1 >>> reverse("recursion")
2 'noisrucer'
3 >>> reverse("Hello there.")
4 .ereht olleH
```

[7]An alternate base case would be to check for a string of length 1. In that case the function should return that single-character string since one cannot reverse a single character.

This same recursive approach can be used to reverse a list, except for a list the base case would be when the list had a length of zero and the return value would be an empty list.

There are, in fact, many ways to reverse a string (or a list). The easiest and clearest implementation is simply to use a slice with a increment of -1, i.e., the reverse of the string s is the string s[: : -1]. A solution that requires slightly more code, but is still simpler than the recursive approach, is to use a for-loop where the index would pick out the characters of the string in reverse order. However, for the sake of illustration, we will consider one more recursive function that reverses the characters of a string.

Listing 12.11 defines two functions. The first, reverse_main(), in lines 1 and 2, takes a single string argument and is not a recursive function. It merely returns whatever the function reverse_help() returns when that function is called with two arguments: the same string as was passed to reverse_main() and the integer 0. The discussion continues following the listing.

Listing 12.11 Another recursive solution to the string reversal problem. In this case the solution is broken into two functions: a "main" function that takes a single string argument and a "help" function that takes two arguments corresponding to the string and what is effectively an integer index.

```
1  >>> def reverse_main(s):
2  ...     return reverse_help(s, 0)
3  ...
4  >>> def reverse_help(s, n):
5  ...     if n == len(s):
6  ...         return ""
7  ...     return reverse_help(s, n + 1) + s[n]
8  ...
9  >>> reverse_main("pleh")
10 'help'
11 >>> reverse_main("Hello there.")
12 .ereht olleH
```

The function reverse_help(), given in lines 3 through 6, is a recursive function. It returns the reserve of its first (string) argument starting from the (integer) index specified by the second argument. Thus, when reverse_help()'s second argument is 0, the entire string is reversed. If, as in the code, the second argument is called n, the general case is to return the reverse of the string starting from the (n + 1)th character concatenated with the nth character. The base case is, as shown in lines 5 and 6, to return the empty string if the index is equal to the length of the string (i.e., return the empty string if there are no more characters that must be rearranged). In some sense this code is similar to the end message in r3() in Listing 12.4 where the printed arguments counted down from the maximum value to zero. Lines 9 through 12 demonstrate the function works properly.

12.4 Merge Sort

All the previous examples were intended to illustrate the implementation and behavior of recursive functions. They were not, however, very practical and there are much simpler ways to obtain identical results without using recursion. Now we want to consider a much more practical, "real world" application of recursion. We want to demonstrate how recursion can be used to efficiently sort the elements of a `list`. Before digging into the details, there are two important things to note. First, the subject of sorting is actually quite a complicated one! Though quite interesting at times, we will not delve into any of the complexities of the subject. Suffice it to say that, although the algorithm we will consider here is quite good, it is not optimum. Second, don't forget that `lists` have a `sort()` method and, in fact, there is a built-in function called `sorted()` that can be used to sort iterables. These can be used to sort iterables more efficiently than the approach described here.

The algorithm we will implement is called merge-sort. A key component of the algorithm is the ability to merge two `lists` that are assumed to be in sorted order. The resulting `list` is also in sorted order. Given the ability to merge `lists` in this way, we can start with an unsorted `list` and (recursively) break it down into a collection of single-element `lists`. A single-element `list` is inherently sorted. We can merge the single-element `lists` into sorted two-element `lists`. We can then merge the resulting two-element `lists` into four-element `lists`, and so on, until we have obtained all the elements in sorted order. (Despite the description here, the number of elements does not have to be a power of two as will be made more clear below.)

Let's start by considering the `merge()` function shown in Listing 12.12. This is a non-recursive function that returns the a single `list` that contains all the elements of the two `lists` it is given as arguments. It is assumed the two `lists` this function is passed are already sorted. The `list` the function returns is also sorted. So, for example, if the function is passed the `lists` `[1, 4]` and `[2, 3]`, the resulting `list` is `[1, 2, 3, 4]`. The discussion continues following the listing.

Listing 12.12 The `merge()` function that merges two sorted `lists` and returns a single sorted `list`.

```
1  >>> def merge(left, right):
2  ...     result = []      # Accumulator for merged list.
3  ...     while len(left) > 0 or len(right) > 0:
4  ...        if len(left) > 0 and len(right) > 0:
5  ...           if left[0] <= right[0]:      # left smaller than right.
6  ...              result.append(left[0])    # Append element from left.
7  ...              left = left[1 : ]         # Remove first left element
8  ...           else:                        # right smaller than left.
9  ...              result.append(right[0])   # Append element from right.
10 ...              right = right[1 : ]       # Remove first right element.
11 ...        elif len(left) == 0:      # No elements in left.
12 ...           result.extend(right)   # Extend by remaining right elements.
13 ...           break                  # Terminate loop.
14 ...        else:                     # No elements in right.
15 ...           result.extend(left)    # Extend by remaining left elements.
```

```
16 ...            break                  # Terminate loop.
17 ...        print("result: ", result) # For sake of illustration.
18 ...    return result
```

The two `lists` passed to the function are called `left` and `right`. Values are taken from these `lists` and appended to the `list result` which is initialized to the empty `list` in line 2. As an element is taken from `left` or `right`, the `list` from which the element was taken is reduced in size to now exclude this element. This continues until all the elements have been move from `left` and `right` to `result`. To accomplish this, a `while` loop, whose header is in line 3, continues to iterate while there is at least one element in either `left` or `right`. The first statement in the body of the loop, line 4, checks if both `left` *and* `right` have at least one element. If this is true, then the first elements of the two `lists` are compared (the first elements are the smallest of both `lists`). The smaller of the two elements is appended to `result` and the `list` from which the element came is reset to exclude the element.

If, however, `left` and `right` don't both have at least one element, then, in line 11, we check to see if length of `left` is zero. If that's the case, then we *extend* `result` by whatever elements remain in `right`. Finally, the `else` clause in line 14 ensures that if there are no more elements in `right`, then `result` will be extended by any elements that remain in `left`. The `print()` statement in line 17 is merely for the sake of illustration: it displays the contents of the `result` list for each iteration of the loop.

The `merge()` function is demonstrated in Listing 12.13. In line 1 the function is called with two `lists`, each of two elements. The resulting `list` shown in line 5 is the correctly merged sorted `list`. In line 6 `merge()` is called with a `list` of three elements and another of five elements. The result show in line 14 is again the correctly sorted `list`.

Listing 12.13 Demonstration of the `merge()` function.

```
1  >>> merge([1, 4], [2, 3])
2  result:  [1]
3  result:  [1, 2]
4  result:  [1, 2, 3]
5  [1, 2, 3, 4]
6  >>> merge([1, 5, 10], [0, 4, 6, 7, 8])
7  result:  [0]
8  result:  [0, 1]
9  result:  [0, 1, 4]
10 result:  [0, 1, 4, 5]
11 result:  [0, 1, 4, 5, 6]
12 result:  [0, 1, 4, 5, 6, 7]
13 result:  [0, 1, 4, 5, 6, 7, 8]
14 [0, 1, 4, 5, 6, 7, 8, 10]
```

Now that we know how to merge two sorted `lists`, let's write a function called `merge_sort()` that takes a single unsorted `list` as its argument. This function uses recursion as well as the `merge()` function to return a sorted version of the `list` it was passed. Before considering the

code to implement this, let's consider how it works. Any `list` that has more than one element can be divided into two "sublists" of (nearly) equal length. If the original `list` has an even number of elements, then the two sublists will have an equal number of elements. If the original `list` had an odd number of elements, then one of the sublists will have one element more than the other sublist. Continuing this process, we can divide the original `list` into a collection of single-element `lists`. Note that even if the original `list` is relatively large, this can be done very quickly! If there are N elements in the original `list`, then it takes at most $\log_2(N)$ steps to divide this into a collection of N single-element `lists`. So, for example, if N is 1,000,000,000, then it takes only about 30 sets of subdivisions to divide this into individual elements!

As mentioned above, a single element `list` is inherently sorted. We can merge the single-element `lists`, using the `merge()` function of Listing 12.12 to form two-element `lists`. From these we can form four-element `lists`, and so on. (However, when reassembling the sublists, we may obtain `lists` with an odd number of elements if, in the process of creating the sublists, we encountered an odd number of elements.) Eventually we get to the point where all the sublists have been merged to obtain a new `list` that contains all of the elements of the original `list`, but now in sorted order. This process is illustrated in Fig. 12.1 where the merge-sort algorithm is used to sort a `list` of ten integers. The original order of these integers is shown in "Step 1."

In Fig. 12.1, Steps 2 through 5 involve dividing the original `list` into progressively smaller sublists. Note that Step 3 yields `lists` of unequal length owing to the fact that `lists` of five elements had to be divided in two. At Step 5 all that remains are single-element `lists`.

At Step 6 (i.e., the step following the double lines), we begin to merge the single-element `lists` into two-element `lists`. The last elements to be subdivided are the first to be merged (these are the pairs of numbers (5, 6) and (0, 2)). The merged `lists` are always in sorted order. In Step 9 we obtain the complete `list` of sorted values.

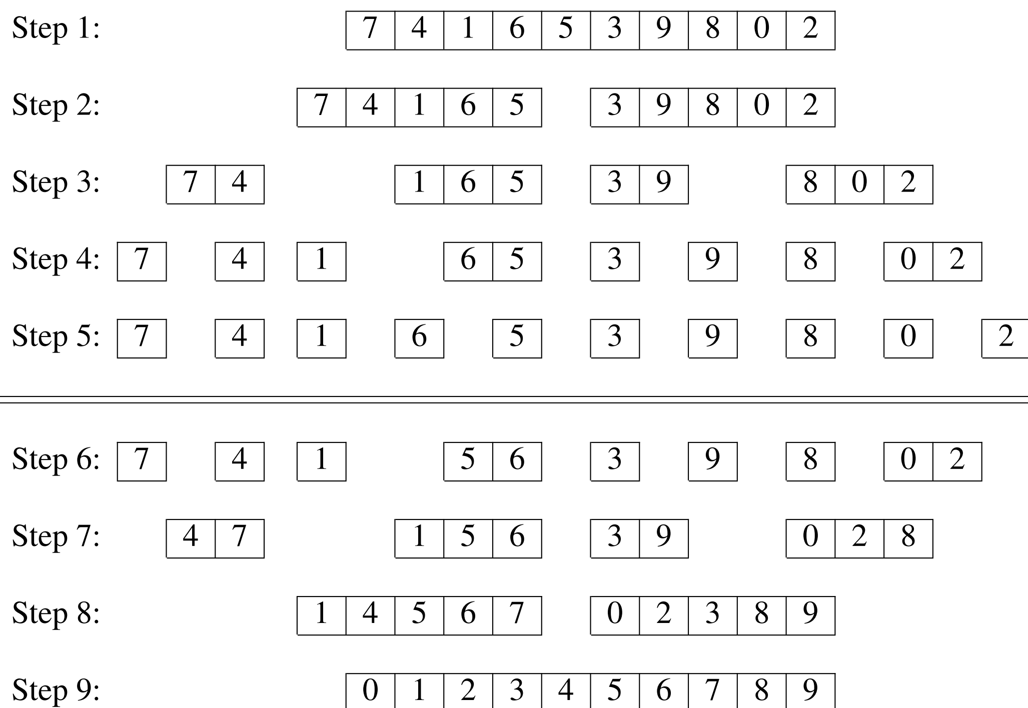
With that understanding, a suitable function to implement the merge-sort alogorithm is shown in Listing 12.14. Th `merge_sort()` functions starts, in lines 2 and 3, by checking if the `list` the function was passed has a single element (or no elements). If so, there is nothing to sort and the function merely returns this `list`. This is the base case. The discussion continues following the listing.

Listing 12.14 The `merge_sort()` function takes an unsorted `list` as its argument and returns a `list` with the elements sorted.

