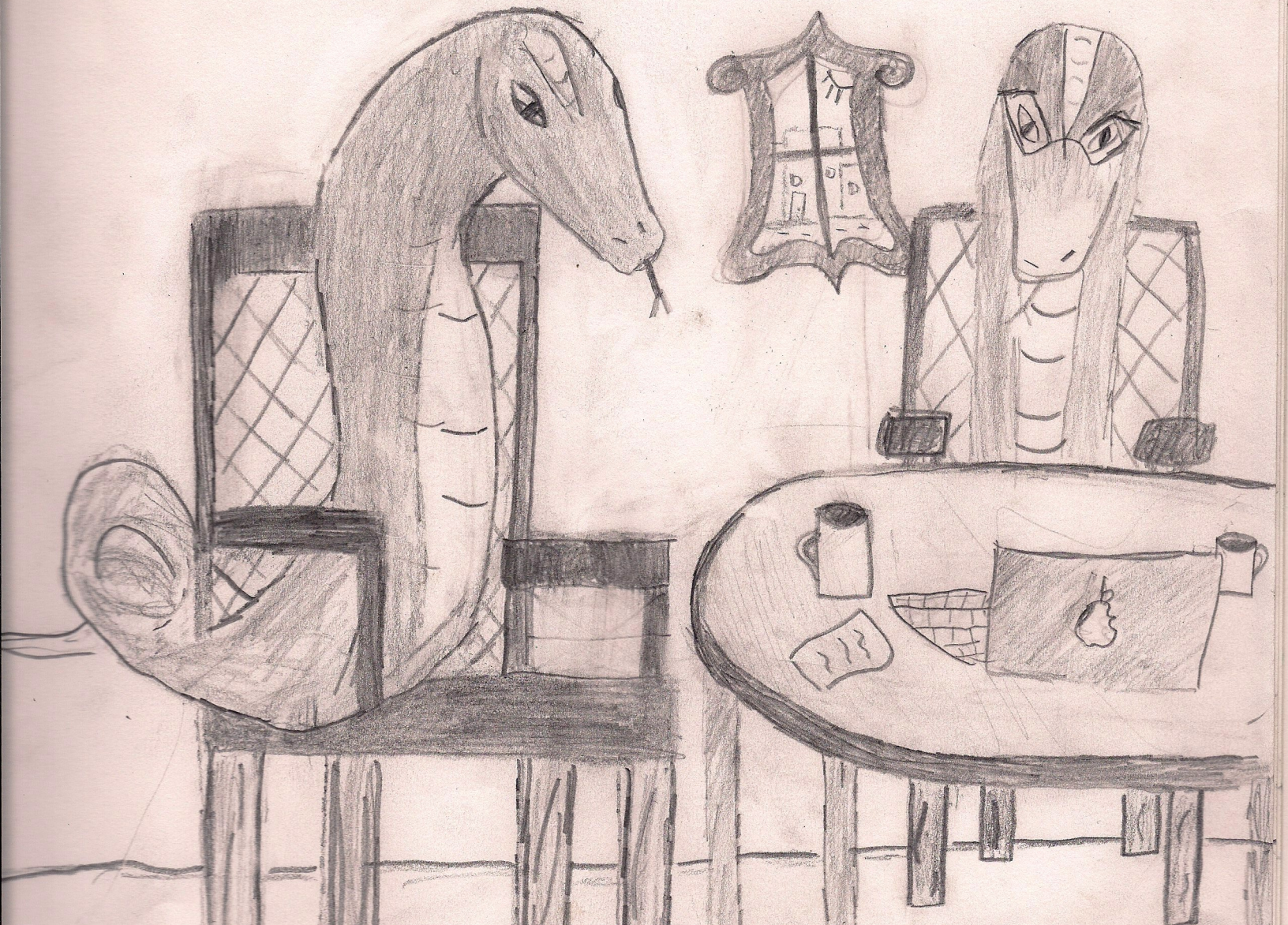
Conversational python



Conversational Python

An Introduction to Computer Science Experienced Through Conversational Banter

Jason B. Shepherd, Ph.D.

Contents

Preface

Perhaps some day this book will have a real preface – one in which the author waxes philosophically about the wonders of this textbook, how it was a labor of love, and how you should think very intentionally about the book's organization despite the fact that you (the reader) really don't care. For now, I shall simply thank the following.

- My wife Lisa and my children Ian, Noah, and Tatum for their support of my teaching career, and especially my daughter Tatum for drawing the cover for this book. She was 11 years old at the time she drew it.
- My CMSC 181 Intro to Computer Science students from Fall 2016 for providing detailed and thoughtful feedback on the first draft of this book.
- My CMSC 181 Intro to Computer Science Students from Fall 2017 for continuing the journey of engaging this book. Special thanks to Tyler Freese for finding the most inaccuracies, mistakes, etc., in this tome (and especially for being polite about it).
- Everyone at Buena Vista University for making the university a remarkable place, especially my colleagues in computer science and data science: Nathan Backman, Anton Bezuglov, Ben Donath, and Shawn Stone.
- My God, for allowing me to live and serve as I do.

On to Python!

Chapter 1

Basic I/O, Statements, Expressions, Variables, and Types

1.1 Getting set up to program

Computers consist of hardware and software. Hardware is the physical stuff you can hold in your hands, like memory, disks, keyboards, etc. Software tells the hardware what to do. The hardware stores the software and "runs" it. In this book, we're going to learn how to create software. Creating software is called *programming* or *coding*. When coding, we give the computer instructions for it to do.

Typically, hardware consists of a bunch of electrical circuits. Instructions for controlling the circuits are called *machine code* (which is basically a bunch of 1's and 0's that tell the computer which electrical circuits to turn on and off… beep beep boop boop). However, we will not need to learn machine code or worry about electrical circuits. We will write code that looks something more like normal human language, and then we will use another software program called a *compiler* or an *interpreter* to translate our nice, readable code into machine code that will control the hardware. Nice, eh?

The code we write will be written in a high-level programming language

named *Python*. Although programming languages might look technical, they are more human reader-friendly than the hardware's machine code language. Figure 1.1 shows the relationship between Python language code, the interpreter, and the hardware.

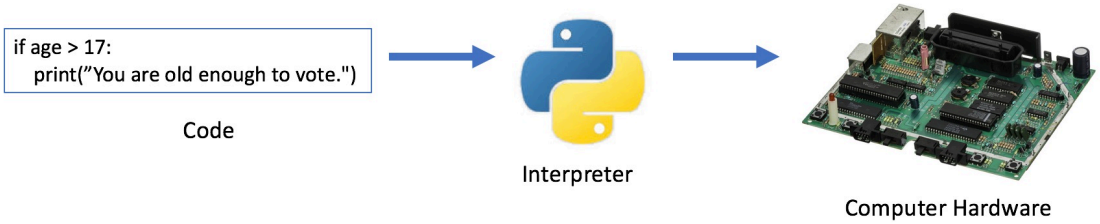


Figure 1.1: Code translated by an interpreter to hardware language

To get started programming in Python, we will need to install two software programs on our computer. One program will help us write code for the computer to execute. The second program will run our code so that we can see what it does. The first program is named *Thonny*. The second program is simply known as the *Python interpreter*. Fortunately, we don't need to download these two programs separately. When we download and install Thonny, Thonny contains the Python interpreter as well for free!

Let's install Thonny, and by extension, the Python interpreter.

- Open your favorite Web browser, again (mine is Google Chrome).
- In your Web browser, go to the Web address http://thonny.org/.
- The Thonny Web page will appear. In the upper-right hand you will see options to download Thonny for your computer. If you have a Windows computer, choose Windows. If you have Mac, choose Mac. If you have Linux, choose Linux.
- Find the file that you've downloaded and double-click on it to run it.

- An installer program will be displayed. Follow the steps in the installer to finish installing Thonny. If you are unsure of what to do at a particular step, accept the default information in that step.

Now, find the Thonny program on your computer and run it. The main Thonny window is shown in Figure 1.2.

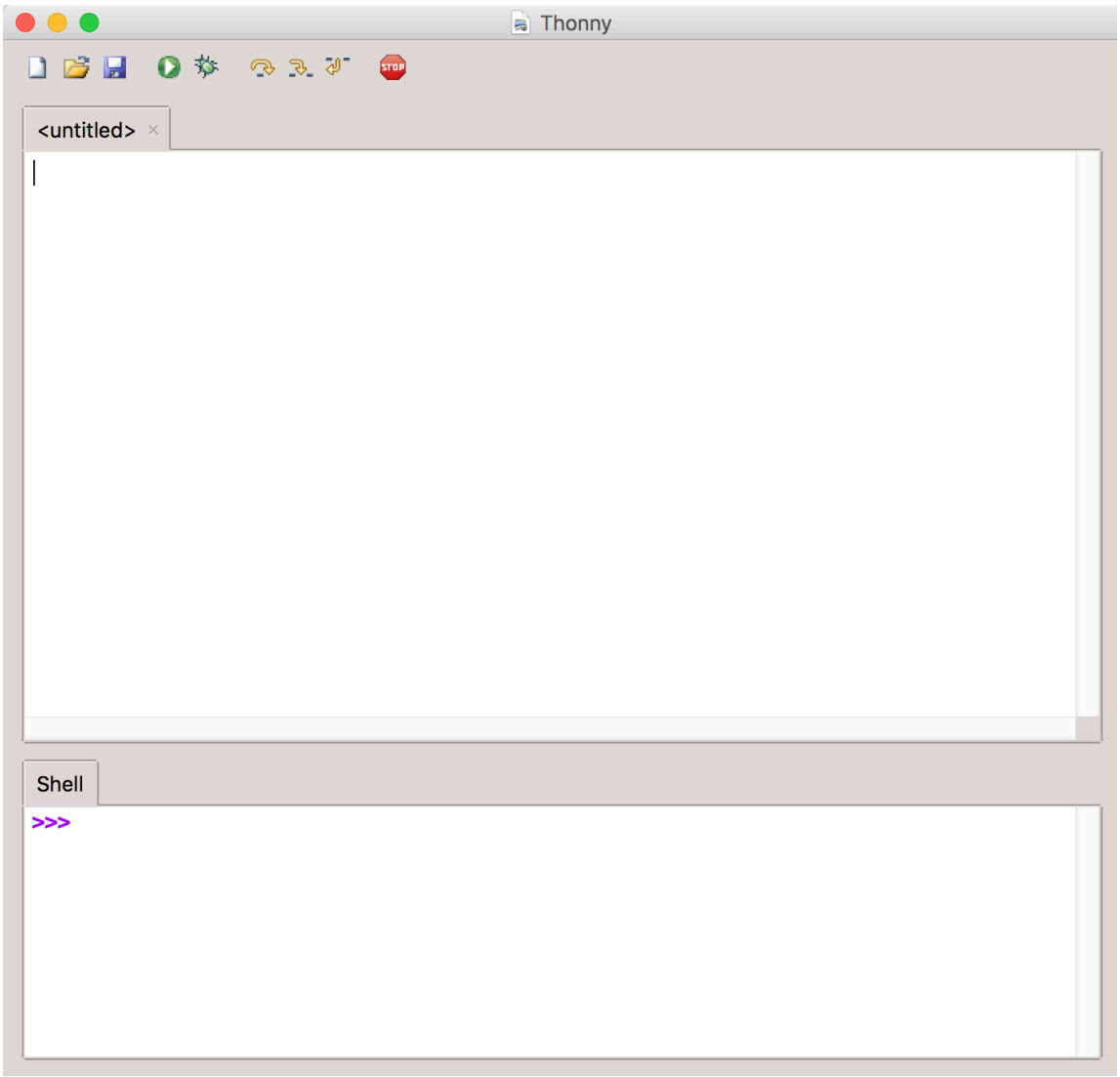


Figure 1.2

Notice that there are two "parts" to the Thonny window. The top part is where you type Python code. The bottom part is called the Python *Shell* window. The shell is where you will see the result of your code, which we call the *output*.

Normally, you'll be able to write code in code window, save it to a file, and then run it to see the output in the Python Shell window. Additionally, you can type single lines of code into the Python Shell window if you want to experiment a little bit before writing your finished program code. We'll encourage you to do both types of activities throughout this book.

Notice that the code window in Thonny displays something like "untitled." Let's go ahead and start saving any code we type into a file. That way, after we close Thonny we can always come back and work on our code some more later. In Thonny, choose **File –> Save As...** from the top menu. A dialog window will appear. Choose an appropriate folder location for your new file (and all other Python code files you'll create in this book), and then let's name this first file **`hello.py`**. Python files always end with the suffix **`.py`**. This helps us remember what type of file it is. We call these files Python source code files, or simply Python source files. Click the **Save** button and return to the Thonny code window.

Okay, let's learn some Python!

1.2 The `print` statement

You are now ready to start typing *statements* into the file **`hello.py`**. These statements will make up a program. Each statement gives the computer an instruction. A statement might tell the computer to put something on the screen (which is called "printing"), it might prompt the user to type something, or it might tell the computer to remember some information so that we can recall it and use it later in the program, etc.

Any time from here forward that you click the play button in Thonny, Thonny will hand your statements to the Python interpreter. The Python interpreter will take each statement, one at a time, and change the statement into machine code (again, 1's and 0's... beep beep boop boop). If any of your statements put words

or numbers on the screen, those will show up in the Python Shell window.

Let's try to write a program that makes some text show up in the Shell window.

So, let's type the following into your new **hello.py** file.

```
print("Hello")
```

Run your program by pressing the play button.

What do you see? You should see in your Python Shell window the word **Hello** on a separate line. If you don't, ask your friendly neighborhood programmer for help.

Cool. Why does this work, and can we break it and learn something in the process? The **print** statement allows us to put words and numbers on the screen, that is, in the Python Shell window. You type print, and then in parentheses, you put what you want to appear on-screen. What are those double quotes doing there? Let's get rid of them and then run the program to see what happens. Replace your code with this:

```
print(Hello)
```

Notice the double quotes are gone. Run your program using play button again.

Oh no! We've broken our computer! Well, not really. The word **print** means something to Python–it's part of the language–but **Hello** doesn't mean anything to Python. The double quotes tell Python to actually print the text "Hello" onto the screen.

Let's delete the one line we have in our file, and replace it with the following three lines.

```
print("Hello")
print("How are you?")
print("Goodbye.")
```

Run your code again and you will see these three lines appear on-screen in order.

What if you change the order of the statements?

```
print("Hello")
print("Goodbye.")
print("How are you?")
```

Notice we switched the order of the last two statements. Python will only execute your statements in the order you give them. Consider the following analogy: programs are to computers as recipes are to cooking. The order of the statements matter just like the order of steps in a recipe matter.

1.3 The `input` statement

Okay, change our code again. Delete the three lines you have so far and then add one new one.

```
print("Hello, Steve.")
```

This code attempts to make our program more personalized, but it makes the bold (and unfortunate) assumption that the user's name will always be Steve. What if we wanted to ask users what their names are, and then greet the user by name? How many statements would we need? We would need two: 1.) to ask for the name, and 2.) to greet the person using that name. Here's a good first attempt.

```
print("What is your name? ")
print("Hello, name.")
```

What do you see? The code does ask for the user's name (that's good), but it does not give the user the ability to type anything in (that's bad).

Let's introduce a new type of statement called **input**. All programs take *input* from the user (possibly from the keyboard, a mouse, or something else) and produce *output* (usually information is *printed* to the screen, but the information could be placed elsewhere, too, like placed in a file or sent over the Internet to a Web site or something). The **input** statement will allow the user to type something in.

Let's try this. Change the first **print** statement to an **input** statement.

```
input("What is your name? ")
print("Hello, name.")
```

Run this code. What happens?

Cool! The user (you) can now type in your name. But, then the program fails to address the person (you, again) using that name. Bummer.

