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.

```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.

```python
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.

```python
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.

```
1 "Hello World!"
2 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

```
1 >>> =========================== RESTART ============================
2 >>>
3 Have we said enough hellos?
4 >>>
```

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.)

```
 1  >>> # Can I write Print()?
 2  >>> Print("Hello World!")
 3  Traceback (most recent call last):
 4    File "<stdin>", line 2, in <module>
 5  NameError: name 'Print' is not defined
 6  >>> # Can I get rid of the parentheses?
 7  >>> print "Hello World!"
 8    File "<stdin>", line 2
 9      print "Hello World!"
10                         ^
11  SyntaxError: invalid syntax
12  >>> # Do I need the quotation marks?
13  >>> print(Hello World!)
14    File "<stdin>", line 2
15      print(Hello World!)
16                      ^
17  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.