```
1  >>> def merge_sort(xlist):
2  ...     if len(xlist) <= 1:
3  ...         return xlist
4  ...     left = xlist[ : len(xlist) // 2]   # First "half" of list.
5  ...     right = xlist[len(xlist) // 2 : ] # Second "half" of list.
6  ...     sorted_left = merge_sort(left)
7  ...     sorted_right = merge_sort(right)
8  ...     return merge(sorted_left, sorted_right)
9  ...
```

If the argument `xlist` has more than one element, then statements in lines 4 and 5 break this `list` into two `lists` called `left` and `right`. (These `lists` will have the same number of

Figure 12.1: Steps used in the merge-sort algorithm to sort a `list` of ten integers. The original `list` is shown in Step 1. The `list` is divided into smaller `lists` until, in Step 5, the `lists` contain a single element. In Steps 5 through 9 these `lists` are merged so that eventually the final `list` correspond to the original `list`, but in sorted order.



elements if xlist has an even number of elements. If xlist has an odd number of elements right will have one more element than left.) Note that left and right are, in general, unsorted lists. Next, in line 6 and 7, merge_sort() is called to sort left and right. Then, finally, in line 8, these sorted lists are passed to merge() and the (sorted) list it produces is returned.

Though it only requires a few lines of code, merge_sort() is *not* a simple function. However, even without thinking about the function very much, you almost certainly have no problem understanding how it behaves for the base case when it is passed a single-element list. From there, it is not hard to figure out how merge_sort() behaves when it is passed a two-element list. In this case, with the initial call, the base case is not triggered. Instead, the two-element list is split into two single-element lists. When merge_sort() is called with these single-element lists, we simply get back these lists and then they are merged with merge() and the resulting two-element list is returned. merge() ensures the elements of the two-element list are in order. You can extend this line of logic to lists of any number of elements. Essentially what happens is the list is broken into a collection of single-element lists which are, in turn, merged into two-element lists. These are then merged, and so on.

Listing 12.15 demonstrates that merge_sort() behaves as advertised. In line 1, it is called with a list of integers. The resulting list, in line 2, is in sorted order. In line 3, merge_sort() is called with a list of strings. The resulting list in line 4 is again in sorted order. Provided all the element of the list can be compared using the relation operator <= (less than or equal to), then the list can be sorted using this implementation of merge_sort(). (The print() statement was removed from merge() for the sake of this demonstration.)

Listing 12.15 Demonstration of the merge_sort() function.

```
1 >>> merge_sort([7, 4, 1, 6, 5, 3, 9, 8 0, 2])
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 >>> merge_sort(['zebra', 'bat', 'dog', 'cat', 'ape', 'eel'])
4 ['ape', 'bat', 'cat', 'dog', 'eel', 'zebra']
```

Chapter 13

Turtle Graphics

13.1 Introduction

Graphical User Interfaces (GUI's) provide a rich environment in which information can be exchanged between a user and the computer. GUI's are not limited to simply displaying text and reading text from the keyboard. GUI's enable users to control the behavior of a program by performing actions such as using the mouse to drag or click on graphical objects. GUI's can make using programs much more intuitive and easier to learn since they provide users with immediate visual feedback that shows the effects of their actions.

There are many Python packages that can be used to create graphics and GUI's. Two graphics modules, called turtle and tkinter, come as a part of Python's standard library. tkinter is primarily designed for creating GUI's. In fact, IDLE is built using tkinter. However, we will focus on the turtle module that is primarily used as a simple graphics package but can also be used to create simple GUI's.

The turtle module is an implementation of turtle graphics and uses tkinter for the creation of the underlying graphics. Turtle graphics dates back to the 1960's and was part of the Logo programming language.[1] This chapter provides an introduction to using the graphics capabilities of the turtle module and demonstrates the creation of simple images and simple GUI's for games and applications. We will not delve into the details of building and designing GUI's, but many of the skills developed here can be applied to more complicated GUI designs if you wish to pursue that in the future. In addition to helping you gain practical programming skills, learning to use turtle graphics is fun and it enables you to use Python to be visually creative!

13.2 Turtle Basics

Among other things, the methods in the turtle module allow us to draw images. The idea behind the turtle part of "turtle graphics" is based on a metaphor. Imagine you have a turtle on a canvas that is holding a pen. The pen can be either up (not touching the canvas) or down (touching the canvas). Now think of the turtle as a robot that you can control by issuing commands. When the

From the file: `turtle-graphics.tex`

[1] See **en.wikipedia.org/wiki/Turtle_graphics**

pen it holds is down, the turtle leaves a trail when you tell it to move to a new location. When the pen is up, the turtle moves to a new position but no trail is left. In addition to position, the turtle also has a heading, i.e., a direction, of forward movement. The turtle module provides commands that can set the turtle's position and heading, control its forward and backward movement, specify the type of pen it is holding, etc. By controlling the movement and orientation of the turtle as well as the pen it is holding, you can create drawings from the trails the turtle leaves.

13.2.1 Importing Turtle Graphics

In order to start using turtle graphics, we need to import the turtle module. Start Python/IDLE and type the following:

Listing 13.1 Importing the turtle module.

```
>>> import turtle as t
```

This imports the `turtle` module using the identifier `t`. By importing the module this way we access the methods within the module using `t.<object>` instead of `turtle.<object>`. To ensure that the module was properly imported, use the `dir()` function as shown in Listing 13.2.

Listing 13.2 Using `dir()` to view the `turtle` module's methods and attributes.

```
1 >>> dir(t)
2 ['Canvas', 'Pen', 'RawPen', 'RawTurtle', 'Screen', 'ScrolledCanvas',
3  'Shape', 'TK', 'TNavigator', 'TPen', 'Tbuffer', 'Terminator',
4  'Turtle', 'TurtleGraphicsError', 'TurtleScreen', 'TurtleScreenBase',
5  'Vec2D', '_CFG', '_LANGUAGE', '_Root', '_Screen', '_TurtleImage',
6  '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
7      ...   <<MANY LINES OF OUTPUT DELETED>>
8  'window_width', 'write', 'write_docstringdict', 'xcor', 'ycor']
```

The `list` returned by the `dir()` function in Listing 13.2 has been truncated—in all, there are 173 items in this `list`. To learn more about any of these attributes or methods, you can use the `help()` function. For example, to learn about the `forward()` method, enter the statement shown in line 1 of Listing 13.3.

Listing 13.3 Learning about the `forward()` method using `help()`.

```
1 >>> help(t.forward)
2 Help on function forward in module turtle:
3
4 forward(distance)
5     Move the turtle forward by the specified distance.
6
```

```
7      Aliases: forward | fd
8
9      Argument:
10     distance -- a number (integer or float)
11
12     Move the turtle forward by the specified distance, in the direction
13     the turtle is headed.
14                ...
15     <<REMAINING OUTPUT DELETED>>
```

From this we learn, as shown in lines 12 and 13, that this method moves "the turtle forward by the specified distance, in the direction the turtle is headed." We also learn that there is a shorter name for this method: `fd()`.

13.2.2 Your First Drawing

Let's begin by telling our turtle to draw a line. Try entering the command shown in Listing 13.4.

Listing 13.4 Drawing a line with a length of 100 units.

```
>>> t.fd(100)
```

As shown in Fig. 13.1, a graphics window should appear in which you see a small arrow 100 units to the right of the center of the window.[2] A thin black line is drawn from the center of the window to the tail of the arrow. The arrow represents our "turtle" and the direction the arrow is pointing indicates the current heading. The `fd()` method is a shorthand for the `forward()` method—the two methods are identical. `fd()` takes one integer argument that specifies the number of units you want to move the turtle forward in the direction of the current heading.[3] If you provide a negative argument, the turtle moves backwards the specified amount. Alternatively, to move backward one can call either `backward()`, `back()`, or `bk()`.

The default shape for our turtle is an arrow but if we wanted to have it look like a turtle we could type the command shown in Listing 13.5.

Listing 13.5 Changing the shape of the turtle.

```
>>> t.shape("turtle")
```

This replaces the arrow with a small turtle. We can change the shape of our turtle to a number of other built in shapes using the `shape()` method. We can also create custom shapes although we won't cover that here.

[2] When it first opens, this window may appear behind previously opened windows. So, you may have to search for it.

[3] By default the units correspond to pixels, i.e., individual picture-elements or dots on your screen, but one can reset the coordinates so that the units can correspond to whatever is most convenient to generate the desired image.

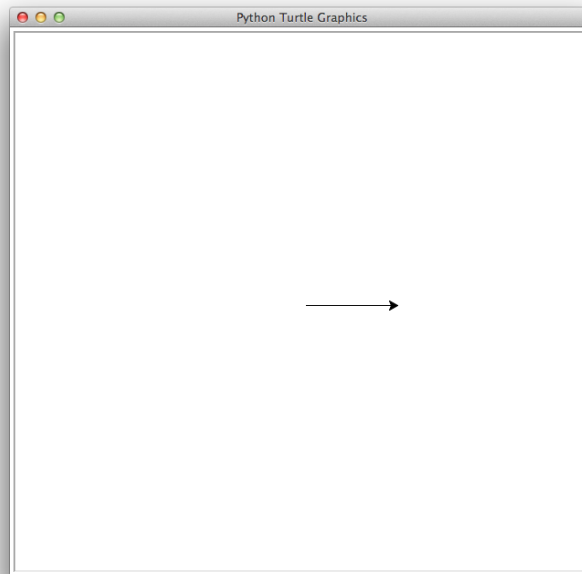


Figure 13.1: A line of length 100 produced by the `fd()` method.

Even though our turtle's shape appears on the graphics window, the turtle is not truly part of our drawing. The shape of the turtle is there to help you see the turtle's current position and heading, but you need to issue other commands, such as `fd()`, to create a drawing. If you have created a masterpiece and you no longer want to see the turtle in your graphics window, you can enter the command shown in Listing 13.6.

Listing 13.6 Command to hide the turtle's shape from the screen.

```
>>> t.hideturtle()
```

This hides the image that currently represents the turtle. In fact, you can continue to create lines even when the turtle's shape is hidden, but you will not be able to see the turtle's current position nor its heading. If you want to see the turtle again, simply issue the command shown in Listing 13.7.

Listing 13.7 Making the turtle visible.

```
>>> t.showturtle()
```

The turtle's heading can be controlled using one of three methods: `left()`, `right()`, and `setheading()`; or the shorter aliases of `lt()`, `rt()`, and `seth()`, respectively. `left()` and `right()` turn the turtle either to the left or right, respectively, by the number of degrees given as the argument. These turns are relative to the turtle's current heading. So, for example, `left(45)` causes the turtle to turn 45 degrees to the left. On the other hand, `setheading()` and `seth()`

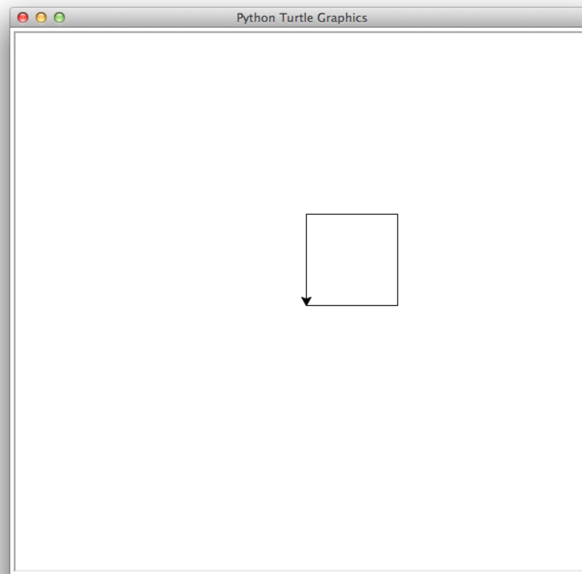


Figure 13.2: A square box.

set the absolute heading of the turtle. A heading of 0 is horizontally to the right (i.e., east), 90 is up (i.e., north), 135 is up and to the left (i.e., northwest), and so on.

Assuming you have previously entered the command of Listing 13.4, enter the commands in Listing 13.8. After doing this you should see a square drawn in the graphics window as shown in Fig. 13.2.

Listing 13.8 Commands to change the turtle's heading and draw a square box.