We need to make the program remember the person's name in the first statement so that it can be used later in the second statement.

Let's change the first statement from this

```
input("What is your name? ")
```

to this

```
firstname = input("What is your name? ")
```

See the difference? We've put **firstname =** in front of the input command.

Here's how it works. The **input** command retrieves the text the user types in from the keyboard. Then, we must *store* it somewhere so that we can use it later in the program. The word **firstname** is a *variable*. Variables are kind of like Post-It notes for the computer to help it remember numbers and text that are important to us. *Variables* store *values*.

We can choose to name the variable almost whatever we want. There are some rules for what you can and can't name a variable. Variables must start with

a letter or an underscore (_). After that, they can include any letters, numbers, or underscores, but no other symbols. Variables cannot have spaces in their name. You also cannot name a variable one of the *reserved words* in Python. That is, there are commands that mean something to Python, like **`if`** or **`while`**. Always name the variable so that you'll remember its name. Instead of

```
firstname = input("What is your name? ")
```

We could have typed

```
dudename = input("What is your name? ")
```

or

```
awesomename = input("What is your name? ")
```

but for now we'll stick with

```
firstname = input("What is your name? ")
```

Now, what about the second line? It's still

```
print("Hello, name.")
```

We need to use the variable **`firstname`** to retrieve the value we stored. Change the print line to this

```
print("Hello,", firstname)
```

Run it. Voila! It works!

Now, instead of putting one thing inside the parentheses after the word **print**, we're listing two things, separated by a comma. The first is a text string **"Hello,"**. The second is the variable that stores the name. Print will print both of those things, separated by a space.

What if we don't want spaces between the things we print? We'll get to that later, too.

Awesome. Try some stuff. Try to break your code. Don't worry, you can always change your code back. Our code so far should look like the code in Listing 1.1.

Listing 1.1

```
1 firstname = input("What is your name? ")
2 print("Hello,", firstname)
```

Sometimes in this book we will show code snippets like we had been previously. Other times, if we want to show a series of statements in context, we will use a listing like we did above. The listings will typically have line numbers before each code statement so that we can draw attention to individual statements if we wish.

Let's learn by breaking things. In Listing 1.1, change the second line by removing **first** from the variable name. The code should now read

```
firstname = input("What is your name? ")
print("Hello,", name)
```

Run this code. What happens? Why do you think this happens?

Once you create a variable named **firstname**, it's called **firstname** for the duration of the program. In the first line, we're creating **firstname**. Then, in the second line, we try to use a variable called **name**, but there is no such variable called **name**. Python rightly vomits red text all over the screen.

It would be like if your name was Jorge and I tried to get your attention by yelling "Hey, Betty!" You wouldn't know I was trying to get your attention.

Let's try one more thing. Let's go back to our original code in Listing 1.1.

Change the variable **firstname** in the second line by capitalizing the first letter. In other words, change **firstname** to **Firstname**, like this.

```
firstname = input("What is your name? ")
print("Hello,", Firstname)
```

Now run it and see what happens. You get an error, don't you? **Firstname** and **firstname** are different variables. Python is a *case-sensitive* language. So we don't make mistakes with mixing uppercase and lowercase in Python, we typically stick to lowercase.

Remember how I said variable names can't have spaces in them? Let's try it anyway. What if we changed **firstname** to **first name**?

```
first name = input("What is your name? ")
print("Hello,", first name)
```

Run it and watch our program crash and burn. It's a good things computers don't have feelings because ours would probably be feeling rather abused right now.

Notice that each time we tried to "break" our code, we ended up with different program errors. Pay attention to what the errors say. At first, the errors look like "tech-ese" but eventually you'll learn to make sense of them, and it will help you in correcting your programs.

One cool thing about programming is it will give you a keener eye, and you'll notice details and mistakes a lot better in other avenues of your life. Well, we'll hope so anyway.

1.4 Assignment statements

As it turns out, we now know a lot about Python even though we likely don't realize it. We know three kinds of statements.

1. **print** statements

2. **input** statements

3. *assignment* statements

An assignment statement is a statement that creates or updates the value of a variable. Our input statement in the previous example was also an assignment statement because it created the variable named **firstname**.

Let's look at more examples of assignment statements. Consider the following. I live on an acreage, and we have a barn where cats tend to gather. We didn't have to buy any cats; they just show up. It's good they are around because they eat mice and we don't like mice.

Suppose we want to store the number of cats we have in our barn at any given time, and suppose we currently have six cats. To store this information in a Python variable, we would type the following.

```
cats = 6
```

To experiment with what's happening here, let's add two more lines so that your code now looks like Listing 1.2.

Listing 1.2

```
1 cats = 6
2 print(cats)
3 print("cats")
```

Can you guess what will happen when you run this code? It is very important to be able to read code line by line to figure out in your mind what will happen. Later on, when you're programming and something doesn't work right, you'll need to look at your code line by line to make sure that it makes logical sense. This code will print

```
6
cats
```

There are two print statements in Listing 1.2. Since there are two print statements, we can reasonably assume there will be two lines that appear on the screen. The first statement makes a new variable named **cats** and assigns it the value **6**. The second statement prints the value stored in cats. The third statement prints the text string "cats" since there are double quotes around it.

If we hadn't created the variable **cats** before printing the value of **cats**, Python would have puked red text again saying that it doesn't know what **cats** is.

It's worth noting that the first statement line **cats = 6** does not print anything to the screen. It's just an assignment statement. It's only job is to *define* a new variable. It does not print anything. Only **print** statements actually print anything on the Python Shell window.

Okay, let's try to break things again! What happens if we switch the order of **cats** and the number **6** in the first line of Listing 1.2, like this.

```
6 = cats
```

Run it. Kablooey! Now we've learned a rule about assignment statements. Think of the = sign as being more like a left arrow ←. When we write **cats = 6**, we can think of it like **cats** ← **6** in that the value **6** is being assigned to the new variable **cats**. You cannot change the order of **cats** and **6**.

See? Computers are picky.

So, on the left-hand side of the equals sign in an assignment statement, we can give the name of a variable. If the variable doesn't exist, it is created brand new. If the variable does already exist, the value of the variable is updated (or, overwritten – update and overwrite are synonymous). So what can we put on the right-hand side of the equals?

On the right-hand side, we can put any *expression* that produces a value. Here are some examples of expressions.

- **6**
- **2 + 4**
- **6 * 1**

- `8 - 2`
- `12 / 2`
- `12 – (3 * 2)`

The asterisk (`*`) or star does multiplication. The forward slash (`/`) performs division. The double forward slash (`//`) performs whole number division by dropping the remainder. For example, `5 // 2` produces the value `2`. The percent (`%`) gives us the remainder. This operation is also known as the modulus, or simply mod. For example, `5 % 2` produces the value `1`.

The `+`, `-`, `*`, `/`, `//`, and `%` are known as *operators*. Operators take two expressions (which produce values) and produces a new value. Parentheses can be used to group expressions much like we do in mathematics. Also as in mathematics, operators have precedence. Expressions produce a value by evaluating the expression left-to-right while performing parenthetical expressions first, then multiplication and division, and finally addition and subtraction.

Just for the heck of it, we could re-write the assignment statement `cats = 6` as

```
cats = 12 - (3 * 2)
```

To recap, the left-hand side (abbreviated *LHS*) of an assignment statement is the name of a variable. The variable will either be created (that is, defined) if it doesn't already exist or updated (that is, overwritten) depending on whether the variable existed previously. The right-hand side (abbreviated *RHS*) is an expression that, when evaluated by Python, produces a value.

With all this in mind, let's try something. What happens when you change our code so that it looks like Listing 1.3.

Listing 1.3

```
1 cats = 6
2 cats = cats + 1
3 print("You have", cats, "cats.")
```

Remember how assignment statements work. They work in two steps.

1. We determine the value produced on the RHS.
2. That value is assigned to the variable on the LHS.

What is the value on the RHS of `cats = cats + 1`? When we reach this statement, `cats` is `6`. Thus, `cats + 1` is the same as `6 + 1`, which is `7`. Therefore, the assignment statement `cats = cats + 1` is essentially the same as saying `cats = 7`.

We can visualize how this assignment statement is *evaluated* in Figure 1.3.

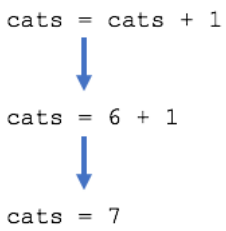


Figure 1.3: Evaluating an assignment statement

Remember, the equals sign performs assignment of the value produced by the RHS to the variable on the LHS. Therefore, the equals sign is more like a left arrow that the mathematical equals sign that says "the thing on the LHS and the thing on the RHS have the same value." The fact that you can type `x = x + 1` in Python looks yucky to mathematicians. Poor, poor mathematicians…

We will say more about the relationship between values, expressions, and statements in the next section.

1.5 Values, types, expressions, and statements

Let's modify the code example from the previous section. It is, after all, silly because you start with a fixed number of cats when, in fact, we could start with any number of cats. Let's ask users for how many cats they have in their barn.

Then, our program should calculate the number of cats they'll have in the barn in a month's time. For the sake of argument, let's suppose the number of cats will increase by 4.

A running program that solves this problem would look like this on the Python Shell window.

```
How many cats do you have? 10
In a month's time, you will have 14 cats!
```

The 10 at the end of the first line is typed in by the user, as an example. Can you write this program? Give it a try. You know how to write code to get input from the user. You know how to produce output. You know how to create variables and perform calculations using expressions. Try to write this program.

(One attempt at a solution follows, but try to shield your eyes and don't look at it right away. You won't learn very well if you don't try and fail every now and then.)

Listing 1.4 shows an attempt that you might have made.

Listing 1.4

```
1 cats = input("How many cats do you have? ")
2 cats = cats + 4
3 print("In a month's time, you will have", cats, "cats.")
```

This seems reasonable and logical, but it doesn't work! You end up with an error. Let's take a very close look at the error. It's very important to understand how to read error messages. If you know how, they often tell you exactly what's wrong.

```
Traceback (most recent call last):
  File "/Users/shep/cs1/code/cats.py", line 2, in <module>
    cats = cats + 4
TypeError: Can't convert 'int' object to str implicitly
```

The error states that the problem is on line 2 of your code. That's helpful, but keep in mind line 2 is only where the error was detected. It's possible that what caused the error occurred earlier in the code than line 2.

Note what the error message says: "Can't convert 'int' object to str implicitly." What does this mean? In order to understand what it means, we need to revisit some of the concepts we touched on in Section 1.4.

A *statement* can consist of one or more expressions. An *expression* is a piece of code that produces a *value*. Every value has a *type*. Consider the following example.

```
carrots = (7 + 3) * 2
```

This is one assignment statement whose right-hand side (RHS) consists of two expressions. The first expression is

```
7 + 3
```

7 and **3** are values known as *integers*, which is just a fancy word for "whole number." In Python, an integer is called an **int**, for short. **7** is a value and its type is **int**. **3** is value and its type is **int**. Since **7** is an **int** and **3** is an **int**, adding them together gives us the value **10**, which is also an **int**. Thus, the following table describes what we know about this expression.

Expression	Value	Type
7+3	**10**	int

Remember, an expression is a piece of code that produces a value. Every value has a type.

Say it again: an expression produces a value, and every value has a type.

If you're not sure what the type of an expression is, you can find out by typing the expression into the Python Shell. If I entered **7 + 3**, the Python Shell would output **10**. If I entered **type(7 + 3)**, the Python Shell would output **"<class 'int'>"**. Try out the type command to see what different expres-

sions and different values have as their type. Try `type(7)`, `type(7.52)`, and `type("Hello")`.

The next expression for us to consider in our current example is `(7 + 3) * 2`. We can see from this expression that expressions can consist of other expressions. We already know that `7 + 3` is an expression whose value is `10` and whose type is `int`. Thus, we can determine the following about this expression.

Expression	Value	Type
`(7+3)*2`	`20`	int

From this, we can determine that the variable `carrots` will have as its value `20` and its type will be `int`.

There is another number type named `float`. Floats are used to represent numbers that have a fractional part. The value `5.2` is an example of a float. Float values can be expressed in scientific notation as well. The value `3e2` is equivalent to 3×10^2, which is 300, for example. The value after the `e` is the power of ten to which we multiply the first number.

Any time we type a specific value like `5` or `5.25` in a program, we call that value a *literal*. That is, `5.25` is a float literal because it is "literally" the value `5.25`. It's important to have the word "literal" in your programmer vocabulary.

Now let us consider a different example, shown in Listing 1.5.

Listing 1.5

```
1 firstname = "Kanye"
2 lastname = "West"
3 fullname = firstname + " " + lastname
```

These three statements all involve text values. Text values have a special type called *string*. In Python, a string is called a `str` for short. If, in the Python Shell, I were to enter `type("Kanye")`, I would see the output `<class 'str'>`. If I entered the first statement `firstname = "Kanye"` and then afterwards entered `type(firstname)`, I would see the output `<class 'str'>`.

There are four expressions in these three statements. They are shown in the following table.

Expression	Value	Type
`"Kanye"`	`"Kanye"`	`str`
`"West"`	`"West"`	`str`
`firstname + " "`	`"Kanye "`	`str`
`firstname + " " + lastname`	`"Kanye West"`	`str`

Some operators can work on strings, too. When we use the plus (+) on two string values, it "smashes" the two strings together to form a new string. Here, we are taking the first name and putting a space on the end of it. Then, we are appending the last name onto that new string that consists of the first name and trailing space. There is a geeky name for "smashing" two strings together to make a new string, and that name is *concatenation*. We would say that the plus (+) *concatenates* two strings.

Remember, an expression is a piece of code that produces a value. Every value has a type. Every variable has a value and a type.

When we write a string literal, we always put quotes around it. The quotes are not part of the string value, however. In other words, if I write `"abc"`, I know `"abc"` is a string literal, and I know the quotes are not a part of the string's value. Also, we can use either double quotes (") or single quotes (') as long as they match one another. That is, we can write `"cheese"` or `'cheese'` but not `'cheese"`.

Types and values are tremendously important in Python, and this is illustrated by our problematic code from earlier in this section. Recall Listing 1.4, which is shown again below:

```
1 cats = input("How many cats do you have? ")
2 cats = cats + 4
3 print("In a month's time, you will have", cats, "cats.")
```

When we ran this code, we got an error on line 2 that told us "Can't convert 'int' object to str implicitly." When line 2 tries to do the plus operation, it gets confused because `4` is an `int` and it thinks that `cats` is a `str` rather than an `int`. Think about it. Can you guess why? Look back at line 2. Look back at line 1.

The **input** command in line 1 retrieves characters entered from the keyboard. These characters could be letters, symbols, and/or numbers. Since the value placed into **cats** by **input** could be any of these things, the type of the value **input** gives us is **str**. Suppose the user typed a **5** at the prompt "How many cats do you have?" The initial value of the variable cats will be the **str** value **"5"** rather than the **int** value **5**.

It doesn't make sense to Python to add a string and an integer. After all, what is reasonable to assume about the type and value of an expression like **"Hello" + 32**? In the above example, we need to convert **cats** from a string to an integer so that we can treat **cats** as an integer. Let us add a new line of code after the **input** statement in Listing 1.6 at line 2.

Listing 1.6

```
1 cats = input("How many cats do you have? ")
2 cats = int(cats)
3 cats = cats + 4
4 print("In a month's time, you will have", cats, "cats.")
```

The **int** command converts the string value stored in **cats** to an integer, and then it overwrites the value of **cats** to this new **int** value. Converting a value from one type to another is called *casting*.

When we write code, it is bound to have errors we need to correct. Sometimes, those errors make the program crash and we see an actual error message in red text on the screen. Other times, however, we don't get a nice error message. Instead, the program appears to behave erroneously. Erroneous code is called a *bug*, and it is up to us to "debug" the program.

At some point in this book's sure-to-be-glorious future, we'll have a nice detour in this box about the origin of the term "debug." It's an amusing historical tale. For now, read this: http://www.wired.com/2013/12/googles-doodle-honors-grace-hopper-and-entomology/.

Let's practice detecting and fixing bugs. We'll again use Listing 1.4 as our starting point, only this time instead of adding **4** to the number of cats, we will double the number of cats by multiplying by **2**. After all, when it comes to feral barn cats, this is a more accurate representation of what happens to cat

Table 1.4: Arithmetic operators

Operator	Usage
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`//`	Integer division - Returns the whole number result only from division. Example: `5 // 2` gives us `2` rather than `2.5`.
`%`	Modulus, or mod - Returns the remainder from division. Example: `5 % 2` gives us `1` since `5` divided by `2` is `2` remainder `1`
`**`	Exponentiation - Returns the result of raising a number to a power. Example: `2 ** 3` gives us `8`

populations. Consider our new Listing 1.7.

Listing 1.7

```
1 cats = input("How many cats do you have? ")
2 cats = cats * 2
3 print("In a month's time, you will have", cats, "cats.")
```

Note that we forgot to cast `cats` to be an `int`. We might expect this program to have a `TypeError` since it doesn't make sense to multiply a string and an integer. In fact, this is not what happens. Type `3` for the number of cats. What happens?

`33` cats?! Good grief! As it turns out, Python allows string values to be repeated using the asterisk/star (`*`) operator. Instead of performing `3 * 2`, the expression we've inadvertently performed is `"3" * 2`, which is the same as `"3" + "3"`, which is the same as `"33"`. If we properly cast `cats` to an `int` before multiplying, we get the proper result and we have "debugged" the program.

It is important to understand what operators are available to us. Tables 1.4 and 1.5 provide a more comprehensive list of these operators and their function.

Table 1.5: String operators

Operator	Usage
`+`	Concatenation `Example:` `"ab" + "cd"` gives us `"abcd"`
`*`	Repetition `Example:` `"-" * 5` gives us `"-----"`

`%`	Formatting
	The string format operator allows us to insert one string value into the middle of another string, v
	`Example 1:`
	Suppose we have a variable `forks` that contains the integer value `5`.
	`"There are %d forks on the table." % forks`
	would produce the string `"There are 5 forks on the table."`
	`Example 2:`
	Suppose we have two variables `first` and `last` that contain string values `"Bob"` and `"Barker`
	`"Hi, %s %s." % (first, last)`
	would produce the string `"Hi, Bob Barker."` Note that if we have multiple values to be inser
	`Example 3:`
	Suppose we have a variable `ch` that contains a single character `'!'`
	`"OMG%c%c%c" % (ch, ch, ch)`
	would produce the string `"OMG!!!"`.

Table 1.6: String format specifiers

Specifier	Description
`%d`	the value to be inserted is an integer
`%5d`	the value is aligned to the right across a 5-character column
`%-5d`	the value is aligned to the left across a 5-character column
`%f`	the value to be inserted is a float
`%10.2f`	the value to be inserted is a float aligned to the right across a 10-character column, and we should
`%s`	the value to be inserted is a string
`%10s`	the value is aligned to the right across a 10-character column
`%-10s`	the value is aligned to the left across a 10-character column
`%c`	the value to be inserted is a single character

1.6 Calling functions

In Section 1.4, we identified the three types of statements that we knew at that point.

1. **`print`** statements
2. **`input`** statements
3. *assignment* statements

We have since learned about other statements. For example, we can use **`type`** to determine the type of an expression or **`int`** to cast a **`str`** to an **`int`**. We have also been using the word *command* to refer to words like **`print`**, **`input`**, and **`type`** that seem to have important meaning to Python. In fact, these commands are actually called *functions*. It's important to learn to speak like a programmer when you are writing code, so we will call them functions from here forward.

You'll note that in this book, we try to steer clear of technical terms until we reach an appropriate time to introduce them. This seems to be a better approach than throwing every possible technical term at you right away and then expect you to memorize them without any context whatsoever.

So, let's practice speaking like programmers.

Instead of…	Programmers would say…
use a function	*call* a function
function *produces* a value	function *returns* a value

Consider the following code.

```
answer = input("Do you wish to continue (y/n)? ")
```

Programmers would say they are *calling* the input function, and the input function will *return* a string value.

All functions return a value, even something like **print**. Just for fun (wheee!), type the following into the Python Shell window.

```
var = print("Hello.")
```

Now, type **var** in the Shell window and press ENTER. Hmm, normally when we type the name of a variable or we type an expression into the Python Shell window, it tells us its value. We get nothing. Type the expression **type(var)** into the Shell. Aha! The variable **var** has a special type called **NoneType**. Every function call is an expression that returns some value, even if at least that value belongs to **NoneType**.

Okay, it makes sense to have **int**, **float**, and **str** because it's easy to think of whole and fractional numbers and text values, but why does **NoneType** exist? We're really not ready for the answer yet, but rest assured we'll cover it eventually. The short answer is that seasoned Python programmers can use **NoneType** to make their code really easy to read in some circumstances. Stay tuned.

We can now simplify the list of statements we know about to these.

1. function call statements
2. assignment statements

From now on, we will refer to things like **print**, **input**, and **type** as functions rather than as commands. Note that assignment statements can have function calls in them, like

```
name = input("What is your name? ")
```

This statement is an assignment statement, and the RHS is a function call.

The expressions that are placed between the parentheses after the function's name are called *arguments*. Some functions take no arguments, some functions take one argument, and some other functions can take several arguments. Arguments are separated with commas. Arguments tell the function how to do its

job. The code in Listing 1.8 shows different examples of how to call functions with differing numbers of arguments.

Listing 1.8

```
1 age = input("What is your age? ")
2 age = int(age)
3 age = age + 1
4 print("In one year, you will be", age, "years old.")
```

Lines 1 and 2 demonstrate calling a function with one argument. Line 4 has three arguments. The first is a string value, the second is a string variable, and the third is another string value.

Because functions return values, we can call one function and immediately give its return value to another function. Look at lines 1 and 2 in Listing 1.8 again. Since casting can be performed on any expression, we could do both the **input** and the **int** cast on one line, like this.

```
age = int(input("What is your age? "))
```

The **input** function is called first, and the result returned by **input** is then given to **int**, which casts the result from a string to an integer. Students are often puzzled by what appears to be "double right parentheses" at the end of the above statement. Note that the last right parenthesis matches the left parenthesis for **int**, and the next to last right parenthesis matches the left parenthesis for **input**. This is shown visually in Figure 1.4.

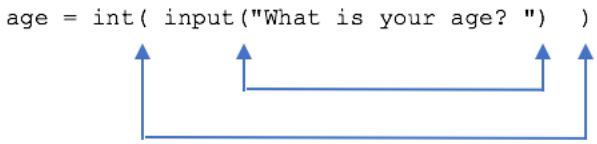


Figure 1.4: Use of arrows to showing matching parentheses

If the user were to type **18**, the code above would be executed and "transformed" through the steps shown in Figure 1.5.

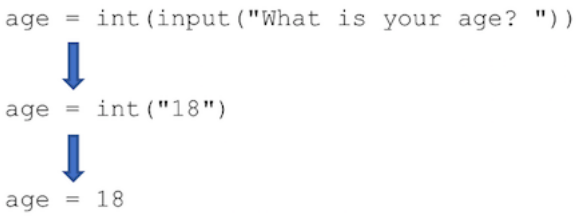


Figure 1.5: Statement execution

Just for fun (again: whee!!), let's change line 4 of Listing 1.8 and try out the string formatting operator (**%**). Instead of this,

```
print("In one year, you will be", age, "years old.")
```

We could do this:

```
print("In one year, you will be %d years old." % age)
```

Here, the value of **age** gets inserted into the format string in place of the **%d**. There is no advantage to one way or the other, per se. There will be lots of ways to write code, though you should try to write code so that is *readable*. If you write code that is easy to read, it will be easier to change. Some programmers like the second way because it is easy to see the format of what the output will be, and it can be easier to control where the spaces go in the output.

There is another way to do string formatting in Python that is newer and is now preferred as of Python 3.5. We will introduce it later in the book. However, we show this method for string formatting since this is an introductory computer science textbook, and this "style" of string formatting is one you'd encounter in other programming languages (e.g., C, Java, etc.).

To review, we started with four lines of code.

```
age = input("What is your age? ")
age = int(age)
age = age + 1
print("In one year, you will be", age, "years old.")
```

Then, we "tweaked" the code so that, ultimately, it looked like this.

```
age = int(input("What is your age? "))
age = age + 1
print("In one year, you will be %d years old." % age)
```

As you program, you will develop your own coding style. You will want to decide which of the two blocks of code (or a combination of them) looks the most readable to you.

1.7 Handy functions

The purpose of this book is to teach novice programmers how to program. The purpose is not to have you learn every single thing about the Python programming language. To that point, this book is not intended to be a desk reference for all things Python. If you want to find information about a particular language feature or a list of available functions, the best way to look is to use a Web search (like Google) or go straight to the Python Documentation online.

That said, there are a number of handy functions that come "pre-packaged" with Python and ready for you to use. We will list a few of them here in the subsections that follow, since we're highly confident you'll use them very soon. We'll introduce more functions throughout the book. Eventually, you'll learn how to create your own functions. That will happen in Chapter 4. Creating your own functions is pretty cool.

Some functions can be called just by stating their name. For example, we can use the **`print`** function just by typing something like **`print("Hello")`**. Other functions are part of what we call *libraries*. One example is the **`math`** library. In order to use functions in the **`math`** library, we must do two things. First, we must type the statement **`import math`**. Then, **`math`** library functions

start with the prefix **math.** (read aloud as "math-dot"), so when we call them we must use the **math.** prefix. Here is an example of using a function from a library (see Listing 1.9).

Listing 1.9

```
1 import math
2
3 number = float(input("Enter a number: "))
4 squared = math.pow(number, 2)
5 print("Your number squared is %f." % squared)
```

In Listing 1.9 line 4, the function named **pow** is contained in the library **math**. Because of this, we must type **math.pow** for the function name in order to call it. As you may be able to guess, **pow** raises a number to a power. In this case, we are raising whatever number the user types to the second power, which is called squaring the number.

1.7.1 Math functions

Suppose **i** is an **int** variable and **f** and **g** are both **float** variables. Said another way, **type(i) == int**, **type(f) == float**, and **type(g) == float**. Some of the functions in Table 1.8 belong to the **math** library and some do not. If we want to use these functions, we need to first type the statement **import math**. If we don't write **import math**, we will get a **NameError** that tells us **math** is not defined.

Here is how to read Table 1.8. If the function is shown as **i = math.ceil(f)**, that means the function expects us to pass it a **float** (hence the **f** in parentheses), and the function will return an **int** (hence the **i** on the LHS of the equals).

There are many more functions found in **math**, but these are the ones you're most likely to use in the near future. Again, consult the online Python Documentation if there's something specific you're looking for that's not mentioned in this subsection.

Let's consider an example of how we might use some of these functions.

Suppose we want to buy a whole bunch of fidget spinners in bulk and then re-sell them to make a profit. If we buy them in bulk, then we can get a good

Table 1.8: Math functions

Function	Returns

`i = round(f)`	The integer resulting from rounding `f`.
	Examples:
	`round(5.2) == 5`
	`round(5.78) == 6`
	`round(6) == 6`
	`round(-1.2) == -1`

`i = math.ceil(f)`	The smallest integer $\geq$ `f`.
	Examples:
	`math.ceil(5.2) == 6`
	`math.ceil(5.78) == 6`
	`math.ceil(6) == 6`
	`math.ceil(-1.2) == -1`

`i = math.floor(f)`	The largest integer $\leq$ `f`.
	Examples:
	`math.floor(5.2) == 5`
	`math.floor(5.78) == 5`
	`math.floor(6) == 6`
	`math.floor(-1.2) == -2`

`g = abs(f)`	The absolute value of `f`.
	Examples:
	`abs(23.2) == 23.2`
	`abs(-23.2) == 23.2`

deal because they'll be cheaper per fidget spinner. Let's have users enter the number of fidget spinners they want and the number of spinners that come in a case. The program should tell them how many cases they'll need to buy to get at least that many number of fidget spinners.

To write this program, we'll need to ask for two inputs: the number of spinners and the number of spinners per case. Then, we'll need to calculate the number of cases needed and then output that number. Listing 1.10 shows how to do this in Python code.

Listing 1.10

```
1 spinners = int(input("How many spinners do you need? "))
2 spinners_per_case = int(input("How many spinners come in a case? "))
3
4 cases = math.ceil(spinners / spinners_per_case)
5
6 print("You need to order %d cases." % cases)
```

In line 4 of Listing 1.10, we divide **spinners** by **spinners_per_case** to get how many cases we'll need. But this gives us a fractional number potentially. For example, if we wanted **18** spinners and **12** come in a case, that would be **1.5** cases, but we can't order one case and then another half of a case. We actually need **2** cases. This is where **math.ceil** comes in. We take the "fractional" number of cases needed and find the *ceiling* of it. This makes **cases** an integer that is greater than or equal to the number of fractional cases.

1.7.2 Random functions

The library **random** has functions that help us generate random numbers. This is useful for writing programs that involve random chance, for example, rolling dice, flipping coins, etc. There are several useful functions in **random**, but the two we'll focus on right now are **random** and **randint**.

Suppose **i**, **start**, and **end** are **int** variables and **f** is a **float** variable.

Listing 1.11 shows an example of how one might use a **random** library function.

Table 1.9: Math functions

Function	Returns
`f = random.random()`	A float value between `0` and `1` inclusive.
`i = random.randint(start, end)`	An integer value between `start` and `end` inclusive.

Listing 1.11

```
1 import random
2
3 print("Rolling a six-sided die......")
4 die_roll = random.randint(1, 6)
5 print("You rolled a %d." % die_roll)
```

The code in Listing 1.11 may produce a different die roll every time you run the program.

1.8 Comments

As we go forward in learning Python, our programs will get longer and more intricate. It may be helpful to annotate our code with short comments to remind us what our code does. Listing 1.12 shows a (somewhat silly) example.

Listing 1.12

```
1 # Get the user's age.
2 age = input("What is your age? ")
3 age = int(age)
4
5 # Tell the user his or her age one year from now.
6 age = age + 1
7 print("In one year, you will be", age, "years old.")
```

The lines that start with a `#` symbol are called comments. Any line that starts with `#` will be ignored by the Python interpreter. Those lines are only for the programmer to read. Again, this is a somewhat silly example because our code is relatively simple and probably does not require comments.

Programmers will also use comments at the beginning of a code file to document what the program does. Here is an example (see Listing 1.13).

Listing 1.13: A header comment at the top of a code file

```
# Program: tictactoe.py
# Programmer: Susan McConnell
# Description:
#   This program allows users to play Tic Tac Toe against
#   a computer opponent.  Users choose the row and column
#   to place their 'X' or 'O' on the game grid in each turn.
```

Because placing a # symbol at the start of a line hides the code from Python, another use of comments is to hide old code. Sometimes, we want to save old code without deleting it. This can occur when we're not sure if new code we're trying out is going to work, and so we may not want to lose our old code in case we need to go back to it later. Listing 1.14 demonstrates this concept.

Listing 1.14: Commenting out code

```
1 #age = input("What is your age? ")
2 #age = int(age)
3
4 age = int(input("What is your age? "))
```

Python will not execute lines 1 and 2 because they are "commented out." It will, however, execute line 4.