```
1 >>> t.left(90)
2 >>> t.fd(100)
3 >>> t.left(90)
4 >>> t.fd(100)
5 >>> t.left(90)
6 >>> t.fd(100)
```

What if we want to change the location of the turtle without generating a line? We can accomplish this by calling the method `penup()` before we enter commands to move the turtle. To re-enable drawing, we call the method `pendown()`.

We can also move the turtle to a specific position within the graphics window by using the `setposition()` method (or its aliases `setpos()` and `goto()`). `setposition()`'s arguments are the desired `x` and `y` values. The change of position does not affect the turtle's heading. If the pen is down, when you call `setposition()` (or its aliases), a straight line is drawn from that starting point to the position specified by the arguments. To demonstrate the use of `penup()`, `pendown()`, and `setposition()`, issue the commands shown in Listing 13.9. The resulting image is shown in Fig. 13.3.

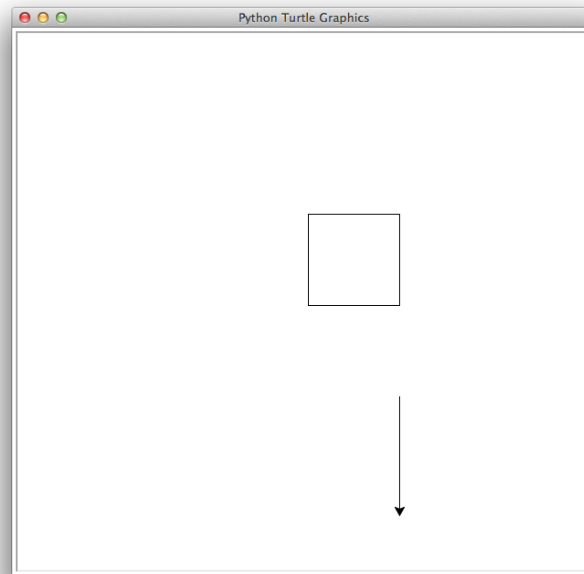


Figure 13.3: Result of moving the turtle without drawing a line and then, once at the new location, drawing a line.

Listing 13.9 Using `penup()` and `setposition()` to move the turtle without making a line.

```
1 >>> t.penup()
2 >>> t.setposition(100, -100)
3 >>> t.pendown()
4 >>> t.fd(130)
```

If we want to erase everything that we previously drew, we can use either the `clear()` or `reset()` methods. `clear()` clears the drawing from the graphics window but it leaves the turtle in its current position with its current heading. `reset()` clears the drawing and also returns the turtle to its starting position in the center of the screen. To illustrate the behavior of `clear()`, enter the statement shown in Listing 13.10.

Listing 13.10 Using the `clear()` method to clear the image.

```
>>> t.clear()
```

You should now see that the drawing that our turtle generated has been cleared from the screen but the turtle is still in the state that you last specified. To demonstrate what `reset()` does, enter the command shown in Listing 13.11.

Listing 13.11 Resetting the turtle and clearing the image.

```
>>> t.reset()
```

The turtle is moved back to the center of the screen with its original heading. Note that we do not need to call `clear()` before we call `reset()`. `reset()` will also clear the drawing—in the above example they were done sequentially solely for demonstrating their behavior.

13.3 Basic Shapes and Using Iteration to Generate Graphics

The commands shown in Listing 13.8 that we used to draw a square box would be rather cumbersome to type repeatedly if we wanted to draw more than one box. Observe that we are typing the same commands four times. As you already know, this sort of iteration can be accomplished in a much simpler way using a `for-loop`. Let's create a function called `square()` that draws a square using a `for`-loop. The function takes one argument which is the length of the square's sides. Enter the commands in Listing 13.12 that define this function and then call it three times, each time with a different length. The result should be the squares that are shown in Fig. 13.4.

Listing 13.12 A function that draws a square using a `for-loop`.

```
1  >>> def square(length):
2  ...     for i in range(4):
3  ...         t.fd(length)
4  ...         t.left(90)
5  ...
6  >>> square(60)
7  >>> square(100)
8  >>> square(200)
```

Building a square from straight lines is relatively straightforward, but what if we want to draw circles? Turtle provides a method called `circle()` that can be used to tell the turtle to draw a complete circle or only a part of a circle, i.e., an arc. The `circle()` method has one mandatory argument which is the radius of the circle. Optional arguments specify the "extent," which is the degrees of arc that are drawn, and the "steps," which are the number of straight-line segments used to approximate the circle. If the radius is positive, the circle is drawn (starting from the current position) by turning to the left (counterclockwise). If the radius is negative, the circle is drawn (starting from the current position) by turning to the right (clockwise). To demonstrate this, enter the commands shown in Listing 13.13. After issuing these commands, the graphics window should appear as shown in Fig. 13.5

Listing 13.13 Drawing circles using the `circle()` method.

```
1  >>> t.reset()       # Remove previous drawings and reset turtle.
2  >>> t.circle(100)   # Draw circle counterclockwise with radius 100.
3  >>> t.circle(-50)   # Draw circle clockwise with radius 50.
```

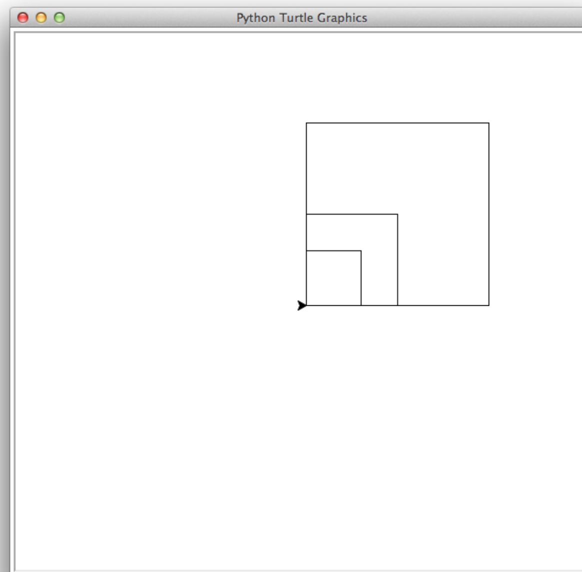


Figure 13.4: Drawing multiple boxes with `square()` of Listing 13.12.

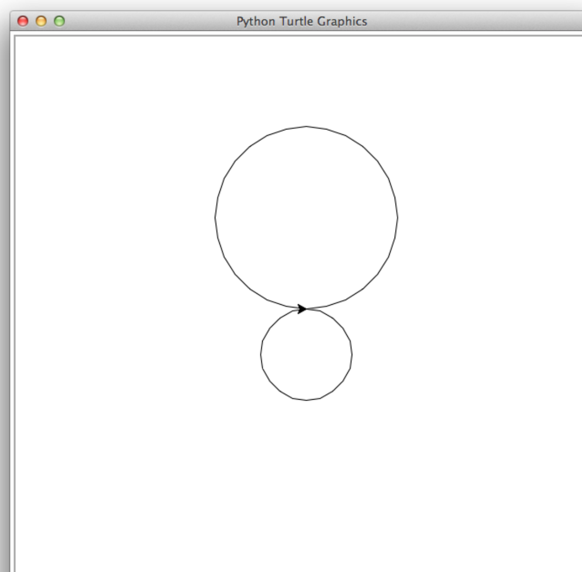


Figure 13.5: The circles drawn by the code in Listing 13.13.

13.3.1 Controlling the Turtle's Animation Speed

If you have been watching the turtle move in the graphics window as you issue commands, you will have noticed that the turtle goes through its motions relatively slowly. Sometimes watching the turtle move can be helpful but there are ways that you can speed up the movement (and hence speed up the drawing process). There is a `speed()` method for which the argument specifies the speed as an integer between 0 and 10. A value of 1 is the slowest speed. Speeds generally increase with increasing arguments so that 2 is faster than 1, three is faster than 2, and so on. However, rather than 10 being be fastest speed, it is actually the second to the fastest speed—0 is the fastest speed. The default speed is 3. To demonstrate this, issue the commands shown in Listing 13.14.

Listing 13.14 Speeding up the animation by using the `speed()` method.

```
1 >>> t.reset()
2 >>> t.speed(0)
3 >>> t.circle(100)
4 >>> t.circle(-50)
```

The image should be the same as 13.5, but it should render noticeably faster than it did previously. However, even though supplying the `speed()` method with an argument of 0 makes the animation faster, it is still quite slow if we want to draw a more complex drawing. In order to make our drawing appear almost immediately we can make use of the `tracer()` method. `tracer()` takes two arguments. One controls how often screens should be updated and the other controls the delay between these update. To obtain the fastest possible rendering, both these arguments should be set to zero as shown in Listing 13.15.

Listing 13.15 Turning off the animation with `tracer()`.

```
>>> tracer(0, 0)
```

By calling `tracer()` with bother arguments set to zero, we are essentially turning off all animation and our drawings will be drawn "immediately." However, if we turn the animation off in this way we need to explicitly update the image with the `update()` method after we are done issuing drawing commands.

If you want to reset `tracer()` to its original settings, its arguments should be 1 and 10, as shown in Listing 13.16.

Listing 13.16 Restoring animation to the default settings.

```
>>> tracer(1, 10)
```

13.4 Colors and Filled Shapes

Now that we know how to create some basic shapes, let's explore how we can create more complex images. In order to change the color of our turtle's pen we can use the `color()` method as shown in line 2 of Listing 13.17.

Listing 13.17 Changing the color of the turtle's pen.

```
1 >>> t.reset()
2 >>> t.color("blue")
```

This change the turtle's pen to blue. There are actually numerous ways of specifying a color in Python's implementation of turtle graphics. You can, as shown in Listing 13.17, specify a color via a string.[4] Alternatively, we can specify a color by providing numeric values the specify the amount of red, green, and blue. To learn more about the use of colors, use the `help()` function to read about `t.color` or `t.pencolor`.

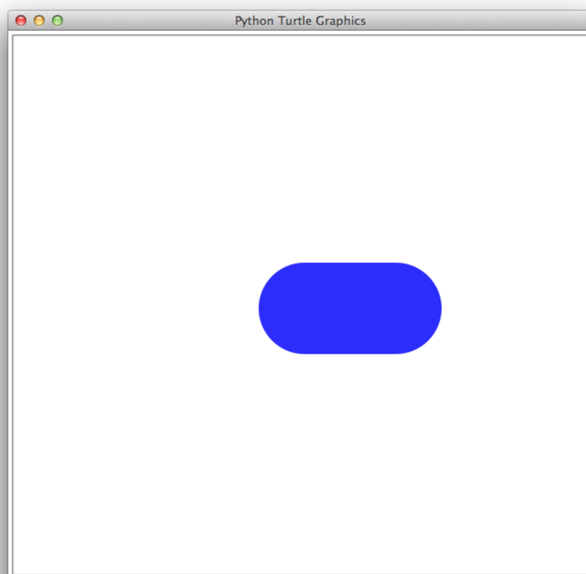


Figure 13.6: A blue line with a thickness and width of 100.

We can also change the thickness of the turtle's pen by using the `pensize()` method and passing it an integer argument that specifies the thickness. This is demonstrated in Listing 13.18. After issuing these commands (and those of Listing 13.17), you will see that a thick blue line has been drawn in the graphics window as shown in Fig. 13.6

[4]The string must be one of the Tk color specifications. A listing of these can be found at www.tcl.tk/man/tcl8.4/TkCmd/colors.htm.

Listing 13.18 Changing the thickness of the turtle's pen.