Another use for comments is to help us program. As human beings, we do not naturally think in code. Even experienced programmers struggle to think purely in terms of programming language code. One good way to program is to write comments first in plain English to help us organize our logic and thoughts, and then we write Python code beneath the comments. For example, we might start with:

```
# Get the user's age.

# Print how old they'll be in a year.
```

Then, we can fill in the details.

```
# Get the user's age.
age = input("What is your age? ")
age = int(age)

# Print how old they'll be in a year.
age = age + 1
print("In one year, you will be", age, "years old.")
```

Comments end up being very important later on in the book when we start creating our own functions (yes, we get to make our own functions eventually). Practice writing comments when you write your own code.

1.9 Exercises

1. Write a program that creates as its output a face on the screen by arranging different symbols, letters, and/or numbers. Here is an example.

```
\\///
 0 0
  v
 ---
```

2. What is the output of the following program?

```
a = 1
b = a * 2
c = 2 * b + 1
print(a)
print(b)
print(c)
```

3. What is the output of the following program?

```
a = 2
b = a * 2
c = b ** a
a = c % 2
print(a)
print(b)
print(c)
```

4. What is the output of the following program?

```
a = 5
b = a // 2
c = a / 2
a = a % 2
print(a)
print(b)
print(c)
```

5. What is the output of the following program if the user enters a **2** at the first prompt and a **3** at the second prompt?

```
x = input("Enter a whole number: ")
y = input("Enter another whole number: ")
z = x + y
print(z)
```

   Be careful. Try to type this program into the Python Shell and see what you get.

6. Write a program that asks users for two whole numbers. It should then add them and print the result. Here is an example of what the output should look like.

```
Enter a whole number: 2
Enter another whole number: 3

2 + 3 = 5
```

7. What is the output of the following program?

```
s = "John"
t = "Smith"
r = s + t
print(r)
```

8. What is the output of the following program?

```
s = "John"
t = "Smith"
print("%s, %s" % (t, s))
```

9. What is the output of the following program?

```
s = "John"
t = "Smith"
r = "%s, %s" % (t, s)
print(r)
```

10. What is the output of the following program?

```
s = "John"
t = "Smith"
r = "%s, %s"
print(r % (t, s))
```

11. What is the output of the following program? Write down the answer exactly how it would appear on-screen.

```
print("%s %s" % ("Item", "Price"))
print("%s %f" % ("Soda", 1.75))
print("%s %f" % ("Pizza", 2.00))
print("%s %f" % ("Hot Dog", 1.50))
print("%s %f" % ("Crab Legs", 30.99))
```

12. What is the output of the following program? Write down the answer exactly how it would appear on-screen.

```
print("%-10s %10s" % ("Item", "Price"))
print("%-10s %10.2f" % ("Soda", 1.75))
print("%-10s %10.2f" % ("Pizza", 2.00))
print("%-10s %10.2f" % ("Hot Dog", 1.50))
print("%-10s %10.2f" % ("Crab Legs", 30.99))
```

13. What is the output of the following program? Write down the answer exactly how it would appear on-screen.

```
print("%-10s %10s" % ("Item", "Price"))
fmt = "%-10s %10.2f"
print(fmt % ("Soda", 1.75))
print(fmt % ("Pizza", 2.00))
print(fmt % ("Hot Dog", 1.50))
print(fmt % ("Crab Legs", 30.99))
```

14. What is the output of the following program? Write down the answer exactly how it would appear on-screen.

```
print("%-10s %10s" % ("Item", "Price"))
fmt = "%-10s %10.2f"
print(fmt % ("Soda", 1.75))
print(fmt % ("Pizza", 2.00))
#print(fmt % ("Hot Dog", 1.50))
#print(fmt % ("Crab Legs", 30.99))
print(fmt % ("Pretzel", 1.50))
print(fmt % ("Nachos", 2.25))
```

15. Write a program that asks the user for a single character. The program should then greet them in block letters "HI" consisting solely of that character. Here is an example of a running program.

```
Enter a character: #

##  ##  #####
######    #
##  ##  #####
```

Here is another example of a running program if the user were to type a different character.

```
Enter a character: +

++  ++  +++++
++++++    +
++  ++  +++++
```

Chapter 2

Conditional Statements

2.1 The `if` statement and the Boolean type

Do you remember our sample program involving cats in Chapter 1? Let's re-write it here (Listing 2.1).

Listing 2.1

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
3 print("In a six-months, you will have %d cats!" % cats)
```

Feral barn cats can multiply rapidly if not spayed or neutered, so this is not that ridiculous of an example. Suppose we want our program to make the observation "That's a lot of cats!" if the barn owner ends up with more than 20 cats. Our first attempt at this, based on what we know so far, might look like this.

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
3 print("In a six-months, you will have %d cats!" % cats)
4 print("That's a lot of cats!")
```

Python programs always execute in sequence. That is, first Python executes line 1, then line 2, line 3, and finally line 4. With that in mind, line 4 will always

execute no matter what. This is absurd given what we are trying to accomplish, and even more absurd if the user types 0 for the number of cats initially.

What we need is a way to say "only print if the number of cats is 20 or more." Python allows us to do this by using something called an **if** statement. Let's modify our code as follows (Listing 2.2).

Listing 2.2

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
3 print("In a six-months, you will have %d cats!" % cats)
4 if cats > 19:
5     print("That's a lot of cats!")
```

Notice how we've added a line just before our final print statement. Note also how our print statement in line 5 is now tabbed over one tab-stop. Now try out your program using several different inputs for cats. It works!

To understand more about why it works and how if statements work in general, let's add one more line of code (for now).

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
3 print("In a six-months, you will have %d cats!" % cats)
4 if cats > 19:
5     print("That's a lot of cats!")
6 print("Thank you for using our program.")
```

Try out your program using several different inputs for cats again. Notice how no matter how many cats you enter, the program always prints "Thank you for using our program." Now, let's tab the last line over one tab-stop so that it lines up with the previous print statement.

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
3 print("In a six-months, you will have %d cats!" % cats)
4 if cats > 19:
5     print("That's a lot of cats!")
6     print("Thank you for using our program.")
```

Try out your program again.

Notice how now the program only prints "Thank you for using our program." if the value of **cats** is greater than **19**. This tells us something about how **if** statements work. When the expression **cats > 19** is true, any statements that are indented beneath the **if** statement are executed, but they are only executed if the expression **cats > 19** is true. A sequence of statements separated together by indentation (i.e., tabs) is called a *block* of statements.

Let's try one more thing to understand how tabs/indentation work. What happens if we de-indent line 5, i.e., the first line under the **if**?

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
3 print("In a six-months, you will have %d cats!" % cats)
4 if cats > 19:
5 print("That's a lot of cats!")
6     print("Thank you for using our program.")
```

Run your code. Kablooey! The error indicates that whenever Python sees an **if**, there should be at least one indented section of code beneath it.

Earlier, we referred to **cats > 19** as an expression. If it is an expression, then it must produce a value and must have a type, right? Let's go to our Python Shell window and type the following: **type(cats > 19)**. The result we are given is something called a **bool**. The word "bool" stands for Boolean. The word Boolean refers to George Boole, an English mathematician who formalized mathematical logic. Valid Boolean values are either **True** or **False**.

Whenever the value of the Boolean expression given to an **if** statement is **True**, the indented block of code beneath it will be executed. If it is **False**, the block will be skipped.

Based on what we know about variables having types, we might conclude that we can create Boolean variables, too, and we'd be right! Consider the code in Listing 2.3.

Listing 2.3

```
1 cats = int(input("How many barn cats do you own? "))
2 cats = cats * 2
```

```
print("In a six-months, you will have %d cats!" % cats)

too_many = cats > 19
if too_many:
    print("That's a lot of cats!")

print("Thank you for using our program.")
```

Now, suppose we want to say "You should get more cats" if the barn owner does not yet have 20 cats. That is, we want to say this message if the Boolean expression is **`False`**. We can adjust our code by changing this:

```
if cats > 19:
    print("That's a lot of cats!")
```

to this

```
if cats > 19:
    print("That's a lot of cats!")
else:
    print("You should get more cats.")
```

Any **`if`** block can have an **`else`** block to handle situations where the **`if`** condition is **`False`**, but having an **`else`** block is optional.

Boolean expressions can be formed using *logical operators*. We've already seen greater-than (**`>`**). Table 2.1 shows the rest of the logical operators in Python. Note that, for example, **`>=`** denotes greater-than-or-equal-to, presumably because the symbol $\geq$ does not exist on your computer keyboard. Also, note that comparing two values for equality uses the **`==`** double-equals symbol. We already use the equals (**`=`**) symbol for assignment statements so **`==`** is used to compare two values as part of a Boolean expression.

2.2 `elif` and Boolean Operators

Different states in the U.S. have different rules for when one is allowed to obtain a license to drive an automobile. One common practice is to allow individuals who are 16 or older to get a license. Suppose we wished to write a Python

Table 2.1: Logical operators

Operator	Description
==	equals
!=	not equals
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

program that tells people if they are eligible to obtain a driver's license in their state. We might prompt users for their respective ages, and then we could respond appropriately. Let's use the code in Listing 2.4 as our starting point.

Listing 2.4

```
age = int(input("What is your age? "))
if age >= 16:
    print("You are old enough to take your license exam.")
else:
    print("Perhaps you should ride a bike for now.")
```

The situation in some states is more involved than this. In fact, let us suppose that if users are 14 or older, they can obtain a learner's permit where they are able to drive if accompanied by an adult. If they are 16 and they also have had a driver's education class, then they can take the license exam. Got it? Let's ignore driver's education for right now to simplify things, and let's try to re-write our program using what we know so far.

```
age = int(input("What is your age? "))
if age >= 14:
    print("You may obtain a learner's permit.")
else:
    if age >= 16:
        print("You are old enough to take your license exam.")
    else:
        print("Perhaps you should ride a bike for now.")
```

That's a lot of indentation! Fortunately, the indentation makes the logic of our code a bit easier to follow, which is a nice feature of Python code. When we reach the first **else**, our code beneath the **else** starts at a new indentation level.

Try running this code using a variety of inputs. Do you notice a problem?

It appears that no matter what age we enter, we can never get it to tell us we're old enough to take the license exam. Our Boolean conditions are not in the correct order. Even if we type that we are 17 years old, for example, the first condition is **True**, so it prints "You may obtain a learner's permit." Let's fix the order of our code to place the most restrictive Boolean expression first.

```
age = int(input("What is your age? "))
if age >= 16:
    print("You are old enough to take your license exam. ")
else:
    if age >= 14:
        print("You may obtain a learner's permit.")
    else:
        print("Perhaps you should ride a bike for now.")
```

This is much better.

You might imagine that if you have a lot of different conditions you wish to handle in your program, your code many eventually look like a very long series of stair steps. This would make our code a bit difficult to read and maintain, so Python provides us with a statement for making a series of else-if conditions. This statement is called **elif**, for short. We can re-write the code above as follows.

```
age = int(input("What is your age? "))
if age >= 16:
    print("You are old enough to take your license exam. ")
elif age >= 14:
    print("You may obtain a learner's permit.")
else:
    print("Perhaps you should ride a bike for now.")
print("Done.")
```

Note that we've also added a statement that prints "Done" for illustrative purposes.

Having **elif**'s work the same way as having an **else** block followed by an **if** block. If the first condition is **True**, we print that we are old enough to take the exam. Then, none of the other conditions are checked. Instead, Python drops down to the "Done" **print** statement. If the first condition is **False**, then, and only then, is the next condition checked. If that condition is **True**, we print that they can obtain a learner's permit and we drop down to the "Done" **print** statement. If that condition was **False**, our code descends to the **else** block.

else blocks are always optional, as are **elif** blocks. You can have any number of **elif** blocks, but **if**/**elif**/**else** blocks must always come in that order: first an **if** statement, then any **elif**'s, and finally an **else**.

Now let's consider driver's education in our example. Individuals must be 16 years old and have had a driver's education class to take the license exam. So, we'll need to ask them if they've taken the exam by using another **input** statement.

Listing 2.5

```
age = int(input("What is your age? "))
drivers_ed = input("Have you passed driver's education? (y/n) ")
if age >= 16:
    print("You are old enough to take your license exam. ")
elif age >= 14:
    print("You may obtain a learner's permit.")
else:
    print("Perhaps you should ride a bike for now.")
print("Done.")
```

Notice how we are prompting users to enter either **"y"** or **"n"** for their answer to the driver's education question. Also notice that we do not need to cast the value returned from the **input** function to another type like we did in the previous **input** statement (the one that asks for the age). The variable **drivers_ed** can remain a string, because **"y"** or **"n"** are string values.

Now, let's adjust our logic. We'll only show the part of the code that deals with taking the license exam for now. This code:

```
if age >= 16:
    print("You are old enough to take your license exam. ")
```

can become this code:

```
if age >= 16:
    if drivers_ed == "y":
        print("You are old enough to take your license exam. ")
```

This new code says that in order to print **"You are old enough to take your license exam"** first **age >= 16** must be **True**, and then **drivers_ed == "y"** must be **True**, too.

There is another way to do this that you might like better. We can use something called a *Boolean operator*. Instead of this:

```
if age >= 16:
    if drivers_ed == "y":
        print("You are old enough to take your license exam. ")
```

we can say this:

```
if age >= 16 and drivers_ed == "y":
    print("You are old enough to take your license exam. ")
```

The **and** keyword is the Boolean operator. We can now modify Listing 2.5 as follows in Listing 2.6.

Listing 2.6

```
1 age = int(input("What is your age? "))
2 drivers_ed = input("Have you passed driver's education? (y/n) ")
3 if age >= 16 and drivers_ed == "y":
4     print("You are old enough to take your license exam. ")
5 elif age >= 14:
6     print("You may obtain a learner's permit.")
7 else:
8     print("Perhaps you should ride a bike for now.")
9 print("Done.")
```

Table 2.2: Logical operators

Operator	Description
`and`	Suppose `a` and `b` are Boolean expressions. If `a == True` and `b == True`, then `a and b == True`. If `a == True` and `b == False`, then `a and b == False`. If `a == False` and `b == True`, then `a and b == False`. If `a == False` and `b == False`, then `a and b == False`. In other words, the expression `a and b` is only `True` if both `a` and `b` are individually `True`.
`or`	Suppose `a` and `b` are Boolean expressions. If `a == True` and `b == True`, then `a or b == True`. If `a == True` and `b == False`, then `a or b == True`. If `a == False` and `b == True`, then `a or b == True`. If `a == False` and `b == False`, then `a or b == False`. In other words, the expression `a or b` is `True` whenever either `a` or `b` is individually `True`, or b
`not`	Suppose `a` is a Boolean expression. If `a == True`, then `not a` is `False`. If `a == False`, then `not a` is `True`.

When we join two separate Boolean expressions together with the keyword `and`, the resulting expression is `True` only if both of the expressions are each `True`. If either `age >= 16` is `False` or `drivers_ed == "y"` is `False`, then the whole Boolean expression `age >= 16` and `drivers_ed == "y"` is `False`.

There are three Boolean operators in Python, which are given in Table 2.2.

Let us look at another example where these Boolean operators become useful. Suppose we want to write a program that tells people if they have an increased risk for heart disease. There are a number of known risk factors, but for now we'll only consider three. Without Boolean operators, we might write the program in Listing 2.7.

Listing 2.7

```
bp = input("Do you have high blood pressure (y/n)? ")
smoke = input("Do you smoke (y/n)? ")
hist = input("Do you have a family history of heart disease (y/n)? ")

if bp == "y":
    print("You have an increased risk of heart disease.")
elif smoke == "y":
    print("You have an increased risk of heart disease.")
elif hist == "y":
    print("You have an increased risk of heart disease.")
else:
    print("You do not have an increased risk for heart disease.")
```

Note that each of the statements that follow the **if**/**elif** conditions all print the same message. Instead, we can use the Boolean operator **or** to simplify the code resulting in Listing 2.8.