```
>>> t.pensize(100)
>>> t.fd(100)
```

We can also change the background color of the graphics window. To demonstrate this, you should enter the commands of Listing 13.19.

Listing 13.19 Changing the background color of the window.

```
>>> t.bgcolor("blue")   # Change the background to blue.
>>> t.bgcolor("white")  # Change it back to white.
```

Let's take what we have learned so far and draw a more complex image. Enter the commands shown Listing 13.20. The `for`-loop that starts in line 3 sets the heading to angles between 0 and 345 degrees, inclusive, in 15 degree increments. For each of these headings it draws a circle with a radius of 100. (Note that a heading of 360 is the same as a heading of 0, so that why a circle is not draw with a heading of 360 degrees.) The resulting image is shown in Fig. 13.7.

Listing 13.20 Creating a red flower.

```
>>> t.reset()
>>> t.color("red")
>>> for angle in range(0, 360, 15):
...     t.seth(angle)
...     t.circle(100)
```

We can also use iteration to change the color and thickness of the turtle's pen more dynamically. As an example, try typing the commands shown in Listing 13.21. This should result in the image shown in Fig. 13.8.

Listing 13.21 Code the creation of a colorful spiral.

```
>>> colors = ["blue", "green", "purple", "cyan", "magenta", "violet"]
>>> t.reset()
>>> t.tracer(0, 0)
>>> for i in range(45):
...     t.color(colors[i % 6])
...     t.pendown()
...     t.fd(2 + i * 5)
...     t.left(45)
...     t.width(i)
...     t.penup()
...
>>> t.update()
```

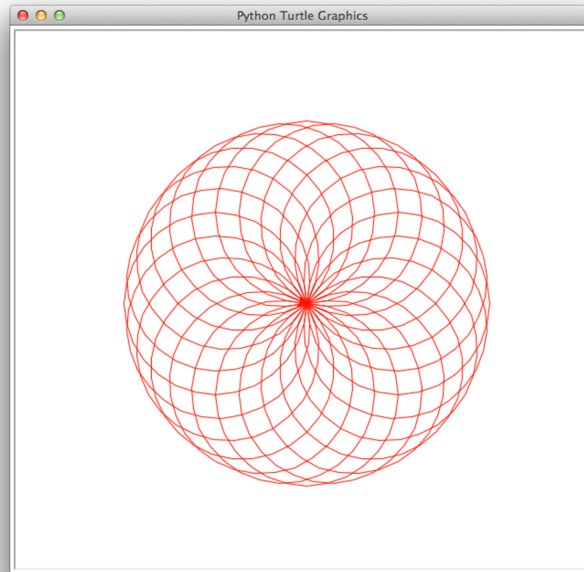


Figure 13.7: A red flower using `circle()`.

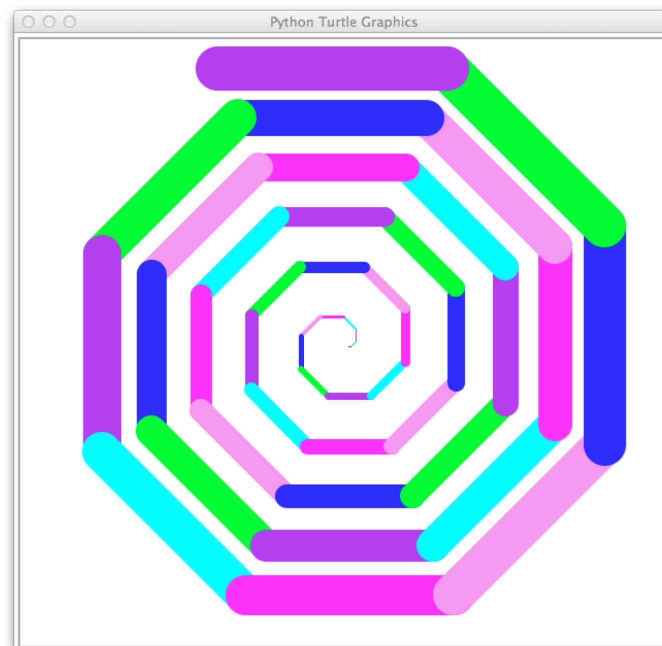


Figure 13.8: A colorful spiral.

13.4.1 Strange Errors

As you work through the examples in this chapter you may encounter strange, complicated errors that result from small typos or bugs in your code. These errors may look alarming and confusing but you can usually discern the cause of the error if you look at the last few lines. For example, enter the command shown in line 1 of Listing 13.22.

Listing 13.22 Error example.

```
1  >>> t.color("pumpkin")
2  Traceback (most recent call last): File "<stdin>", line 1, in <module>
3  File "<string>", line 1, in color File
4  "/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/turtle.py",
5  line 2209, in color pcolor = self._colorstr(pcolor) File
6  "/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/turtle.py",
7  line 2689, in _colorstr return self.screen._colorstr(args) File
8  "/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/turtle.py",
9  line 1151, in _colorstr raise TurtleGraphicsError("bad color string: %s" %
10 str(color)) turtle.TurtleGraphicsError: bad color string: pumpkin
```

The resulting error message, shown in lines 2 through 10, is long and possibly confusing but if you look at the last line it tells you that "`pumpkin`" is not a valid color string.

In the introduction to this chapter we mentioned that the turtle module uses tkinter for its underlying graphics. Sometimes the errors you get will contain messages that pertain to tkinter errors by using tk in the message. If you encounter one of these errors you typically do not need to know anything about tkinter. It is likely the cause of the error is somewhere in your turtle code, e.g., you are passing a method an incorrect value, so review your code carefully and make sure it conforms to the requirements of the turtle module.

13.4.2 Filled Shapes

We can also fill the shapes we draw by using the methods `begin_fill()` and `end_fill()`. To fill a shape, we must call first `begin_fill()`, then issue commands to draw the desired shape, and finally call `end_fill()`. When we call `end_fill()` the shape will be filled with the currently set color. To demonstrate this, try entering the commands of Listing 13.23. After entering these commands you should see a green filled box in the graphics window as shown in Fig. 13.9.

Listing 13.23 Commands to create a filled green box.

```
1  >>> t.reset()
2  >>> t.color("green")
3  >>> t.begin_fill()
4  >>> square(100) # The square() function of Listing 13.12.
5  >>> t.end_fill()
```

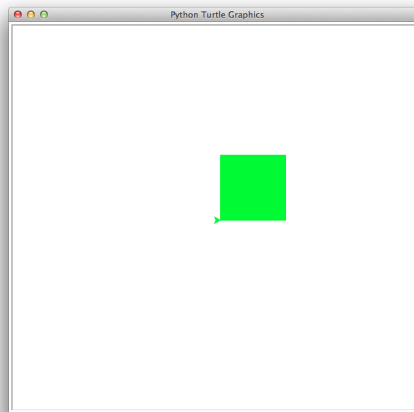


Figure 13.9: A filled green box.

You don't need to completely connect (close) the shape you want to fill. To illustrate this, try entering the code in Listing 13.24 and observe what you have drawn at each step. After issuing all these commands you should see the shape shown in Fig. 13.10

Listing 13.24 A three-sided shape that is not close.

```
1  >>> t.reset()
2  >>> t.width(10)      # Set line thickness to 10.
3  >>> t.begin_fill()  # Begin to define fill-shape.
4  >>> t.fd(150)        # Draw horizontal edge.
5  >>> t.seth(45)       # Set heading to 45 degrees.
6  >>> t.fd(150)        # Draw second edge.
7  >>> t.seth(90)       # Set heading to 90 degrees.
8  >>> t.fd(150)        # Draw vertical edge.
```

Now issue the statements shown in Listing 13.25. You should now see the image shown in Fig. 13.11. Note that the shape was filled as if there were an edge between the start-point and the end-point, but it will not draw a line along this edge. Instead, the current color is merely painted up to this edge.

Listing 13.25 Fill the shape started in Listing 13.24.

```
9  >>> t.color("blue")
10 >>> t.end_fill()
```

13.5 Visualizing Recursion

Chapter 12 provides an introduction to recursion. If you haven't done so already, it is recommended that you read that chapter before working through the examples in this section.

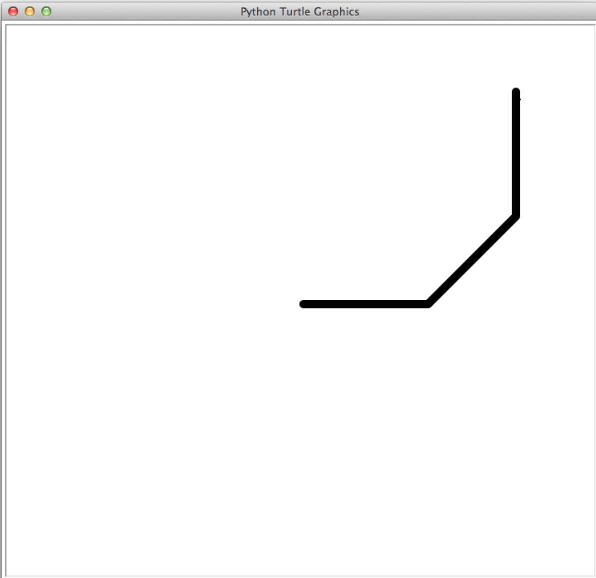


Figure 13.10: Shape that has not been closed.

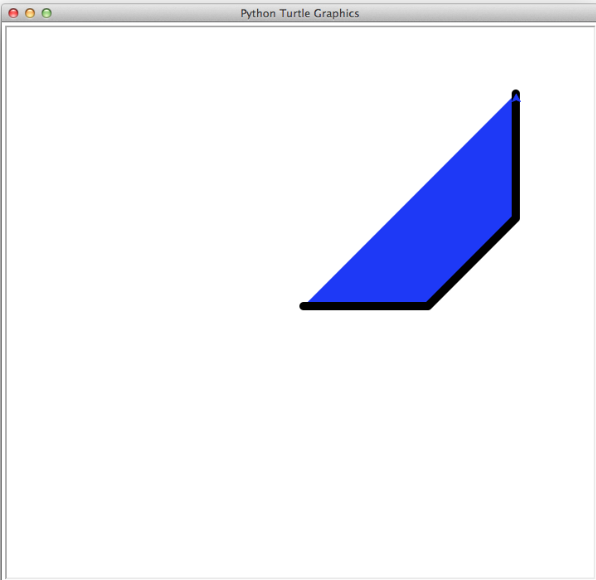


Figure 13.11: A shape can be filled even if it isn’t closed.

Recall the Fibonacci `fib()` function from Listing 6.23. Now consider the function `drawfib()` shown in Listing 13.26. This function takes two arguments. The second is identified as `len_ang` and, by inspection of the code, we see this parameter is used both as a length (when it is used to control movement in lines 2 and 13) and as an angle (when it used to alter the heading in lines 8, 10, and 12) The value of `len_ang` is never changed. The `drawfib()` function starts by drawing a line (line 2). Then, if the first argument is greater than 1, it alters the heading and calls itself twice, once with the value of the first argument reduced by 1 and another time with the argument reduced by 2. The final statement of the function moves the pen back to the starting point. If the first argument is 0 or 1, then the function function doesn't do anything before moving the pen back to the starting point. We accomplish doing "nothing" by using the `pass` statements given in lines 4 and 6. Note that bodies of loops, functions, conditional statements, and classes cannot be truly empty. However, we can make them effectively empty by using a `pass` statement.

Because we are not doing anything in the bodies of the first two clauses of the conditional statement, we could have eliminated lines 3 through 6 and replaced line 7 with "`if n > 1`." However, we have elected to write things as given to help establish a connection with the way numbers are generated in the Fibonacci sequence. The discussion of this code continues below the listing.

Listing 13.26 Code to draw a tree recursively. The resulting tree is related to the Fibonacci sequence.