Listing 2.8

```
bp = input("Do you have high blood pressure (y/n)? ")
smoke = input("Do you smoke (y/n)? ")
hist = input("Do you have a family history of heart disease (y/n)? ")

if bp == "y" or smoke == "y" or hist == "y":
    print("You have an increased risk of heart disease.")
else:
    print("You do not have an increased risk for heart disease.")
```

Lastly, we can use the **not** operator to improve the readability of code. Recall how, in our driver's license example (Listing 2.6), we asked if the user had taken a driver's education course. With this scenario in mind, consider the following (Listing 2.9).

Listing 2.9

```
age = int(input("What is your age? "))
drivers_ed = input("Have you taken driver's ed (y/n)? ") == "y"
if 14 <= age and age <= 16 and not drivers_ed:
    print("You should consider taking a driver's ed course.")
```

This example includes a few interesting code constructions. Line 2 of Listing 2.9 is different from what you've seen before. Previously in Listing 2.6, we let `drivers_ed` be a string variable whose value should be `"y"` or `"n"`. In this code, `drivers_ed` is a Boolean variable. Why?

Notice that the `input` part of the statement is followed by a logical operator, the double-equals (`==`). Thus, we are comparing what the user types to the string `"y"`. If the user typed `"y"`, then the RHS of the assignment statement is `True`, so `drivers_ed` gets the value `True`. If the user types anything else, `drivers_ed` becomes `False`.

Next, we have code that checks to see if the user age's is still typically in the range that students take a driver's education class. If so, and they've not had driver's education, they are advised to do so via a `print` statement. The only way the `print` statement will happen is if the age is between `14` and `16`, and they've not had driver's education.

Keep in mind that the programming practice in line 2 of Listing 2.9 might not be a good idea. Your author has only written code this way to show you that it can be done. The latter part of the expression (`== "y"`) might be hard to notice, and because it is hard to notice, it may make the code more difficult to read and therefore maintain. In the future, we may want to check what the user typed in to make sure it's what we expect (i.e., a `"y"` or a `"n"`). Checking inputs is something we'll discuss in Chapter 3.

2.3 Nuances of Boolean operators

Students will sometimes write code like this.

```
age = int(input("What is your age? "))
if age == 14 or 15:
    print("You can get a learner's permit.")
```

Run this code and type `18`. Uh oh. Why does it print "You can get a learner's permit"?

Remember than anything on the LHS and RHS of a Boolean operator must be itself a Boolean expression. In other words, the LHS and RHS must both be

`True` or **`False`**. In the above example, there is an expression on either side of the double-equals. One is

```
age == 14
```

and the other is

```
15
```

The expression **`age == 14`** is **`False`**. The expression **`15`** is **`True`**. **Anything non-zero in Python is treated as `True`.** Therefore, the Boolean expression is transformed as in Figure 2.1.

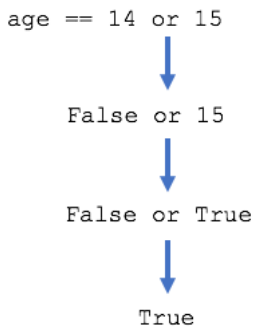


Figure 2.1: Evaluating a Boolean **`or`** expression

Instead of

```
age == 14 or 15
```

a programmer should write

```
age == 14 or age == 15
```

Bottom line: be very careful when using and/or.

2.4 Exercises

1. Suppose `x` is an integer variable. What is the difference between the expression `x = 2` and the expression `x == 2`?

2. Given the following code:

```
x = 3
y = 6
```

   What is the type and value of each of the following expressions?

```
x != 0
y >= 3
x == y
not (x == 0)
x > 1 and y < 5
2 < x and x < 5
7 < y or y < 10
```

3. Identify the error in the following code.

```
num = float(input("Enter a real number: "))
if num < 0.0:
    print("%.2 is negative." % num)
else:
    print("The number is 0.0.")
elif num > 0.0:
    print("%.2 is positive." % num)
```

4. Identify the error in the following code.

```
num = float(input("Enter a real number: "))
if num = 0.0:
    print("The number is 0.0.")
elif num < 0.0:
    print("%.2 is negative." % num)
else:
    print("%.2 is positive." % num)
```

5. What is the output of the following code?

```
major = "Music"

if major == "Computer Science" or "Math":
    print("Calculus is required for your major.")
elif major == "Music":
    print("Music Theory is required for your major.")
```

6. Consider the following code.

```
if a > 2:
    if b < 3:
        print("Statement 1")
    else:
        print("Statement 2")
else:
    if b > 3:
        print("Statement 3")
    else:
        print("Statement 4")
```

What is the output of the program if **a** and **b** were given the following values prior to the start of the code?

```
a = 3
b = 4

a = -2
b = 2
```

7. What is the output of the following code?

```
r = 10
if 5 < r and r < 15:
    print("1")
if 5 < r or r < 15:
    print("2")

r = 100
if 5 < r and r < 15:
    print("3")
if 5 < r or r < 15:
    print("4")
```

8. Write code that rounds a float number up or down *without* using the **round** function introduced in Section 1.7. That is, write code that asks for a number, and then it takes that number and rounds it up or down appropriately without using **round**. For example, if the user types **3.6**, the program should output a **4** since the fractional part **.6** is greater than **.5**. The user were to type **3.2**, the program should output a **3** since the fractional part **.2** is less than **.5**.

   It may help to know that if you use the **int** function to cast a float to an integer, the fractional part disappears. This is called *truncation*. For example, in the following code

```
f = 3.6
i = int(f)
```

   the value of **i** ends up being **3**.

Chapter 3

Loops and String Manipulation

3.1 `while` loops

Most of the examples of code we've seen so far have been fairly short – a few lines here and there. Now we're going to start looking at programs that are a bit longer. Don't be too daunted by this. If you practice, you'll gain comfort with knowing how to break down programs section by section. Let's go ahead and practice doing this starting with this chapter.

Sometimes we'll want to repeat an action in code. One example might be that we repeatedly ask the user for input until we run out of that input. Suppose we write a program that calculates the amount of sales tax on different items at a store. We might write the code found in Listing 3.1.

Listing 3.1: Code to compute total cost on items including sales tax

```
1 tax = 0.07   # that is, 7 percent
2 print("Enter the amount of each item.")
3 print("Enter 'quit' to end the program.")
4 amount = input("Amount of item? ")
5 while amount != "quit":
6     amount = float(amount)
7     total = (amount * tax) + amount
8     print("The total for that item will be %.2f." % total)
```

```
9       amount = input("Amount of item? ")
10  print("Thank you for using this program.")
```

To some readers, this program might look intimidating at first, but each section contains ideas we've seen before. We've just added one idea after the next. So, what are these familiar code sections? Well, we are getting input from the user. If the input is not the word "quit," we cast the amount to a float (line 6) and compute the total price of the item with tax included (line 7).

What is special about the above code is the keyword **`while`** in line 5. **`while`** works similar to **`if`** in many ways. When we use **`if`**, we specify a Boolean condition immediately after the word **`if`**, and if the condition is **`True`**, Python executes the code that is indented beneath it. **`while`** works the same way, however, once we reach the end of the indented block, the code "loops" back up to the condition again. If the condition is again **`True`**, we reenter the indented block of code. Thus, we can make code repeat itself as long as a condition is met. A programming language "construct" that allows code to be repeated is known as a *loop*.

We call **`amount != "quit"`** in Listing 3.1 the *loop condition*. We call the indented block of statements that follows the loop condition the *loop body*. We say that the statements in the indented block are *inside* the loop body. The final **`print`** statement in line 10 of Listing 3.1 is not indented and is therefore *outside* the loop body.

Try out the code above if you haven't already. Does it behave the way you expect?

Now, notice that we ask for input twice in this program: once just before we try to enter the loop (line 4), and once at the end of the indented block inside the loop body (line 9). Why? The value returned by the initial **`input`** statement gets us into the loop body, though it doesn't have to. It is possible that the user types "quit" right away, in which case the program bypasses the loop and prints our "thank you" message. At the end of the loop body, we need to ask the user for the next input value, so that we have a new amount. Otherwise, the program would just continue to calculate the same total value over and over again and the program would never have the chance to end.

To see this for yourself, delete the last line of the loop body in Listing 3.1,

i.e., delete line 9. Instead of deleting it, you could just comment it out by putting a # symbol at the start of that line. Run your code and type an initial amount, say, 4.00. Your code will endlessly compute sales tax with no end in sight! Press the "Control" and "C" keys simultaneously to abort your program (we will write this Ctrl+C in the future because that's how programmers write such things).

With this in mind, we will often see loops written using the pattern in Figure 3.1.

```
x = some initial value
while condition involving x:
    code to be repeated
    x = a new value of x
```

Figure 3.1: A common while loop pattern

We call each new time through the body of the loop an *iteration*. We may refer to the first iteration, second iteration, next iteration, final iteration, etc., when discussing the behavior of a loop.

To get a sense of the mechanics of how while loops work, beginning programmers often consider "toy" examples that showcase a number of "gotchas" when it comes to programming loops. Consider Listing 3.2.

Listing 3.2: "Toy" while loop example 1

```
1 x = 1
2 while x <= 5:
3     print(x)
4     x = x + 1
5 print("Done.")
```

Run it. This prints the following.

```
1
2
3
4
5
Done.
```

We don't always need to have a non-indented statement following a loop. We are only doing this in the examples so it is clear in the output that we left the loop body.

In each iteration, we print the value of `x` and then increase `x` by `1`. Adding `1` to a value is such a common thing in programming that it has a special name: *increment*. That is, we are *incrementing* `x` by adding `1` to it. When `x` eventually becomes `5`, we print the `5` and then add `1` more to make `x` become `6`. When we evaluate the expression `x <= 5`, we get `6 <= 5`, which is `False`, so our code falls down to the non-indented section of code.

It is good to observe that, in this example, the last value we print is `5`, but the value of `x` when we exit the loop is `6`.

Let's look at another "toy" example. We will modify Listing 3.2, which was the following.

```
x = 1
while x <= 5:
    print(x)
    x = x + 1
print("Done.")
```

Notice that we've highlighted the statements that form the loop body. Let's change the order of those loop body statements to see what, if any, effect it has (Listing 3.3).

Listing 3.3: "Toy" while loop example 2

```
x = 1
while x <= 5:
    x = x + 1
    print(x)
print("Done.")
```

Run it. Our output is different from the first example.

```
2
3
4
5
6
Done.
```

Can you explain why this is?

In Listing 3.3, we are incrementing `x` before we print it. Thus, the first value we print is not `1` but `2`. The order of statements matters in programming, and it definitely matters in loops.

Consider another example (Listing 3.4).

Listing 3.4: "Toy" while loop example 3

```
1 x = 1
2 while x <= 5:
3     print(x)
4 print("Done.")
```

Here, we have removed the line that increments `x`. You should be able to guess what will happen when we run this program (hint: be ready to press `Ctrl+C`).

Since we have removed the line that increments `x`, `x` never changes, and so the program just prints `1` over and over again. Loops that are never-ending are called *infinite loops*. You will inadvertently create infinite loops every now and then when you program. It becomes very important to know how to debug them.

Let's try another example (Listing 3.5).

Listing 3.5: "Toy" while loop example 4

```
1 x = 0
2 while x <= 4:
3     print(x)
4     x = x + 1
5 print("Done.")
```

All we've done is changed the initial value of **x** and the value of **x** that causes us to not reenter the loop. The output is:

```
0
1
2
3
4
Done.
```

Let's make a small change to Listing 3.5. Suppose we change line 2 so that the code appears as it does in Listing 3.6.

Listing 3.6: "Toy" while loop example 5

```
1 x = 0
2 while x < 4:
3     print(x)
4     x = x + 1
5 print("Done.")
```

Now instead of **<=**, we have **<**. This makes the code end the loop one value of **x** earlier. The output is:

```
0
1
2
3
Done.
```

Let's change the condition in line 2 again so that it uses **>** instead of **<**. The code now reads as in Listing 3.7.

Listing 3.7: "Toy" while loop example 6

```
1 x = 0
2 while x > 4:
3     print(x)
4     x = x + 1
5 print("Done.")
```

What is the output? Why?
The output will be, as you may or may not have expected, is:

```
Done.
```

The variable `x` is set to zero initially. In the next line, the expression `x > 4` becomes `0 > 4`, which is `False`. Thus, the code bypasses the loop body entirely. There is no "rule" that says we have to do the loop body at least once. Computers are stupid. They only do what you tell them to. In this case, because the condition is `False` initially, we never do the loop body and go straight to the non-indented line of code past the loop body.

All of our "toy" examples thus far have involved incrementing a loop variable `x`, that is, adding `1` to `x`. However, we can change our variable however we wish. We could create the code in Listing 3.8.

Listing 3.8: "Toy" while loop example 7

```
1 x = 2
2 while x <= 8:
3     print(x)
4     x = x + 2
5 print("Who do we appreciate?")
```

In this code, we add `2` to `x` each time. The output of this code is:

```
2
4
6
8
Who do we appreciate?
```

We've used the heck out of the variable name `x`. Let's pick a different name for a while just for the sake of using something different. In Listing 3.9, we will use the name `count` for our loop variable name.

Listing 3.9: Blastoff example

```
count = 5
while count > 0:
    print(count)
    count = count - 1
print("Blastoff!")
```

In this code, we subtract `1` from `count` each time. This is called *decrementing* `count`. The output of this code is:

```
5
4
3
2
1
Blastoff!
```

Toy examples are a great way to understand how loops work from a "mechanical" perspective, but let's see how we can use them to solve real problems. Recall our very first example in this section, Listing 3.1, which dealt with reporting the price of an item after sales tax. Our next example is also related to money. Suppose we open a savings account, one that pays 5% interest per year into the account as long as we don't withdraw any money that we deposit. Let's say we start the account by depositing $10,000.00. How many years will it take to double our investment?

Mathematicians would attempt to derive an equation to solve this problem, but with computers we don't have to. We can make the computer figure it out for us. We just need to know how to write the code to tell the computer what to do.

The question "How many years will it take to double our investment?" tells us a lot about the code we would need to write. We would need to keep track of our balance as it grows by 5% each year. We would need to keep track of the number of years that have elapsed so far in our code. We would also need to check to see if the balance has doubled. If, in describing what our program is supposed to do, we say we need to check something, we are often talking about a Boolean condition found in an `if` statement or a `while` loop.

Try to program this example on your own first. You can't learn to be a good programmer if you don't try things regularly by starting with a blank screen or a blank sheet of paper. If you get really stuck, then (and only then) glance at the answer below.

Okay, let's look at a solution found in Listing 3.10 (Warning: Our initial solution will have flaws, as initial solutions often do).

Listing 3.10: Doubling investment example

```
balance = float(input("Enter starting balance: "))
years = 0
while balance < 2*balance:
    balance = balance * 0.05 + balance
    years = years + 1
    print("The balance after %d years is $%.2f." %
            (years, balance))
print("It will take %d years to double your investment." % years)
```

This is a good first attempt. Here, we are letting the loop do the work of adding to the balance year after year, each time checking to see if the balance has doubled and if we can leave the loop. However, our logic is flawed a bit. Consider the loop condition **`balance < 2*balance`**. A variable's value will never be greater than twice itself (unless that value is negative, but let's steer clear of negative values in bank accounts!).

The problem here is that we're trying to use the variable **`balance`** for two very different purposes simultaneously. On one hand, we are using **`balance`** to keep track of the current balance as it changes year after year. On the other hand, we're pretending that it still holds the starting balance from our initial investment. We should really keep the starting balance in a separate variable, which we will name **`startbalance`**.

With this in mind, we can modify the previous code as follows in Listing 3.11.