```
1  >>> def drawfib(n, len_ang):
2  ...     t.forward(2 * len_ang)
3  ...     if n == 0:
4  ...         pass // Do nothing.
5  ...     elif n == 1:
6  ...         pass // Do nothing.
7  ...     else:
8  ...         t.left(len_ang)
9  ...         drawfib(n - 1, len_ang)
10 ...         t.right(2 * len_ang)
11 ...         drawfib(n - 2, len_ang)
12 ...         t.left(len_ang)
13 ...     t.backward(2 * len_ang)
14 ...
15 >>> # Six different starting points for six different trees.
16 >>> start_points = [[-300, 250], [-150, 250],
17 ...                 [-300, 110], [-80, 110],
18 ...                 [-300, -150], [50, -150]]
19 >>>
20 >>> # For each starting point, draw a tree with n varying
21 ... # between 1 and 6 and len_ang set to 30.
22 >>> n = 0
23 >>> for start_point in start_points:
24 ...     x, y = start_point
25 ...     n = n + 1
```

```
26 ...         t.penup()
27 ...         t.setpos(x, y)
28 ...         t.pendown()
29 ...         drawfib(n, 30)
```

It probably isn't at all obvious why we would bring up the Fibonacci sequence in connection with the code in Listing 13.26. Before reading on, you should consider entering the function and see what it produces for different arguments. Consider values of n between 1 and 10 and values of len_ang in the range of 20.

Now, to help illustrate the connection between drawfib() and the Fibonacci sequence, the code in lines x through y of Listing 13.26 draw six different trees where n varies between 1 and 6 while the len_ang is held fixed at 30. The resulting trees are shown in Fig. 13.12. The "trunk" of each tree is the horizontal line that is the right-most component of each tree. If we had drawn a tree with an n of 0, it would appear the same as with an n of 1 (i.e., the top left drawing in Fig. 13.12. If we count the number of "tips" or "branches" on the left side of the tree, for an n or 0 or 1, there is only 1 tip. For an n of 2 there are 2 (top right tree). When n is 3 there are 3 (middle left tree). When n is 4 there are 5 tips (middle right). And, we see for an n of 5 or 6, there are 8 or 13 tips, respectively (the two bottom trees). Thus, the number of "tips" in our trees corresponds to the numbers in the Fibonacci sequence! (The number of tips is the nth number in the sequence.)

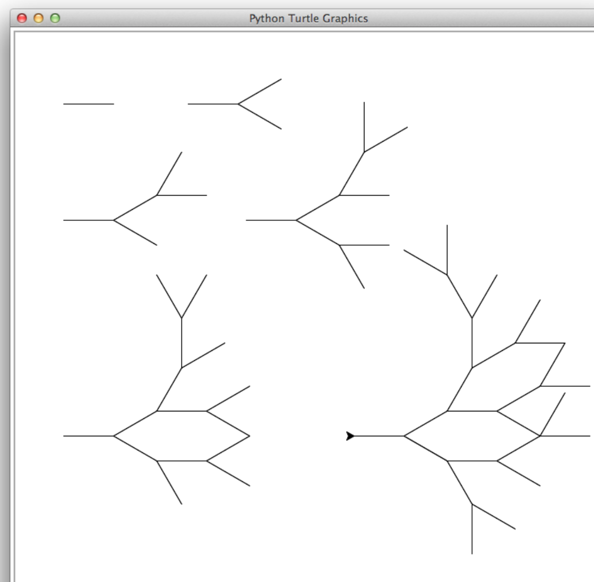


Figure 13.12: Drawings of multiple Fibonacci trees using the drawfib() function of Listing 13.26. The first argument of this function (n) varies between 1 and 6. The upper left tree was generated with an n of 1 while the bottom right tree had an n of 6.

A more complicated tree is drawn using the code shown in Listing 13.27. In this case n is 15 which results in 987 tips (although it would be almost impossible to count these from the tree figure itself). The resulting tree is shown in Fig. 13.13.

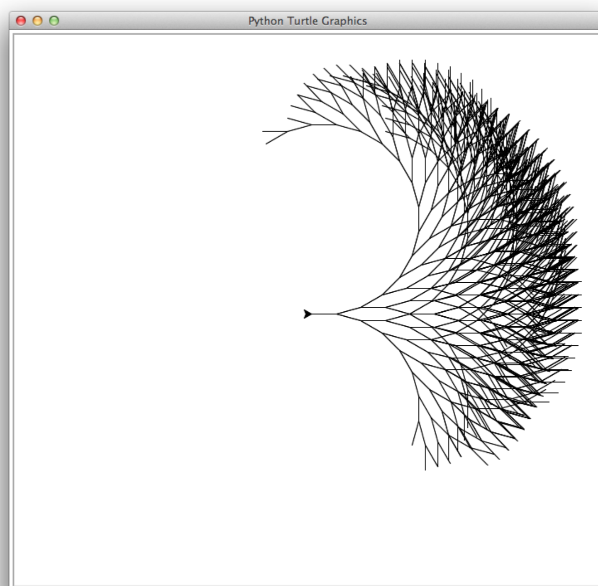


Figure 13.13: A Fibonacci tree with 987 tips.

Listing 13.27 Recursive tree using the `fib()` function with 987 branches.

```
1 >>> t.reset()
2 >>> t.tracer(0, 0)
3 >>> t.drawfib(15, 10)
4 >>> t.update()
5 >>> t.tracer(1, 10)
```

It might take a few seconds before this image appears:

Recursive definitions can also be used to draw beautiful and complex fractal images. A classic example of a fractal is a Koch snowflake. To generate a Koch snowflake, begin by entering the commands shown in Listing 13.28. The function `ks()` takes two arguments. The first is a length and the second specifies the recursive "depth" `d`. When `d` is 0, the function merely draws a straight line and returns. If `d` is greater than 0, the length is reduced by a factor of three and then the `ks()` function is called with this new length, with the recursion depth reduced by one-third, and with various headings. The last two lines of the listing produce the drawing shown in Fig. 13.14.

Listing 13.28 Code for producing (part of) a Koch snowflake.

```
1 >>> t.tracer(0, 0)
2 >>> def ks(length, d):
3 ...     if d == 0:
4 ...         t.forward(length)
5 ...     else:
```

```
6  ...         length = length / 3
7  ...         d = d - 1
8  ...         ks(length, d)
9  ...         t.right(60)
10 ...         ks(length, d)
11 ...         t.left(120)
12 ...         ks(length, d)
13 ...         t.right(60)
14 ...         ks(length, d)
15 ...
16 >>> ks(200, 3)
17 >>> t.update()
```

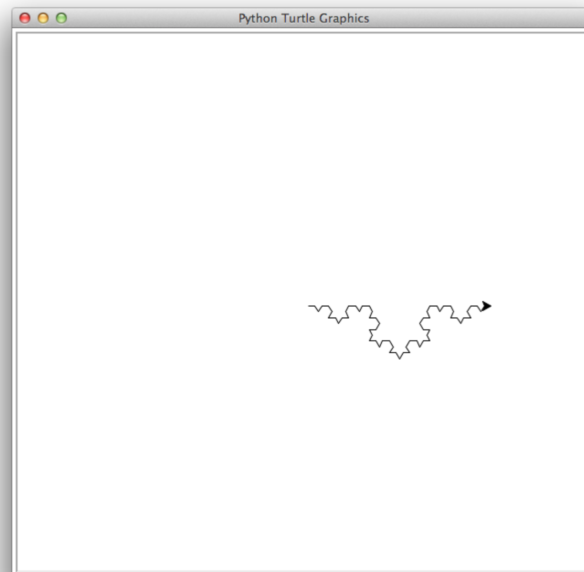


Figure 13.14: One side of a Koch snowflake.

As you can see, the `ks()` function only draws one side of the snowflake. If we want to complete the snowflake we can reuse this function and rotate the turtle appropriately to complete the picture. The code to accomplish this is given in Listing 13.29. The resulting drawing is shown in Fig. 13.15.

Listing 13.29 Code to produce a complete Koch snowflake.

```
1 >>> t.reset()
2 >>> colors = ["red", "orange", "pink"]
3 >>> for i in range(3):
4 ...     t.color(colors[i])
5 ...     ks(200, 3)
```

```
6 ...     t.left(120)
7 ...
8 >>> t.update()
```

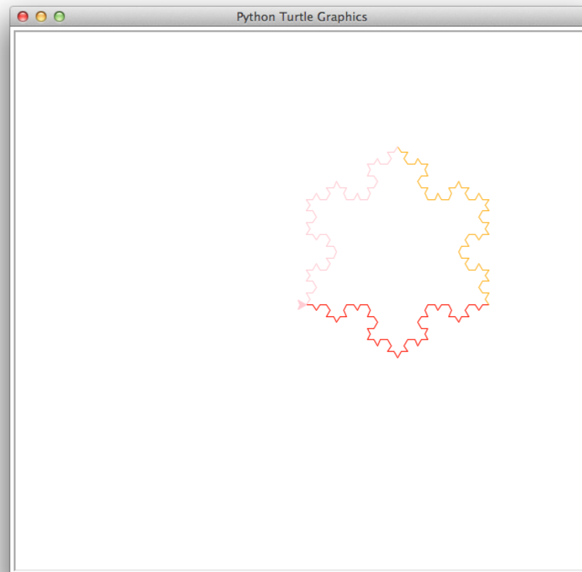


Figure 13.15: The Koch snowflake produced by the code of Listing 13.29.

13.6 Simple GUI Walk-Through

The turtle module provides simple GUI functionality. One common way GUI's interact with a user is to respond to the user's mouse clicks and perform certain actions based on those clicks. This section provides a walk-through for building a simple GUI application using "callback" functions to respond to mouse clicks. Callback functions are functions we write that specify what should be done when a particular event occurs. We need to "register" a callback function with the GUI software so that the appropriate function be called when the event occurs that the function was written to process.

13.6.1 Function References

We have previously seen that we can have a reference or alias to a `list`, i.e., we can have different identifiers that point to the same underlying memory. Also, when we pass a `list` as an actual parameter to a function, the function's formal parameter is a reference to that `list`. Python also allows us to have references to functions or, to put it another way, Python allows us to assign functions to identifiers. When we write a function name followed parentheses we are indicating that we want to call that function but what happens when we write a function name without parentheses? To find out, try typing the code shown in Listing 13.30.

Listing 13.30 Calling the `print()` function and obtaining a reference to it.

```
>>> print() # This prints a newline.
>>> print   # This returns a reference to the print() function.
<built-in function print>
```

The statement in line 2 and the output in line 3 show that "calling" a function without parentheses behaves differently than when we use parentheses. In fact, without the parentheses, we are not actually calling the function. When we write `print` without parentheses we obtain a reference to the function, i.e., an object that represents the `print()` function. Thus we can assign the `print()` function to an valid identifier, as Listing 13.31 indicates.

Listing 13.31 assigning `print()` to a variable

```
>>> foo = print
>>> type(foo)
<class 'builtin_function_or_method'>
>>> foo("Hello world!")
Hello world!
```

The `type()` function, in line 2, shows that `foo()` is now also a function. Notice we're even told that `foo()` is a built-in function! Of course, there isn't a `foo()` function built-into Python, but at this point we have made `foo()` indistinguishable from the actual built-in function `print()`. If we try to use `foo()` in place of `print()`, we find, as lines 4 and 5 indicate, that `foo()` now behaves exactly as `print()` does!

Given that we can assign functions to identifiers, it also makes sense that we can pass functions as arguments to other functions. To demonstrate this, the `fun_run()` function in Listing 13.32 takes two arguments. The first argument is a function while the second is an argument that first argument is supposed to act upon.

Listing 13.32 Demonstration of passing a function as parameter to another function.