Listing 3.11: Doubling investment example - final version

```
startbalance = float(input("Enter starting balance: "))
balance = startbalance
```

```
years = 0
while balance < 2*startbalance:
    balance = balance * 0.05 + balance
    years = years + 1
    print("The balance after %d years is $%.2f." %
          (years, balance))
print("It will take %d years to double your investment." % years)
```

3.2 Manipulating and printing strings

Given our newly found exposure to loops, now is a good time to revisit strings. It turns out, there is a fair amount of nifty stuff we can do with strings using loops. Suppose we create a new string variable named **s** in the following way.

```
s = "abc def"
```

Programmers sometimes talk about different important parts of a string. They call these parts *substrings*. There are a lot of different possible substrings of **"abc def"** including **"a"**, **"ab"**, and **"def"**, just to name a few. The string **""** is also technically a substring of **s**. Note that **""** is different from **" "**. The former is called the *empty string*; it is a string without any characters. The latter is a string with a single space in it.

Note that the string **s** contains a space between the substrings **"abc"** and **"def"**. That will become an important observation later. Now, we can inspect individual characters by using square brackets and character's position in the string. Consider the following example.

```
print(s[0])
print(s[1])
print(s[2])
print(s[3])
print(s[4])
print(s[5])
print(s[6])
print(s[7])
```

If we run this code, we see the following output.

```
a
b
c

d
e
f
Traceback (most recent call last):
  File "/Users/shep/cs1/code/ch3strings.py", line 9, in <module>
    print(s[7])
IndexError: string index out of range
```

The syntax for retrieving a single character from a string **`s`** is **`s[index]`** where **`index`** is an integer representing the position of the character we wish to retrieve. The first character (in this case **`"a"`**) is at position **`0`** rather than position **`1`**. That may take a little getting used to. We will use the words "index" and "position" interchangeably to describe the location of a character in a string (also, note that the plural of "index" is "indices"). Let's re-list the code again in Listing 3.12, this time with comments to explain each line.

Listing 3.12: String indexing

```
1 s = "abc def"
2 print(s[0]) # print "a"
3 print(s[1]) # print "b"
4 print(s[2]) # print "c"
5 print(s[3]) # print " " (which ends up as just a blank line)
6 print(s[4]) # print "d"
7 print(s[5]) # print "e"
8 print(s[6]) # print "f"
9 print(s[7]) # Kablooey! There is no character at index 7.
```

Between the square brackets, you can place one index or you can give a range of indices, as we see in Listing 3.13.

Listing 3.13: String slicing

```
1 s = "abc def"
2 print(s[0:2])
```

```
3 print(s[0:3])
4 print(s[3:6])
5 print(s[3:])
6 print(s[:4])
```

This code yields the following output.

```
ab
abc
 de
 def
abc
```

We can observe that the first print in line 2 of Listing 3.13 does not print characters at indices **0** through **2**. Rather, it prints characters at positions **0** and **1** but not **2**. We might conclude from this that when we have two indices in square brackets following a string, the first number is the first position *inclusive* and the second number is the last position that will be selected up to, but not including, or *exclusive*. Note that the indices are separated by a colon (**:**). In this situation, we will call the colon the *slicing operator*.

Lines 4 and 5 in Listing 3.13 show us that the first and second indices of the slicing operator are optional. If one is omitted, Python assumes that the we mean "go to the ends of the string." When we use square brackets to select one or more characters from a string, we say that we are obtaining a substring of the original string. Obtaining a substring from some original string in this manner is called *string slicing*.

What appears to be indentation in lines 3 and 4 of the output is actually the single space found at index **3** in the string **s**. It is difficult to tell if there is a space at the end of line **5**. One way to tell is to append an additional character to the output to see the space. For example:

```
print(s[:4] + "$")
```

The output is:

```
abc $
```

Can you see the space in the output above?

There are quite a few things we can do with strings. Another interesting feature of strings is that they have their own set of special functions. Consider Listing 3.14.

Listing 3.14

```
1 t = "I like cheese."
2 shout = t.upper()
3 print(shout)
```

Most of the functions we have seen thus far (like **`print`**, **`input`**, etc.) do not have a period/dot in front of them. Calling a function which operates specifically on a value or a variable uses *dot-notation*, like **`t.upper()`**.

On the Internet, typing something in all capital letters is a convention that indicates shouting. In effect, the upper() function can be used on a string value (which is often contained in a variable) to return an all-uppercase version of that string. The output of Listing 3.14 is:

```
I LIKE CHEESE.
```

It is very important to note that **`upper()`** does not change the contents of the string variable **`t`**. If I do this:

```
t = "I like cheese."
shout = t.upper()
print(shout)
print(t)
```

I get this as output:

```
I LIKE CHEESE.
I like cheese.
```

Functions like **`upper()`** are not restricted to being called on variables. They may be called on any string value, like this:

```
shout = "I like cheese.".upper()
print(shout)
```

This works because Python transforms the first statement as shown in Figure 3.2.

Figure 3.2

A common mistake is to omit the parentheses at the end of **`upper()`**. Remember that **`upper`** is just the name of the function; the parentheses are what calls the function. If we remove the parentheses, we get output that looks strange (Listing 3.15).

Listing 3.15: Function call missing parentheses

```
1 t = "I like cheese."
2 print(t.upper)
```

This produces something like:

```
<built-in method upper of str object at 0x1019807e8>
```

This is Python's way of saying "yes, **upper** is the name of a function. If you want to actually use that function, put parentheses at the end." Let's add the parentheses to the end of the function call (Listing 3.16).

Listing 3.16: A fix for Listing 3.15

```
1 t = "I like cheese."
2 print(t.upper())
```

And now, it works.

Observe that when we "shout" by using **upper()**, the punctuation displayed is a period. Shouldn't we change that to an exclamation point (**!**)? Consider the following code in Listing 3.17.

Listing 3.17: Using **replace**

```
1 t = "I like cheese."
2 shout = t.upper().replace(".", "!")
3 print(shout)
```

The function **replace** is another string function. It takes two arguments. The first is a pattern to find in the string, and the second is a string we wish to insert instead.

Consider how Python executes line 2 of Listing 3.17 (see Figure 3.3).

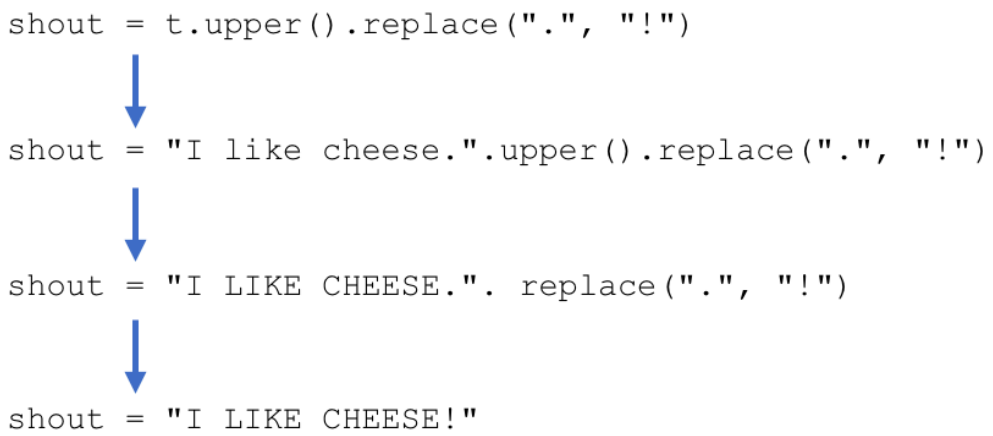


Figure 3.3

This example works because **t** is a string, so calling **t.upper()** returns another string. And, because **t.upper()** is a string, we can call **replace** on it to return yet another string. Calling a series of functions in one line is called *function chaining*.

Note that the original string value of **t** still hasn't changed. **t** is still **"I like cheese."**. If we wanted to actually change **t**, we would need to use an assignment statement as in Listing 3.18.

Listing 3.18: Using assignment with **replace**

```
t = t.upper().replace(".", "!")
```

Now, **t** is changed to the newly returned string value.

We can even chain calls to **replace** (see Listing 3.19):

Listing 3.19: Function chaining with **replace**

```
t = t.upper().replace("LIKE","LOVE").replace(".", "!")
```

As you might guess, and in the interest of completeness of our discussion, in addition to **upper** there is a function named **lower** for returning all lowercase versions of strings.

Now, let us explore the relationship between strings and loops. Consider the code in Listing 3.20.

Listing 3.20: Using a loop to print characters in a string

```
s = "abc def"
index = 0
while index < 7:
    print(s[index])
    index = index + 1
```

Listing3.20 produces the following output:

```
a
b
c

d
e
f
```

Why?

The variable **index** starts at **0**. Each time in the loop body, we print the value of the expression **s[index]**, which in the first loop iteration is **s[0]**, which has the value **"a"**. The code adds **1** to **index**, so now index is **1** (it was **0** previously). The code loops back around to the beginning of the loop. We print the value of **s[index]** again, which this time is **s[1]**, and **s[1]** is **"b"**. We continue in this fashion with each iteration of the loop having a new value of **index**. Eventually, **index** becomes **7**, at which point we exit the loop.

This is yet another "toy" example intended to get us to see the relationship between loops and strings, which is a powerful relationship we will exploit very shortly.

Let's make one change to Listing 3.20. We will change only the value of the string variable **s**. Our new code is in Listing 3.21.

Listing 3.21

```
s = "protagonist"
index = 0
while index < 7:
    print(s[index])
    index = index + 1
```

The output is now:

```
p
r
o
t
a
g
o
```

Oh no! Where is the rest of the string? Why did it stop after `7` characters? It stopped after `7` characters because we told it to!

Unfortunately, we have *hard-coded* the loop condition as `index < 7`. We could change the value of `s`, and we shouldn't assume the length of `s` will always be `7` characters. From a logic standpoint, the `7` is supposed to refer to the length of the string, which is also `1` more than the last index in the string. Recall from the previous example that the last index of `s` was `6` and the number of characters (the length of the string) was `7`. In Listing 3.21, the length of the string is `11`, not `7`.

Any time we hard-code a value, like `7`, it makes our code brittle and less resistant to change. Let us replace this line:

```
while index < 7:
```

with this line:

```
while index < len(s):
```

The **len** function returns the number of characters in the string. Our new code is shown in Listing 3.22.

Listing 3.22

```
s = "protagonist"
index = 0
while index < len(s):
    print(s[index])
    index = index + 1
```

Alternatively, we could have done things a little differently. See Listing 3.23 and note the highlighted lines.

Listing 3.23

```
s = "protagonist"
index = 0
lastindex = len(s) - 1
while index <= lastindex:
    print(s[index])
    index = index + 1
```

We have created a new variable named **lastindex**. In the case of the string **"protagonist"**, the value of **lastindex** will be **10** since **len(s)** is **11**. It is very important to see the relationship between the last index and the length of the string. Because indices start at **0**, not **1**, the last index is always the length minus **1** (in this case, the value of the expression **len(s)-1**).

Let's see how well this is all sinking in. Can you write code to print the characters of a string, one on each line, but in reverse order this time? In other words, if my string is **"abc"**, can you write code that prints:

```
c
b
a
```

Give it a try.

Let's check your work. With some experimentation, you might have been able to get close to a solution that looks like the code in Listing 3.24. (If not, keep trying and practicing.)

Listing 3.24

```
s = "abc"
index = len(s) - 1
while index >= 0:
    print(s[index])
    index = index - 1
```

Note that we are setting the initial index variable to the last index of string (again, length minus `1`), and in our loop body we are subtracting `1` rather than adding `1`.

In Python, as it is in other programming languages, there is often more than one way to write a program. Another approach to printing a string in reverse could take advantage of the fact that you can put *negative* indices in square brackets to obtain characters in a string starting from the end rather than from the beginning. For example, suppose we have the following code.

```
foo = "abc"
print(foo[-1])
print(foo[-2])
print(foo[-3])
```

This prints:

```
c
b
a
```

With this in mind, we could write new code that produces the same output as Listing 3.24, but in a different way. Consider Listing 3.25

Listing 3.25

```
foo = "abc"
index = -1
while index >= -len(foo):
    print(foo[index])
    index = index - 1
```

Note the loop condition. If `len(foo)` in this case is `3`, then `-len(foo)` is `-3`.

(Why did we name our variable `foo`? That's weird, right? You're correct; it is weird! There's a historical reason why computer science books often name variables `foo` and `bar`. Search the Web for `foobar` and its original form `FUBAR` so that you can learn where this nomenclature comes from.)

3.3 `for` loops

Loops of the form found in Listing 3.26 are very common. Suppose `n` is an integer variable defined earlier in a program.

Listing 3.26

```
k = 0
while k < n:
    print(k)
    k = k + 1
```

Loops of this form are so common, in fact, that there is a shorter version of the loop known as a `for` loop. Listing 3.26 could be re-written as a `for` loop as shown in Listing 3.27.

Listing 3.27

```
for k in range(0, n):
    print(k)
```

Gone is the explicit declaration of the variable **k**. Gone also is the increment statement **k = k + 1**. The range expression gives **k** an initial value of **0** in this example, and then it allows the loop to continue until **k == n**, at which point the computer leaves the loop.

Recall Listing 3.22 in Section 3.2, which is shown below. To print the characters of a string, each on a separate line, using a **while** loop, we could write:

```
s = "protagonist"
index = 0
while index < len(s):
    print(s[index])
    index = index + 1
```

We can now re-write this code using the shorter, more succinct **for** loop (see Listing 3.28).

Listing 3.28

```
s = "protagonist"
for index in range(0, len(s)):
    print(s[index])
```

for loops can also countdown or count in multiples just as easy as **while** loops can. Suppose we want to print a string in reverse order like we did in Listing 3.24. As a reminder, the code looked like this:

```
s = "abc"
index = len(s) - 1
while index >= 0:
    print(s[index])
    index = index - 1
```

Instead, we can add a third argument to **range** which specifies what we add to the loop variable to change it prior to each iteration of the loop. Thus, we can do this (Listing 3.29).

Listing 3.29

```
s = "cba"
for index in range(len(s)-1, -1, -1):
    print(s[index])
```

That's a lot to mentally unpack. Let's take it a piece at a time. Let's take the first emphasized expression in **range** below.

```
for index in range(len(s)-1, -1, -1)
```

This sets the last index in the string **len(s)-1** as the initial value of the variable **index**. The next expression/value we wish to emphasize in **range** is:

```
for index in range(len(s)-1,-1, -1)
```

This says that when **index** goes past **0** and reaches **-1**, we should exit the loop. Finally, we have:

```
for index in range(len(s)-1, -1,-1)
```

This **-1** specifies how **index** should be changed prior to each iteration. This means that **index** will have **1** subtracted from its value.

Let's see how we are doing here. Suppose I have the code in Listing 3.30.

Listing 3.30

```
for i in range(1, 9, 2):
    print(i)
```

What is the output of this code?

We would expect to see the following on-screen.

```
1
3
5
7
```

Can you explain why?

The variable **i** gets the value **1** to start the loop. In each iteration, the value of **i** is printed. Just before each iteration, we add **2** to **i**. Therefore, **i** will have the values **1**, **3**, **5**, and **7** within the loop body. When **i** reaches **9**, the **for** loop will end.

Let us try to use our knowledge of **for** loops to write a program that reverses a string. In other words, given user input, reverse the input and print it to the user. If the user enters **"abc def"**, then the program should output **"fed cba"**.

Here is our attempt (Listing 3.31).

Listing 3.31

```
1 s = input("Enter a string: ")
2 rev = ""
3 for i in range(len(s)-1, -1, -1):
4     rev = rev + s[i]
5 print("The reverse of the string is: %s." % rev)
```

s is the input string variable and **rev** will be the string variable that holds **s**'s reverse. We take each character from **s** using a **for** loop and we concatenate it onto the end of **rev**.

Consider how **rev** changes with each iteration of the loop, as shown in Table 3.1.

Table 3.1 is known a *trace table*. It serves as a tool for programmers to make sense of how values change from iteration to iteration of a loop.

3.4 for loops with strings, revisited

We've barely scratched the surface of what **for** loops can do for us. **for** loops in Python have a nifty additional form that allows us to write even simpler,

Table 3.1: Trace table for Listing 3.31

rev	i	s[i]
"f"	6	"f"
"fe"	5	"e"
"fed"	4	"d"
"fed "	3	" "
"fed c"	2	"c"
"fed cb"	1	"b"
"fed cba"	0	"a"

more readable code. We'll explore this alternate form now. Let's set it up in the context of an example. Suppose we wish to iterate through the characters of a string **s** with the intent of reversing it. We might write the code found in Listing 3.32.

Listing 3.32

```
1 s = input("Enter a string: ")
2 rev = ""
3 for index in range(0, len(s)):
4     rev = s[index] + rev
5 print("The reverse of the string is: %s." % rev)
```

Notice how we've changed the logic from what we did in Listing 3.31, our previous attempt. In the previous code, we traversed the string starting at its last character and working towards its first character, and for each character, we appended it to the end of the reversed string **rev**. In this code (Listing 3.32), we are doing the opposite. We are traversing the string from first character (the "front" of the string) to last character (the "back" of the string) and prepending the character to the front of the reversed string **rev**. Both methods solve the same problem and produce the same output.

As it turns out, we don't even need to concern ourselves with having an **index** variable. Python has a version of the **for** loop syntax that operates exclusively on strings. Consider the following highlighted adjustments to Listing 3.32 shown in Listing 3.33.

Listing 3.33

```
s = input("Enter a string: ")
rev = ""
for ch in s:
    rev = ch + rev
print("The reverse of the string is: %s." % rev)
```

In this form of the Python **for** loop, the loop variable **ch** is assigned a different character from the string **s** each time the loop condition is checked. Notice that there is no more **index** variable or **range** function.

3.5 Loops for input validation and translation

We have been asking users for input in many of our previous examples. We have always assumed users will type the correct information, but what happens when they do not. Suppose we ask users to enter their age. We have been doing this as follows.

```
age = int(input("Enter your age: "))
```

What if users' fingers slip and they enter **"1r"** instead of **"14"**? The **int** function will crash because **"1r"** cannot be cast to an integer. This happens because we have prematurely tried to cast the string to an integer without checking the input first. We need to wait to perform the **int** cast until we know it is safe to do so. Another way to say this is we need to postpone the type conversion until we have *validated* the input. The code in Listing 3.34 demonstrates how we might do input validation before casting a string to an integer.

Listing 3.34

```
age_str = input("Enter your age: ")
while not age_str.isdigit():
    print("That is not a valid age.")
    age_str = input("Enter your age: ")
age = int(age_str)
```

isdigit (line 2 of Listing 3.34) is a function that we can always use on a string value. The "dot" (**.**) separates the word **isdigit** from the string it refers to, in this case the value stored in the variable **age_str**. **age_str.isdigit()** returns **True** if every one of the characters in **age_str** are numbers, otherwise it returns **False**.

(Note: What do you suppose would be the value of the expression **"".isdigit()**? In other words, is the result of **isdigit** **True** or **False** when the string contains no characters at all. You may wish to try typing this statement into the Python Shell for yourself to see what happens.)

Let us examine another example that uses loops to operate on strings. Suppose we ask users for their full names using an **input** statement.

```
fullname = input("Enter your full name: ")
```

The variable **fullname** could contain **"Javier Sanchez"**, **"Mary Johnson"**, or just about anything, though in most cases, there will be a first name and a last name separated by a space. Let us try to extract the first name from the value in **fullname**. To do this, we'll need to use the slicing operator (**:**). Depending on the length of the first name, the range of the string slice will be a little different.

```
# Suppose fullname == "Javier Sanchez"
#                     01234567...
firstname = fullname[0:6]

# Or, suppose fullname == "Mary Johnson"
#                          01234567...
firstname = fullname[0:4]
```

Since we can't possibly know the length of the first name ahead of time, we will need to find the index of the space in order to slice the appropriate substring. If we are able to put the index of the space into a variable named **space**, then we can extract the first name as follows.

```
firstname = fullname[0:space]
```

If **fullname** is **"Javier Sanchez"** then **space** would be **6**, if **fullname** is **"Mary Johnson"** then **space** would be **4**, and so on. Finding the index of the space requires us to search the string character by character, which is something we have already done, both with **while** loops and with **for** loops.

Here is our strategy. We will use a loop to examine each character in **fullname**, one at a time. If we encounter the space, we will set the value of **space** to the current index of the loop, and then we will jump out of the loop using the command **break**. After all, once we've found the space, our work is done; there is no need to keep looping.

Thus, our code appears in Listing 3.35.

Listing 3.35: Finding the index of a space in a string

```
1 fullname = input("Enter your full name: ")
2 for index in range(0, len(fullname)):
3     if fullname[index] == ' ':
4         space = index
5         break
6
7 firstname = fullname[0:space]
8 print("Your first name is %s." % firstname)
```

Try out this code on several full names that you can think of. Try to find *edge cases* that may not work. *Edge cases* are exceptional cases of program input that could cause your code to fail. One such edge case in this problem is full names that do not have spaces, i.e., the names of individuals who only have one name. Consider celebrities' names such as Prince, Cher, Sting, Madonna, etc. Those names are their full names and first names simultaneously. Try typing "Prince" in as the input to your program.

```
Enter your full name: Prince
Traceback (most recent call last):
  File "/Users/shep/cs1/code/ch3fullname.py", line 6, in <module>
    firstname = fullname[0:space]
NameError: name 'space' is not defined
```

Can you explain this error to yourself? Why is **space** not defined in line **6**? Where in the code do we create the **space** variable normally?

In this case, **space** is first created inside the loop. This is generally bad programming practice. If the variable is used by code outside of a loop, the variable should also be defined outside the loop. So, let's define **space** before we reach the loop, but what should be its initial value? Let's choose a value that doesn't look like a normal index. Normal indices would be **0**, **1**, **2**, **3**, etc., so let's choose **-1** as the initial value of **space**. Our code can now be written as in Listing 3.36.

Listing 3.36: Finding the index of a space in a string, attempt 2

```
1  fullname = input("Enter your full name: ")
2
3  space = -1
4  for index in range(0, len(fullname)):
5      if fullname[index] == ' ':
6          space = index
7          break
8
9  if space == -1:
10     firstname = fullname
11 else:
12     firstname = fullname[0:space]
13
14 print("Your first name is %s." % firstname)
```

Finding a character or a pattern in a string is such a common operation, Python gives us a way to do it without writing a loop every time. We can simplify the code above using the **find** function as follows in Listing 3.37.

Listing 3.37: Finding the index of a space in a string, attempt 3

```
1  fullname = input("Enter your full name: ")
2
3  space = fullname.find(" ")
4
5  if space == -1:
6      firstname = fullname
7  else:
8      firstname = fullname[0:space]
9
10 print("Your first name is %s." % firstname)
```

find (line 3 of Listing 3.37) is a function that we can always use on a string value. The "dot" (**.**) separates the word **find** from the string it refers to, in this case the value stored in the variable **fullname**. The argument is the single character or multiple character pattern to search for. **find** returns the index where that pattern starts in the value of **fullname**.

Here are some examples of using **find**.

```
print("abcde".find("b"))     # prints 1
print("abcde".find("cd"))    # prints 2
print("abcde".find("cash"))  # prints -1, which indicates the
                             # pattern was not found
print("abcde".find("abcde")) # prints 0
```

Now that we know how to find a pattern (and specifically, a space) in a string, consider what that allows us to do. Suppose we have a sentence that we wish to break into individual words. How are words separated in the English language? If you said "using spaces" then you are correct (for the most part).

Let's try to do something like this through an example. If you spend time on the Web, you might have learned that Internet citizens have tabbed September 19th as "International Talk Like a Pirate Day." Imagined pirate-speak draws from many sources in popular culture. Fictional pirates might say "Ahoy!" instead of "Hello!" or they might use the word "matey" to describe a friend. If we wanted to translate a single English word into its Pirate equivalent, we might do something like Listing 3.38.

Listing 3.38

```
1  english_word = input("Enter a word: ")
2  if english_word == "hello":
3      pirate_word = "ahoy"
4  elif english_word == "friend":
5      pirate_word = "matey"
6  else:
7      # The word has no known pirate translation.
8      # Leave it in English.
9      pirate_word = english_word
10 print("%s in pirate is %s." % (english_word, pirate_word))
```

What if we wanted to change an entire English sentence into Pirate instead of just translating a single word? How would we accomplish that?

Let's keep this as simple as possible, so for now let's ignore "edge cases" like uppercase letters and punctuation. Let's only worry about translating all-lowercase words separated by spaces.

You might not have any idea how to accomplish this in code, but there are a number of ways to get started. One way is to realize that every computer program tends to take input and transform it into output. If we envision a program as taking variables and changing their values in order to reach a goal, we can write down how we want those variables to change, and then write down the code to perform the changes.

Let's think about what variables we would want to have to make an English-to-Pirate translator program work. Suppose we have two variables, one named **sentence** and the other named **word**. **sentence** contains the English sentence the user types, and **word** will store each word as we extract it from **sentence**. Each time we extract an English word into **word**, the string in **sentence** should shrink.

Again, if we are thinking about a program as something that makes changes to input variables, we can envision a program looping to change **sentence** and **word** repeatedly. Here is an example.

```
word: ""         sentence: "hi there my friend"
word: "hi"       sentence: "there my friend"
word: "there"    sentence: "my friend"
word: "my"       sentence: "friend"
word: "friend"   sentence: ""
```

Think of this process in a loop. Each time through the loop, we find the index of the space in **sentence** and put that index into a variable named **space**. We use the slicing operator to get the first word in the sentence from **0** to **space**. Then, we use the slicing operator again to "shrink" **sentence** so that **sentence** only contains the remaining words.

This would allow us to translate each word, one at a time. Let's write some code. Our first attempt is found in Listing 3.39.

Listing 3.39

```
sentence = input("Enter an English sentence: ")
word = ""
print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

space = sentence.find(" ")
while space != -1:
    word = sentence[0:space]
    sentence = sentence[space+1:]
    print("word: \"%s\", sentence: \"%s\"" % (word, sentence))
    space = sentence.find(" ")
```

Let's try it out.

```
Enter an English sentence: hello there my friend
word: "" sentence: "hello there my friend"
word: "hello" sentence: "there my friend"
word: "there" sentence: "my friend"
word: "my" sentence: "friend"
```

This is pretty good, but what about the final word in this example: "friend". Since there is no space trailing the last word in the sentence, we will need to add some code after the loop to take care of the last word. Consider Listing 3.40.

Listing 3.40

```
sentence = input("Enter an English sentence: ")
word = ""
print("word: \"%s\", sentence: \"%s\"" % (word, sentence))

space = sentence.find(" ")
while space != -1:
    word = sentence[0:space]
    sentence = sentence[space+1:]
    print("word: \"%s\", sentence: \"%s\"" % (word, sentence))
    space = sentence.find(" ")

if sentence != "":
    word = sentence
    sentence = ""
    print("word: \"%s\", sentence: \"%s\"" % (word, sentence))
```

The highlighted block of code in Listing 3.40 checks to see if there's still text residing in **sentence**.

Now all that's left is to do the translating. We will take each **word** and append its translation to a new string which we'll call **pirate**. We can get rid of our **print** statements, too, since they were only there to aid our understanding of what the code does. Listing 3.41 is our final code.

Listing 3.41: Final English-to-Pirate Translator

```
sentence = input("Enter an English sentence: ")
word = ""
pirate = ""

space = sentence.find(" ")
while space != -1:
    word = sentence[0:space]
    sentence = sentence[space+1:]
    if word == "hello":
        pirate = pirate + "ahoy" + " "
    elif word == "friend":
        pirate = pirate + "matey" + " "
    else:
        pirate = pirate + word + " "
    space = sentence.find(" ")

if sentence != "":
    word = sentence
    sentence = ""
    if word == "hello":
        pirate = pirate + "ahoy" + " "
    elif word == "friend":
        pirate = pirate + "matey" + " "
    else:
        pirate = pirate + word + " "

print("Your sentence translated to pirate is:")
print(pirate)
```

An example of a running program is:

```
Enter an English sentence: hello there my friend
Your sentence translated to pirate is:
ahoy there my matey
```

We can add more translations by adding more **`elif`** conditions to the **`if`**-blocks of statements, though we must add them both to the **`while`** loop code and the **`if`**-block that follows the **`while`** loop. Having to add this logic in two different places makes the code more difficult to maintain, since we might add an **`elif`** to the earlier part of the code but then forget to add it to the later part of the code. In Chapter 4, we will examine functions, which will improve the maintainability of this code dramatically.

3.6 Nested loops

Some interesting and practical things happen when we use a loop as the body of another loop. Consider Listing 3.42, and try to guess as to its output.

Listing 3.42

```
for outer in range(0, 3):
    for inner in range(0, 3):
        print("outer = %d, inner = %d" % (outer, inner))
```

Let's follow the code step-by-step as we would with any code. The first statement we encounter is **`for outer in range(0, 3)`**. Since this is first time we've encountered this statement, this code sets **`outer`** to **`0`** and we enter its loop body. The first statement of the loop body is, of course, another loop. This loop statement is **`for inner in range(0, 3)`**. So, this sets **`inner`** to **`0`** and we enter its loop body. This means that at this point in the code, we are actually inside two loops: an inner one, which is inside an outer one. The first (and only) statement in the inner loop prints the values of **`outer`** and **`inner`** so that we can see how the values change throughout the program.

We call a loop-inside-a-loop construct a set of *nested loops*. That is, the inner loop is nested inside the outer loop.

Before we continue discussing how these loops work, run the code to see the output.

```
outer = 0, inner = 0
outer = 0, inner = 1
outer = 0, inner = 2
outer = 1, inner = 0
outer = 1, inner = 1
outer = 1, inner = 2
outer = 2, inner = 0
outer = 2, inner = 1
outer = 2, inner = 2
```

We can get a pretty good idea of how this works from looking at the output. Once we enter the outer loop, we encounter the inner loop right away. Python will stay in the inner loop until it is completed, and then it will exit to the outer loop. The outer loop will increment **`outer`** by one, and since **`outer`** is still within its range, it will enter the outer loop body, which, once again, will reach the inner loop. Since we left the inner loop once already, reaching the inner loop this time is like reaching it for the first time. The statement **`for inner in range(0, 3)`** sets **`inner`** to **`0`** once again, and the process repeats itself.

We often use nested loops to write programs in Python. For one example, suppose we wish to write a program that repeatedly asks for students' names and exam scores. The output should be students' names followed by their exam average. Listing 3.43 shows how to do this using nested loops.

Listing 3.43

```
1  student = input("Student name? ('quit' to end) ")
2  while student != "quit":
3      sum = 0.0
4      count = 1
5      score = input("Score %d? ('quit' to end)  " % count)
6      while score != "quit":
7          score = float(score)
8          sum += score
9          count += 1
10         score = input("Score %d? ('quit' to end)  " % count)
11
12     # count is one more than it should be since we
13     # increment count before we quit.
14     count -= 1
15
16     if count > 0:
17         average = sum / count
```

```
18          print("%s's average is %.2f" % (student, average))
19      else:
20          print("No grades entered for %s." % student)
21
22      student = input("Student name? ('quit' to end) ")
```

One thing to note about this code, besides the nested **while** loops, is the shorthand assignment statements. Instead of typing

```
count = count + 1
```

we can type

```
count += 1
```

This type of shorthand assignment construct works for most operators in Python.