```
>>> def fun_run(func, arg):
...     return func(arg)
...
>>> fun_run(print, "Hi!")          # Apply print() to a string.
Hi!
>>> foo = print
>>> fun_run(foo, "Hello there!") # Apply reference foo() to a string.
Hello there!
```

13.6.2 Callback functions

In order to make our GUI application respond to a user's mouse clicks, we must write a function that will be called whenever the mouse is used to click on the graphics window. First, as shown in Listing 13.33, let's define a function that takes two arguments and prints the value of those two arguments.

Listing 13.33 Function that prints its two arguments.

```
>>> def printxy(x, y):
...     print(x, y)
```

turtle graphics has a method called `onscreenclick()` that takes a single parameter. This parameter is assumed to be a reference to a function that takes two arguments. This function will be called whenever there is a mouse click on the graphics window. The arguments passed to the function will be the x and y coordinates of the point where the click occurred. To illustrate this, assuming you have entered the code in Listing 13.33, type the code shown in Listing 13.34. In line 1 we "register" the `printxy()` function so that it will be "called back" whenever there is a click on the graphics window. If you are using IDLE, you will also need to call `mainloop()` to enable to processing of events. Once you call `mainloop()`, the interactive prompt will disappear and it won't reappear until you close the graphics window (using the close button and the top of the window).

Listing 13.34 Use of the `onscreenclick()` method to register the `printxy()` function as a callback function.

```
>>> t.onscreenclick(printxy)
>>> t.mainloop()    # If you are using IDLE.
```

Once you have done this, click on the graphics window in various places. You should see pairs of numbers corresponding to the location of the point that was clicked! Note that these pairs of numbers will be displayed in environment in which you have been issuing the Python commands—the numbers will not appear in the graphics window.

13.6.3 A simple GUI

Now let's write a more complicated turtle GUI using the `square()` function you wrote in Sec. 13.4.2. First, let's set up the window where we exercise some fine-grain control of the window's size and coordinates. The code in Listing 13.35 show how this can be accomplished using a combination of the methods `setup()`, `screensize()`, and `setworldcoordinates()`.

Listing 13.35 Dividing the graphics window into thirds.

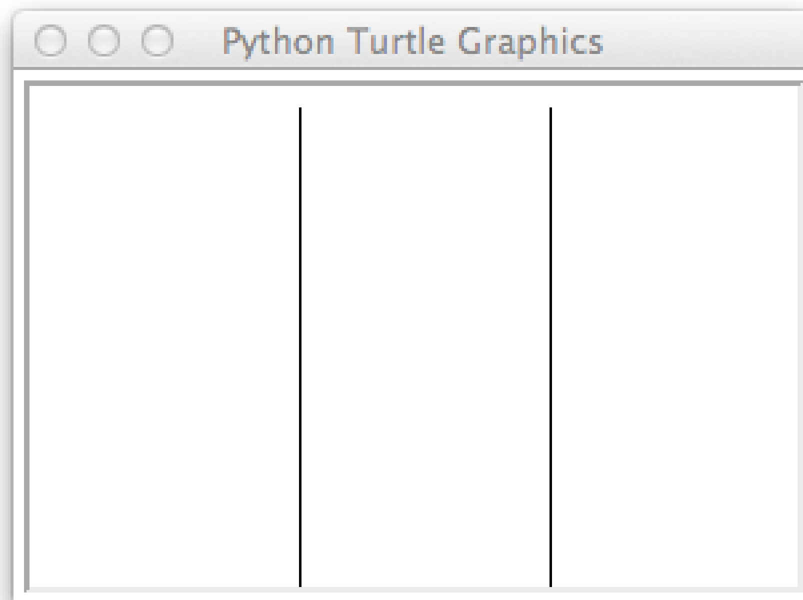


Figure 13.16: Window divided into thirds.

```
>>> # Set window to be 300 by 200 with the point (0, 0) as the
>>> # lower left corner and (300, 200) as the upper right corner.
>>> t.setup(300, 200)
>>> t.screensize(300, 200)
>>> t.setworldcoordinates(0, 0, 300, 200)
>>> # Draw two vertical lines to divide the window into thirds.
>>> t.penup()
>>> t.setpos(100, 0)     # First line.
>>> t.pendown()
>>> t.setpos(100, 200)
>>> t.penup()
>>> t.setpos(200, 0)     # Second line.
>>> t.pendown()
>>> t.setpos(200, 200)
```

After issuing these commands, your window should appear as shown in Fig. 13.16.

Now let's write a callback function that will draw a shape at the point where a click occurred, but the shape that is drawn depends on where the click occurred. The function should draw a green square if the click was in the left third of the window, red circles if the click was in the middle third of the window, and blue squares if the click was in the right third of the window.

Listing 13.36 Callback function to draw various shapes at the point of a click.

```
>>> def shapedrawer(x, y):
...     t.penup()
```

```
3  ...         # Set the position of the turtle to the clicked location.
4  ...         t.setpos(x, y)
5  ...         t.pendown()
6  ...         t.begin_fill()
7  ...         if x <= 100:          # Left third.
8  ...             t.color("green")
9  ...             square(10)
10 ...         elif 100 < x <= 200: # Middle third.
11 ...             t.color("red")
12 ...             t.circle(10)
13 ...         else:                 # Right third.
14 ...             t.color("blue")
15 ...             square(10)
16 ...         t.end_fill()
17 ...
18 >>> t.onscreenclick(shapedrawer)
19 >>> t.mainloop()
```

The last two statements in Listing 13.36 "register" our callback function and then enter the "main loop" to start processing events. Once you have entered this code, click in various places in the graphics window. You should be able to produce images like the one shown in Fig. 13.17.

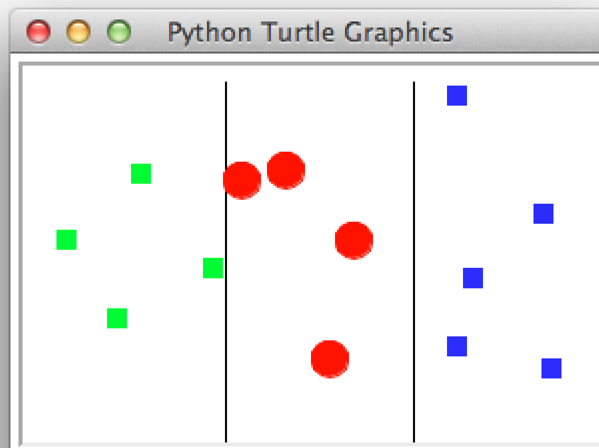


Figure 13.17: Shape GUI.

Chapter 14

Dictionaries

lists and tuples are *containers* in which data are collected so that elements of the data can be easily accessed. lists and tuples are also *sequences* in that data are organized in a well defined sequential manner. One element follows another. The first element has an index of 0, the next has an index of 1, and so on. Alternatively, negative indexing can be used to specify an offset from the last element of a list or tuple. We've discussed the use of slices where a portion of a list or tuple is specified by indices that indicate the start and end of the slice. Almost everything we have done in terms of accessing the elements of a list or tuple has been related to the sequential nature of the container and has relied on numeric indexing.

However, as you know from your day-to-day experiences, there are many situations in which we don't think of collections of data in a sequential manner. For example, your name, your height, and your age are all data associated with you, but you don't think about these in any particular order. Your name is simply your name. If we want to store a person's name, height, and age in a Python program, we can easily store this information in a list (if we want the information to be mutable) or a tuple (if we want the information to be immutable). If we do this, the order of an element dictates how it should be interpreted. So, for example, perhaps the first element in a list is a name (a string), the second element is an age (an integer), and the third element is a height (a float or integer in centimeters). Within a program this organization of information can work well, but there are limits to this. What if we want to keep track of many more facts about a person? Assume, perhaps, there are 15 separate pieces of data we want to record and use. The code to manipulate this information can become difficult to understand and maintain if the programmer only has the position within a list as the key to determining how the data should be interpreted.

As an alternative to lists and tuples, Python provides another container known as a dictionary or dict. Dictionaries share some syntactic properties with lists and tuples but there are many important differences. First, dictionaries are *not* sequential collections of data. Instead, dictionaries consist of key-value pairs. To obtain a "value" (i.e., the data of interest), you specify its associated key. In this way you can create a collection of data in which perhaps the keys are the strings 'name', 'age', and 'height'. The order in which Python stores the key-value pairs is not a concern. We merely need to know that when we specify the key 'name', Python will provide the associated name. When we specify the key 'age', Python will provide the associated age. And, when we specify the key 'height', Python will provide the associated height.

From the file: dictionaries.tex

In this chapter we will explore the ways in which dictionaries can be used. You are already familiar with traditional dictionaries in which the “key” is a given word. Using this key, you can look up the definition of the word (i.e., the data associated with this key). This model works well in some ways for visualizing a `dict` in Python. However, in a traditional dictionary all the keys/words are in alphabetical order. There is no such ordering of the keys in Python. Despite this lack of order, one of the important properties of a `dict` is that Python can return the data for a given key extremely quickly. This speed is maintained even when the dictionary is “huge.”[1]

14.1 Dictionary Basics

To create a `dict`, we use curly braces, i.e., `{}`. If there is nothing between the braces (other than whitespace), the `dict` is empty. It is not uncommon to start with an empty dictionary and later add key-value pairs. Alternatively, key-value pairs can be specified when the dictionary is created. This is done by separating the key from the value by a colon and separating key-value pairs by a comma. This is illustrated in Listing 14.1 which is discussed below the listing.

Listing 14.1 Creation of dictionaries.

```
1  >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2  >>> abe['height']
3  193
4  >>> abe['name']
5  'Abraham Lincoln'
6  >>> james = {}
7  >>> james['name']
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 KeyError: 'name'
11 >>> james['name'] = 'James Madison'
12 >>> james['height'] = 163
13 >>> james['age'] = 261
14 >>> print(abe)
15 {'age': 203, 'name': 'Abraham Lincoln', 'height': 193}
16 >>> james
17 {'age': 261, 'name': 'James Madison', 'height': 163}
```

In line 1 the dictionary `abe` is created with three key-value pairs. The use of whitespace surrounding a colon is optional. Although not done here, the declaration can span multiple lines because the opening brace behaves something like an opening parentheses and the declaration does not end until the corresponding closing brace is entered. To obtain the value associated with a key, the key is specified within square brackets as shown in lines 2 and 4.

[1] Python `dicts` are *hashes*. We will not consider the details of a hash, but essentially what happens in a hash is that a *hash function* is applied to the key. The value this function returns indicates, at least approximately, where the associated value can be found in memory.

An empty `dict` is created in line 6 and assigned to the variable `james`. In line 7 an attempt is made to access the value associated with the key `'name'`. However, since this key doesn't exist in `james`, this produces the `KeyError` exception shown in lines 8 through 10. However, we can add key-value pairs to a dictionary through assignment statements as shown in lines 11 through 13.[2]

In line 14 a `print()` statement is used to display the dictionary `abe`. In the output in line 15, note that the order of key-value pairs is not the same as the order in which we provided them.[3] In the interactive environment, we can see the contents of a dictionary simply by entering the dictionary name and hitting return as is done with `james` in line 16. The output in line 17 again shows that the order in which we specify key-value pairs is unrelated to the order in which Python stores or displays them.

In Listing 14.1 all the keys are strings. But, keys can be numeric values or tuples or, in fact, any immutable object! This is illustrated in Listing 14.2 where, in lines 1 through 4, the dictionary `d` is created with keys that are a string (line 1), an integer (line 2), a `tuple` (line 3), and a `float` (line 4). Lines 5 through 16 demonstrate that the keys do indeed produce the associated values. Note that the value associated with the `tuple` key (line 3) is itself a `dict` and the value associated with the `float` key (in line 4) is a `list`. Although a key cannot be a mutable object, the values associated with a key can be mutable. (`dicts` are mutable, because we can change keys and values.)

Listing 14.2 Demonstration that keys to a `dict` can be any immutable object. The associated values can be either mutable or immutable objects.

```
1  >>> d = {'alma mater' : 'WSU',
2  ...      42 : 'The meaning of life.',
3  ...      (3, 4) : {'first' : 33, 'second' : 3 + 4},
4  ...      5.7 : [5, 0.7]}
5  >>> d['alma mater']
6  'WSU'
7  >>> d[42]
8  'The meaning of life.'
9  >>> d[5.7]
10 [5, 0.7]
11 >>> d[5.7][1]
12 0.7
13 >>> d[(3, 4)]
14 {'second': 7, 'first': 33}
15 >>> d[(3, 4)]['second']
16 7
```

[2]Abraham Lincoln was the tallest President of the United States at 6 feet 4 inches. James Madison was the shortest at 5 feet 4 inches.

[3]The order is dependent on the hash function used. The details of this are something we can easily access and thus we simply need to accept that ordering of values in the dictionary is effectively random. Nevertheless, as we will see, using the `sorted()` function there are ways to order the data from dictionaries.

Let us now take a look at the methods provided by a dictionary. These are shown in Listing 14.3. The methods of interest to us are in slanted bold text. The `keys()` method provides the keys for a `dict` while `values()` provides the values. The method `items()` provides `tuples` of all the key-value pairs. These methods (and the `get()` method) will be discussed further in the following sections.

Listing 14.3 Methods provided by `dicts`.

```
1  >>> dir({})
2  ['__class__', '__contains__', '__delattr__', '__delitem__', '__doc__',
3    '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
4    '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
5    '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
6    '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
7    '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items',
8    'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

14.2 Cycling through a Dictionary

Although a `dict` is not a sequence like a `list` or a `tuple`, it is an *iterable* and can be used in a `for`-loop header. This is demonstrated in Listing 14.4 where the dictionary `abe` is created in line 1 and then used as the iterable in the `for`-loop in line 2. Here we use `foo` for the loop variable name to indicate that it is not obvious what value is assigned to this variable. The body of the loop (line 3) simply prints the value of the loop variable. We see in the output in lines 5 through 7 that the loop variable is assigned the values of the keys. By invoking the `keys()` method on the dictionary `abe`, the `for`-loop defined in lines 8 and 9 explicitly says that the iterable should be the keys of the `dict`. This loop is functionally identical to the loop in lines 2 and 3. Since it is superfluous, typically one does not invoke the `keys()` method in the header of a `for`-loop and instead writes the header as shown in line 2. (Another common idiom is to name the loop variable `key`.)

Listing 14.4 When a `dict` is used as the iterable in a `for`-loop, the loop variable takes on the values of keys.

```
1  >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2  >>> for foo in abe:
3  ...     print(foo)
4  ...
5  age
6  name
7  height
8  >>> for foo in abe.keys():
9  ...     print(foo)
10 ...
```

```
11 age
12 name
13 height
```

Of course, one is usually interested in the values associated with keys and not in the keys themselves. Listing 14.5 demonstrates the printing of keys together with their associated values. In the `print()` statement in line 3, the second argument is `':\t'`. Recall that `\t` is the escape sequence for a tab. This serves to align the output as shown in lines 5 through 7.

Listing 14.5 Display of key-value pairs.

```
1 >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2 >>> for key in abe:
3 ...     print(key, ':\t', abe[key], sep="")  # \t = tab.
4 ...
5 age:    203
6 name:   Abraham Lincoln
7 height: 193
```

An alternative way to display key-value pairs is provided by the `items()` method. As mentioned in the previous section, this method provides a pairing of keys and their associated values. In this way, simultaneous assignment can be used in the header of the `for`-loop to obtain both the key and the associated value. This is demonstrated in Listing 14.6. This listing is identical to Listing 14.5 except in line 2 (where `value` has been added as a loop variable and the `items()` method is invoked) and in line 3 (where `abe[key]` has been replaced by `value`).

Listing 14.6 Use of the `items()` method to obtain both the key and the associated value in the header of the `for`-loop.

```
1 >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2 >>> for key, value in abe.items():
3 ...     print(key, ':\t', value, sep="")
4 ...
5 age:    203
6 name:   Abraham Lincoln
7 height: 193
```

If we are interested in obtaining only the values from a `dict`, we can obtain them using the `values()` method. This is demonstrated in Listing 14.7.

Listing 14.7 Displaying the values in a dictionary.

```
1 >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2 >>> for value in abe.values():
```

```
3  ...     print(value)
4  ...
5  203
6  Abraham Lincoln
7  193
```

Assume a teacher has created a dictionary of students in which the keys are the students' names. Each student is assigned a grade (which is a string). The teacher then wants to view the students' names and grades. Typically such a listing is presented alphabetically. However, with a `dict` we have no way to directly enforce the ordering of the keys. For a `list` the `sort()` method can be used to order the elements, but this cannot be used with the keys of a `dict` because the keys themselves are not a `list` nor does the `keys()` method produce a `list`. Fortunately, Python provides a function called `sorted()` that can be used to sort the keys. `sorted()` takes an iterable as its argument and returns a `list` of sorted values.

In Listing 14.8, in lines 1 through 5, a dictionary of eight students is created. In lines 6 and 7 a `for`-loop is used to display all the student names and grades. Note that the `sorted()` function is used in the header (line 6) to sort the keys. For strings, `sorted()` will, by default, perform the sort in alphabetical order. The body of the `for`-loop consists of a single `print()` statement. A format string is used to ensure the output appears nicely aligned (because of the plus or minus that may appear in the grade, the width of the first replacement field is set to two characters). Note that the listing of students in lines 9 through 16 is in alphabetical order. The `for`-loop in lines 17 and 18 does not use the `sorted()` function to sort the keys nor is a format string used for the output. The subsequent output, in lines 20 through 27, is not in alphabetical order and the names are no longer aligned.

Listing 14.8 Use of `sorted()` to sort the keys of a `dict` and thus show the data "in order."

```
1  >>> students = {
2  ...    'Harry' : 'B+',  'Hermione' : 'A+', 'Ron' : 'B-',
3  ...    'Fred' : 'C', 'George': 'C', 'Nevel' : 'B',
4  ...    'Lord Voldemort' : 'F', 'Ginny' : 'A'
5  ... }
6  >>> for key in sorted(students):   # Sorted keys.
7  ...     print("{:2} {}".format(students[key], key))
8  ...
9  C  Fred
10 C  George
11 A  Ginny
12 B+ Harry
13 A+ Hermione
14 F  Lord Voldemort
15 B  Nevel
16 B- Ron
17 >>> for key in students:             # Unsorted keys.
18 ...     print(students[key], key)
19 ...
```

```
20 A+ Hermione
21 B- Ron
22 B+ Harry
23 B Nevel
24 F Lord Voldemort
25 A Ginny
26 C George
27 C Fred
```

14.3 get()

The dictionary method `get()` provides another way to obtain the value associated with a key. However, there are two important differences between the way `get()` behaves and the way dictionaries behave when a key is specified within brackets. The first argument to `get()` is the key. If the key does not exist within the dictionary, no error is produced. Instead, `get()` returns `None`. This is demonstrated in Listing 14.9. In line 1 a dictionary is created with keys `'age'` and `'height'`. The header of the `for`-loop in line 2 explicitly sets the loop variable `key` to `'name'`, `'age'`, and `'height'`. In the body of the loop, in line 3, the `get()` method is used to obtain the value associated with the given key. The output in line 5 shows that the method returns `None` for the key `'name'`. The `for`-loop in lines 8 and 9 is similar to the previous loop except here the values are obtained using `james[key]` rather than `james.get(key)`. This results in an error because the key `'name'` is not defined. This error terminates the loop, i.e., we do not see the other values for which a key is defined.

Listing 14.9 The `get()` method can be used to look up values for a given key. If the key does not exist, the method returns `None`.

```
1  >>> james = {'age': 261, 'height': 163}
2  >>> for key in ['name', 'age', 'height']:
3  ...     print(key, ':\t', james.get(key), sep="")
4  ...
5  name:   None
6  age:    261
7  height: 163
8  >>> for key in ['name', 'age', 'height']:
9  ...     print(key, ':\t', james[key], sep="")
10 ...
11 Traceback (most recent call last):
12   File "<stdin>", line 2, in <module>
13 KeyError: 'name'
```

The other important difference between using `get()` and specifying a key within brackets is that `get()` takes an optional second argument that specifies what should be returned when a key does not exist. In this way we can obtain a value other than `None` for undefined keys. Effectively

this allows us to provide a default value. This is demonstrated in Listing 14.10 which is a slight variation of Listing 14.9. The only difference is in line 3 where the string `John Doe` is provided as the second argument to the `get()` method. Note that this particular default value (i.e., `John Doe`) appears to be reasonable in terms of providing a missing "name," but it does not make much sense as a default for the age or height.

Listing 14.10 Demonstration of the use of a default return value for `get()`.

```
1 >>> james = {'age': 261, 'height': 163}
2 >>> for key in ['name', 'age', 'height']:
3 ...     print(key, ':\t', james.get(key, "John Doe"), sep="")
4 ...
5 name:   John Doe
6 age:    261
7 height: 163
```

Let's now consider a more practical way in which the default value of the `get()` method can be used. Assume we want to analyze some text to determine how often each "word" appears within the text. For the sake of simplicity, we will assume a word is any collection of contiguous non-whitespace characters. Thus, letters, digits, and punctuation marks are all considered part of a word. Furthermore, we will maintain case sensitivity so that words having the same letters but different cases are considered to be different. For example, all of the following are considered to be different words:

```
  end  end.  end,  End  "End
```

Analysis of text in which one obtains the number of occurrences of each word is often referred to as a *concordance*. As an example of this, let's analyze the following text to determine how many times each word appears:

```
How much wood could a woodchuck chuck if a woodchuck could chuck
wood?  A woodchuck you say?  Not much if the wood were mahogany.
```

We can do this rather easily as demonstrated by the code in Listing 14.11. This code is discussed following the listing.

Listing 14.11 Analysis of text to determine the number of times each word appears.

```
1 >>> text = """
2 ... How much wood could a woodchuck chuck if a woodchuck could chuck
3 ... wood?  A woodchuck you say?  Not much if the wood were mahogany.
4 ... """
5 >>> concordance = {}
6 >>> for word in text.split():
7 ...     concordance[word] = concordance.get(word, 0) + 1
8 ...
9 >>> for key in concordance:
```

```
10 ...     print(concordance[key], key)
11 ...
12 2 a
13 2 wood
14 1 A
15 1 mahogany.
16 1 say?
17 2 could
18 2 chuck
19 1 How
20 2 much
21 3 woodchuck
22 1 were
23 1 Not
24 1 you
25 1 wood?
26 1 the
27 2 if
```

In lines 1 through 4 the text is assigned to the variable `text`. In line 5 an empty dictionary is created and assigned to the variable `concordance`. This dictionary will have keys that are the words in the text. The value associated with each key will ultimately be the number of times the word appears in the text.

The `for`-loop in lines 6 and 7 cycles through each word in `text`. In the header, in line 6, this is accomplished by using the `split()` method on `text` to obtain a `list` of all the individual words. The body of the `for`-loop has a single assignment statement (line 7). On the right side of the assignment statement the `get()` method is used to determine the number of previous occurrences of the given key/word. If the word has not been seen before, the `get()` method returns `0` (i.e., the optional second argument is the integer `0`). Otherwise it returns whatever value is already stored in the dictionary. The value that `get()` returns is incremented by one (indicating there has been one more occurrence of the given word) and this is assigned to `concordance` with the given key/word.

The `for`-loop in lines 9 and 10 is simply used to display the concordance, i.e., the count for the number of occurrences of each word. We see, for example, that the word `wood` occurrs twice while `woodchuck` occurrs three times.

14.4 Chapter Summary

Dictionaries consist of key-value pairs. Any object consisting of immutable components can be used as a key. Values may be any object. Dictionaries themselves are mutable.

Dictionaries are unordered collections of data (i.e., they are not sequences).

The value associated with a given key can be obtained by giving the dictionary name followed by the key enclosed in square brackets. For example, for the dictionary `d`, the value associated with the key `k` is given by `d[k]`. It is an error to attempt to access a non-existent key this way.

The **get()** method can be used to obtain the value associated with a key. If the key does not exist, by default, `get()` returns `None`. However, an optional argument can be provided to specify the return value for a non-existent key.

The **keys()** method returns the keys of a dictionary. When a dictionary is used as an iterable (e.g., in the header of a `for`-loop), by default the iteration is over the keys. Thus, if `d` is a dictionary, "`for k in d:`" is equivalent to "`for k in d.keys()`".

The **sorted()** function can be used to sort the keys of a dictionary. Thus, "`for k in sorted(d):`" cycles through the keys in order. (Similar to the `sort()` method for `lists`, the order can be further controlled by providing a key function whose output dictates the values to be used for the sorting. Additionally, setting the optional argument `reverse` to `True` causes `sorted()` to reverse the order of the items.)

The **values()** method returns the values in the dictionary. Thus, if `d` is a dictionary, "`for v in d.values():`" cycles through the values of the dictionary.

The **items()** method returns the key-value pairs in the dictionary.

14.5 Review Questions

1. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
print(d['c'])
```

   (a) `c`
   (b) `2`
   (c) `'c' : 2`
   (d) This code produces an error.

2. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
print(d[2])
```

   (a) `c`
   (b) `2`
   (c) `'c' : 2`
   (d) This code produces an error.

3. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
print(d.get(2, 'c'))
```

(a) `c`
(b) `2`
(c) `'c' : 2`
(d) This code produces an error.

4. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d):
   print(d[x], end=" ")
```

(a) `a b c`
(b) `0 1 2`
(c) `('a', 0) ('b', 1) ('c', 2)`
(d) This code produces an error.

5. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d.values()):
   print(x, end=" ")
```

(a) `a b c`
(b) `0 1 2`
(c) `('a', 0) ('b', 1) ('c', 2)`
(d) This code produces an error.

6. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d.items()):
   print(x, end=" ")
```

(a) `a b c`
(b) `0 1 2`
(c) `('a', 0) ('b', 1) ('c', 2)`
(d) This code produces an error.

7. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d.keys()):
   print(x, end=" ")
```

(a) `a b c`
(b) `0 1 2`
(c) `('a', 0) ('b', 1) ('c', 2)`
(d) This code produces an error.

8. What is the output produced by the following code?

```
pres = {'george' : 'washington', 'thomas' : 'jefferson',
        'john' : 'adams'}
print(pres.get('washington', 'dc'))
```

(a) `george`
(b) `washington`
(c) `dc`
(d) This code produces an error.

9. What is the output produced by the following code?

```
pres = {'george' : 'washington', 'thomas' : 'jefferson',
        'john' : 'adams'}
for p in sorted(pres):
    print(p, end=" ")
```

(a) `george thomas john`
(b) `george john thomas`
(c) `washington jefferson adams`
(d) `adams jefferson washington`
(e) None of the above.

ANSWERS: 1) b; 2) d; 3) a; 4) b; 5) b; 6) c; 7) a; 8) c; 9) b;

Appendix A

ASCII Non-printable Characters

Table A.1: The ASCII non-printable characters.

Value	Abbr.	Name/description
0	NUL	Null character, \0
1	SOH	Start of Header
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transmission
5	ENQ	Enquiry
6	ACK	Acknowledgment
7	BEL	Bell, \a
8	BS	Backspace, \b
9	HT	Horizontal Tab, \t
10	LF	Line Feed, \n
11	VT	Vertical Tab, \v
12	FF	Form Feed, \f
13	CR	Carriage Return, \r
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	Device Control 1 (XON)
18	DC2	Device Control 2
19	DC3	Device Control 3 (XOFF)
20	DC4	Device Control 4
21	NAK	Negative Acknowledgement
22	SYN	Synchronous idle
23	ETB	End of Transmission Block
24	CAN	Cancel
25	EM	End of Medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Record Separator
31	US	Unit Separator
127	DEL	Delete

Index

Symbols

!= . . . *see* not equal to
\ . . . *see* escaping or escape sequences
** . . . *see* exponentiation
/ . . . *see* division, float
// . . . *see* division, floor
< . . . *see* less than
<= . . . *see* less than or equal to
= . . . *see* assignment
== . . . *see* equal to
> . . . *see* greater than
>= . . . *see* greater than or equal to
. . . *see* comment
% . . . *see* modulo

A

accumulator . . . 129–130
alias . . . 158, 330
and . . . 273–275
application . . . 2f
argument . . . 5, 5f
argument, complex number . . . 182
arithmetic operators . . . 22–24
ASCII . . . 195, 199–200
assignment . . . 24–28
- augmented . . . 40–42
- cascaded . . . 27
- operator . . . 24
- simultaneous . . . 27–28
  - lists . . . 126–127

attributes . . . 96

B

base
- 2 . . . 19
- 3 . . . 19
- 10 . . . 19

binary operator . . . 24
bit . . . 1
body mass index, (BMI) . . . 58–59
bool . . . 256
bool() . . . 257
Boolean . . . 255–258
- expression . . . 255–256
- operator . . . *see* logical operator
- type . . . 256

break . . . 278–280
buffered output . . . 248
bugs . . . 3, 12–13
- semantic . . . 3
- syntactic . . . 3f, 3

C

chr() . . . 203–208
cipher text . . . 206
class . . . 21
class . . . 97–101
clear text . . . 205
close() . . . 242–243
cmath . . . 185–189
code . . . 2
comments . . . 10
comparison
- multiple . . . 275–276
- operator . . . 261–266

compiler . . . 8
complex . . . 19f, 179, 181–187
concatenate
- list . . . 111
- strings . . . 106

conditional statement . . . 255–289
- compound . . . 267

continue . . . 281–283
current working directory . . . 239

D

Decimal 19f
decryption 205
def 69–70
del() 32f
dir() 101–103
division
- float, / 23
- floor, // 37–38, 184

divmod() 38–40
docstring 74

E

empty
- list 111
- string 21

encryption 205
enumerate() 246
equal to, == 263
escape sequences 200–203
escaping 29–30
eval() 57–59
exception 13
- IndexError 116, 168, 220
- KeyError 222, 336–337, 341
- NameError . 13, 76, 78–79, 179–180, 188
- SyntaxError 13, 85, 183, 202
- TypeError . . . 52–53, 69, 79–80, 84, 106, 123, 185, 198
- ValueError 54–56, 185, 188–189, 212–213, 224, 242

exponential notation 20
exponentiation, ** 37
expression 11f, 22–24

F

False 256
- objects considered False 257

Fibonacci sequence 130–132
file object 240–241
- as iterable 245–247

float 19–20, 20f
float() 53–56
for-loop 113–115
format string 218–230
formatting strings . . . . . . *see* string, format()
Fraction 19f

G

greater than or equal to, >= 263
greater than, > 262

H

hello world 3–6, 9–12
help() 13–14

I

identifier 24
if 255–261
if-elif-else 270–272
if-else 267–270
immutable 121–123
import 177–193
- of an entire module 179–181
- of multiple modules 186–187
- using * 189–190
- using as 187
- using from 187–189
- your module 190–193

in 114, 287–289
index 115
indexing 115–117
infinite loop 278–280
__init__() 103–105
input() 51–53
instance 21
int 19–20
int() 53–56
interactive environment
- echoing of expressions . . . 10–11, 20–21, 25
- prompt 9–10, 29

interpreter 3, 8
is 158
iterable 113

K

keywords 31

L

len() 196
- of list or tuple 111

- of string 162
- less than or equal to, <= 263
- less than, < 263
- `list` 110–113
  - `append()` 112, 113
  - `extend()` 112, 113
  - `sort()` 112, 113
- `list` comprehension 168–171, 264f
- `list()` 118
- literal 19–22
  - numeric 19
  - string 4, 11
    - multi-line 29
- logical operator 272–275
- loop variable 114
- lvalue 24, 121

M

- magic number 40
- magnitude 182
- `main()` 81–82
- methods 96
- mode . . . . *see* `open()`, mode
- modulo, `%` 38–40, 184
- modulus 182
- mutable 121–123

N

- name 24
- named constant 40
- namespace 32–36, 76–77
- nesting
  - `if` statements 264
  - `for`-loops in functions 123–126
  - functions 55
- newline character, `\n` 29–30
- non-void functions 72–75
- `None` 68–69
- `not` 273–275
- not equal to, `!=` 263

O

- object oriented programming 21
- `open()` 239–251
  - mode 239, 247
- operator
  - binary 23, 263, 273
  - unary 23, 273
- operator overloading 105–106
- `or` 273–275
- `ord()` 203–208

P

- parameter 5, 5f
  - formal 70
  - optional 82–86
- parameters
  - actual 70
- parentheses 12
  - for multi-line statements 29
  - functions 4–5
  - precedence 23, 273
- `pass` 326
- path 193, 239
- phase 182
- precedence
  - arithmetic operators 23, 38, 39
  - comparison operators 262
  - logical operator 273
- `print()` 3–6, 9, 13–14
  - `end` 5, 13
  - `file` 249
  - `sep` 13, 40
- program 2

Q

- quotes 19
  - double 4, 20
  - repeated three times 29f, 29
  - single 20
  - within a string 28

R

- `range()` 118–121
- `read()` 240–243, 247
- `readline()` 240, 241, 243–244, 247
- `readlines()` 240, 241, 244–245, 247
- recursion 297–310
  - depth 298
- recursive . . . . *see* recursion
- reference 158, 330
- relational operator . . . . *see* comparison, operator

repetition . . . 106
replacement field . . . 218–224
`__repr__()` . . . 211–212
`return` . . . 70
`round()` . . . 92

S

scope . . . 76–79
  functions . . . 78–79
  variables . . . 76–78
semantic error . . . 183
semantics . . . 2
shorcircuit behavior . . . 283–287
simultaneous assignment . . . *see* assignment, simultaneous
statement . . . 24
  multi-line . . . 29
`str` . . . 21
`__str__()` . . . 197
`str()` . . . 56–57, 128
stream position . . . 241–245
string . . . 4–5, 195–231
  `capitalize()` . . . 209–210
  `count()` . . . 210
  `find()` . . . 212–215
  `format()` . . . 218–230
    alignment . . . 223–224
    fill . . . 224–225
    format specifier . . . 222–228
    precision . . . 225–226
    type specifier . . . 226–227
    width . . . 222–223, 225–226
  `index()` . . . 212–215
  `join()` . . . 215–218
  `lower()` . . . 209–210
  `lstrip()` . . . 210–211
  methods . . . 209–230
  `replace()` . . . 215
  `rstrip()` . . . 210–211, 246
  `split()` . . . 215–218
  `strip()` . . . 210–211
  `swapcase()` . . . 209–210
  `title()` . . . 209–210
  `upper()` . . . 209–210
`sum()` . . . 260–261, 282f
syntactic error . . . 13, 183
syntax . . . 2, 95, 273, 335

T

`tell()` . . . 241–243
test expession . . . 257–260
text . . . 195–196
  plain . . . 195
  vs. strings . . . 195
`True` . . . 256
  objects considered `True` . . . 257
`tuple` . . . 26f, 122
`type()` . . . 21

V

variable . . . 24
void functions . . . 68–72

W

`while`-loop . . . 276–283
whitespace . . . 201
  stripping . . . 210
`write()` . . . 248
`writelines()` . . . 250–251