Beyond that observation, we note that the outer **while** loop is responsible for getting each student's name. The inner **while** loop retrieves each score and adds it to a sum. Once we exit the inner loop, we can calculate the average using the sum. We must be careful to make sure that there are any scores at all. If **count** were **0** and we divided by **0** without checking, our program would crash.

The output of this code might look something like the following depending on what the user types.

```
Student name? ('quit' to end) David Chan
Score 1? ('quit' to end)  95.5
Score 2? ('quit' to end)  80.5
Score 3? ('quit' to end)  87.0
Score 4? ('quit' to end)  quit
David Chan's average is 87.67
Student name? ('quit' to end) Sarah McDowell
Score 1? ('quit' to end)  quit
No grades entered for Sarah McDowell.
Student name? ('quit' to end) quit
```

3.7 Exercises

1. Write a program that asks for a word, phrase, or sentence. The program should then print whether the input is a palindrome. (Do a Web search to see what a palindrome is.)

2. Write a program that repeatedly asks for a number until the user enters "quit." The program should print the sum and average of all the numbers.

3. What is the output of the following program?

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

4. What is the output of the following program?

```
for i in range(5, 0, -1):
    print(i)
```

5. What is the output of the following program?

```
for i in range(1, 6, 2):
    for j in range(1, 3):
            print("%d %d" % (i, j))
```

6. Write an English-to-Pirate translator like the one found in Section 3.5, only have your program repeatedly ask for sentences until the user presses ENTER/RETURN without typing anything. You can still ignore capitalization and punctuation.

7. Do the previous problem again, only this time properly handle capitalization and punctuation.

8. Write a program that translates English sentences to Pig Latin. To form the Pig Latin equivalent of a word, remove the first consonant sound of the

word and append it to the end preceded by a dash and followed by "ay". Thus, "cat" becomes "at-cay" and "ship" becomes "ip-shay." The latter example demonstrates that consonant sounds can be blended consonants.

If an English word begins with a vowel, the word is simply restated with "-way" appended to it. In other words, "apple" becomes "apple-way." Your solution should handle capitalization and punctuation.

Example of a working program:

```
Enter English (press 'ENTER' only to quit): Hey there, Delilah!
The Pig Latin equivalent is: Ey-hay ere-thay, Elilah-day!

Enter English (press 'ENTER' only to quit): Welcome to the Apple Store.
The Pig Latin equivalent is: Elcome-way o-tay e-thay Apple-way Ore-stay.

Enter English (press 'ENTER' only to quit):

Thank you for using the Pig Latin translator!
```

9. Write a program that asks for a number and then prints that number of rows of 5 asterisks. For example, if the user enters 4, the program would print the following.

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

10. Write a program that asks for two numbers x and y. The program should print a rectangle of asterisks consisting of x rows and y columns. For example, if the user enters 3 and 8, respectively, the program would print the following.

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

11. Write a program that asks for a number and then prints a triangle of asterisks where the base has that number of asterisks. For example, if the

user enters 4, the program would print the following.

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

12. Write a program that asks for a number and then prints an inverted triangle of asterisks where first row contains that number of asterisks. For example, if the user enters 4, the program would print the following.

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

13. Python programs are capable of generating random numbers (well, "pseudo"-random numbers – more about that in the next chapter). One example of how to do this is as follows.

```
import random
x = random.randint(1,10)
print("Here is a random integer between 1 and 10: %d" % x)
```

If you run this program over and over again you'll see the number change. We can use random integers to simulate real-life occurrences, including games of chance and movement of animals.

Consider one program that demonstrates the use of random numbers. A drunkard stumbling around in a grid of streets picks one of four directions (north, south, east, or west) at each intersection, and then he moves to the next intersection.

Write a program that simulates the drunkard's walk. Represent each intersection location as (x,y) integer pairs and have the drunkard start at (0,0). Have the drunkard walk 100 intersections and print the intersection location where he ends up. Run this program many times. Does the drunkard tend to stay close to (0,0) or does he end up moving far away?

Chapter 4

Functions

4.1 Introduction to Functions

Consider Listing 4.1.

Listing 4.1

```
print("This is program one.")
x = input("Enter a string: ")
if x[0].lower() in "aeiou":
    print("That string starts with a vowel.")
else:
    print("That string does not start with a vowel.")

print("This is program two.")
x = input("Enter a string: ")
if x.isalnum():
    print("That string only contains only a-z's and 0-9's.")
else:
    print("That string contains some \"weird\" characters.")
```

Each of these two sections of code are supposed to be their own separate programs. If I type this code into one code file and run it, of course, both programs will be executed, one after the other. This is, after all, how programs work. Python executes each statement, one after another, and since "program one" comes before "program two," all of program one's statements will be executed and then all of program two's statements will be executed.

In program one, we are checking the first character of **x** using the expression **x[0]** (see line 3 of Listing 4.1). We change it to lowercase, and then we use the **in** operator to see if that single character is a vowel. Without the **in** operator, we would need to write an **if** statement that looks something like this.

```
if x[0].lower() == "a" or x[0].lower() == "e" or \
   x[0].lower() == "i" or x[0].lower() == "o" or \
   x[0].lower() == "u":
```

Yuck.

Let's try something new. Let's construct Listing 4.2 by modifying Listing 4.1 using a new language construct.

Listing 4.2

```
1  def one():
2      print("This is program one.")
3      x = input("Enter a string: ")
4      if x[0].lower() in "aeiou":
5          print("That string starts with a vowel.")
6      else:
7          print("That string does not start with a vowel.")
8
9  def two():
10     print("This is program two.")
11     x = input("Enter a string: ")
12     if x.isalnum():
13         print("That string only contains only a-z's and 0-9's.")
14     else:
15         print("That string contains some \"weird\" characters.")
```

Now run this code. What happens? Yes, nothing! There is absolutely no output, which suggests that none of our statements ran. Oh dear. Well, let's add a bit more code (see the highlighted lines in Listing 4.3).

Listing 4.3

```
1  def one():
2      print("This is program one.")
3      x = input("Enter a string: ")
```

```
4       if x[0].lower() in "aeiou":
5           print("That string starts with a vowel.")
6       else:
7           print("That string does not start with a vowel.")
8
9
10  def two():
11      print("This is program two.")
12      x = input("Enter a string: ")
13      if x.isalnum():
14          print("That string only contains only a-z's and 0-9's.")
15      else:
16          print("That string contains some \"weird\" characters.")
17
18  # Add the following line.
19  one()
```

Try again. Aha! Only the statements of code listed underneath **`def one():`** are executed. Now, try changing **`one()`** to **`two()`** in Listing 4.3. Run it. Now only the statements under **`def two():`** are executed.

What's happening here? Any code that is not indented at all is considered part of the *main* program. Code that is indented beneath a line that starts with **`def`** is like a little mini-program. **`def`** stands for "define." In this case, we are defining a mini-program and giving it a name. Mini-programs are better known as… (*drum roll please*)… *functions*!

So, in Listing 4.3, the main program is simply on lines 18 and 19 (though line 18 is just a comment). Let's change the main program to see what happens. See Listing 4.4 (specifically lines 18 - 20).

Listing 4.4

```
1   def one():
2       print("This is program one.")
3       x = input("Enter a string: ")
4       if x[0].lower() in "aeiou":
5           print("That string starts with a vowel.")
6       else:
7           print("That string does not start with a vowel.")
8
9
10  def two():
11      print("This is program two.")
12      x = input("Enter a string: ")
```

```
13      if x.isalnum():
14          print("That string only contains only a-z's and 0-9's.")
15      else:
16          print("That string contains some \"weird\" characters.")
17
18  print("Main: calling a function.")
19  two()
20  print("Main: end of program.")
```

Run this, and notice what happens. The code starts in the main non-indented section by printing **`"Main: calling a function."`** Then, the expression **`two()`** calls the function, which makes the program "jump" up to the definition of the function, which starts with **`def two():`**. Each of the statements indented under **`def two():`** are executed, one by one, and then once those statements are finished, we jump back down to where **`two()`** was called in the first place. Finally, the main program resumes and we print **`"Main: end of program."`**

In a sense, functions are like "tangents" (not the math kind of tangent – the tangent where we deviate from a conversation to another topic briefly) With functions, we can momentarily leave the main program to execute some pre-prepared code. We have been *calling* functions throughout the entire book already. We have used functions like **`print`**, **`input`**, and **`int`**. Now we have a sense of how to *define* our own new functions.

When we define a new function, we give the function a name and we give the function an indented section known as the function's *body*. There can be more pieces to a function as well, and we will learn about those in the next few sections.

4.2 Parameters

Let's look at another example of how we can define our own functions that we can call later. Suppose, at a certain fictitious university named Higher Ed University, students are given email addresses that follow a naming convention. To form the email address, we take the first four characters of a student's last name, then the first three characters of a student's first name, and then append

"@heu.edu" to it. We could write a function to determine and print out such an email address.

Listing 4.5: Defining `print_email`

```
1 def print_email(first, last):
2     username = last.lower()[0:4]
3     username = username + first.lower()[0:3]
4     print("%s@heu.edu" % username)
```

Notice that the `print_email` function is defined differently than `one` or `two` in Listing 4.4 in Section 4.1. In our previous example, when we defined those functions, their definitions had empty parentheses after the function name. Now, we have two things that look a lot like variables inside the parentheses. In fact, they are variables, and their values are used inside the function's body. Variables that are given inside the parentheses in a function definition are called the function's *parameters*.

Let us call this function. We might type this.

```
print_email("Jack", "Jackson")
```

This would print:

```
jackjac@heu.edu
```

Or, if we typed this:

```
print_email("Shania", "Carrington")
```

It would print:

```
carrsha@heu.edu
```

How does this work? When we call a function, we must provide the correct number of values to the function, in the right order. The correct number of values is based on how many parameters the function has. In this case, the function **`print_email`** has two parameters, **`first`** and **`last`**. The *values* we pass to the definition of a function are called the *arguments*. When a function is called, the arguments' values become the parameters' values. Thus, **`first`** gets the value **`"Shania"`** and **`last`** gets the value **`"Carrington"`**.

Clever readers might suspect that this function has a bug. It looks like if the last name is fewer than four characters long or the first name is fewer than three characters long, the program would crash. Surprisingly, in Python the string range operator is very forgiving. If you were to type the expression **`"abc"[0:20]`**, the substring returned will still be **`"abc"`**. We can read this substring operation aloud as "Start at index 0 and return any characters up to index 20, if it exists." Therefore, there is no bug, but skeptical thinking like this will help us as we program in the future.

What happens when you switch the order of the arguments?

```
print_email("Carrington", "Shania")
```

What happens if you do not have the correct number of arguments?

```
print_email("Shania")
```

Try these things on your own to see what happens. Try to guess what they do before you actually run the code. If they produce errors, make a mental note of what the error looks like. That way, if you encounter that error again in the future, you'll know what caused it and how to fix it.

Let's move on to a different example. Let us write code that defines a new function, and then let's write some code to call that new function (see Listing 4.6).

Listing 4.6: Defining and calling **print_money**

```
1 def print_money(amount):
2     print("$%.2f" % amount)
3
4 my_money = 50.25
5 print_money(my_money)
```

This code will print:

```
$50.25
```

It does not matter when we give an argument to a function call if we provide a value or an expression that returns a value (in this case, the name of a variable).

Now, let's make one more change to the code (see Listing 4.7).

Listing 4.7: Defining and calling **print_money**, attempt 2

```
1 def print_money(amount):
2     print("$%.2f" % amount)
3     amount = amount + 10.00
4
5 my_money = 50.25
6 print_money(my_money)
7 print_money(my_money)
```

In this example, we are doing something new. We are taking the parameter **amount** and actually changing its value in the last statement of the function's body. Where does **amount**'s value come from? It is passed the value of **my_money** down in the main program. One thing we should look out for is "side effects." Does changing the value of **amount** have any effect on the value of **my_money**? In other words, does changing **amount** also change **my_money**? If it did, we would expect the second call to **print_money** to print a different value than the first.

As it turns out, when we run the code, we see this:

```
$50.25
$50.25
```

Changing **amount** does not change **my_money**. This is because arguments are passed to parameters in Python using their value alone. This is called *pass by value*. If changing the parameter had actually changed the argument, then we would have said Python practices *pass by reference*, where a reference to the original variable would have been given to the function rather than just the value. Remember that Python uses pass by value when you are writing function definitions.

It is generally bad programming practice to change the values of parameter variables in the body of a function – again, generally speaking – but there are exceptions. For the most part, a programmer should be able to consult the parameters at any point in the function's body to see what the "input" to the function was. There is one situation, however, where it is desirable to modify the parameter values passed to a function. We will discuss this situation in Chapter 5.

Terminology time! The first line of a function definition is called the function's *signature*. The signature tells us the name of the function and how many parameters it has. For example, suppose we have the following function signature.

```
def print_uppercase(words):
    # function body omitted
```

We can see from this signature that if we want to call this function, we must use the name **print_uppercase** and we must pass one argument to the function. The argument will be assigned to the parameter **words**.

We will use the term signature again in this book, so you should learn to use it, too!

4.3 Return values

If we think of functions in terms of them being like mini-programs, we might observe that programs tend to ask for input, they doing something with the input, and then finally they produce output. In a sense, parameters give programmers the ability to pass "input" to a function. What then might we do to allow functions to provide "output" back to the main program?

As you might recall, we have already seen functions that do this – functions that Python already has defined for us. For example:

```
ssn = input("Enter your social security number: ")
```

input is a function. It allows the programmer to provide an argument, which is the string to use for prompting the user. We can imagine that somewhere, some programmer has written a function definition for **input** that looks something like this:

```
def input(prompt):
    # Code that makes input work goes here.
```

Notice that the **input** function returns a value, which we then assign to a variable. In a function's body, we can tell Python what value to return using the **return** statement. Let's start with an overly simple example so that we understand the mechanics of the **return** keyword. Suppose we define the function shown in Listing 4.8.

Listing 4.8

```
1 def favorite_number():
2     return 42
3
4 number = favorite_number()
5 print(number)
```

Whenever a **return** statement is encountered in a function's body, the function stops immediately and returns to where the function was called. The value

that is returned from the function is whatever value was listed in the **return** statement. In Listing 4.8, there is only one statement in the body of the **favorite_number** function: the return statement itself.

To interpret the code in Listing 4.8, we start at line 4, which is the first line of the main program. The **favorite_number** function is called, so the program "jumps" up to **favorite_number**'s definition. The program enters the function's body and encounters the **return** statement. The **42** listed in the **return** statement causes the function to end, and the value **42** is sent back to line 4. Finally, line 4 can finish as the **42** is assigned to the variable **number**.

Now, let's see how a function might take a parameter value and then return something based on the parameter value. To experiment with this, we will define a function named **next_int** that takes an integer as a parameter. Whatever integer is given, the function will return the "next" integer. If we give **next_int** the value **3**, it will return **4**. Said another way, **next_int(3) == 4**. If we give **next_int** the value **-2**, it will return **-1**. Said another way, **next_int(-2) == -1**. We define **next_int** in Listing 4.9.

Listing 4.9: Function definition for **next_int**

```
1 def next_int(int_value):
2     next_value = int_value + 1
3     return next_value
```

Now, let's call our function and print its resulting value:

```
print(next_int(3))
print(next_int(-2))
```

which produces the following output:

```
4
-1
```

Notice in **next_int**'s function body how we create a new variable named **next_value**. **next_value** holds the value we will return from our function.

There is no rule that says we must return the value of a variable. Any expression that produces a value can follow the **`return`** keyword. In other words, we could have simply returned **`int_value + 1`**, as shown in Listing 4.10.

Listing 4.10: Function definition for **`next_int`**

```
1 def next_int(int_value):
2     return int_value + 1
```

Time to move on and look at a different example function. Consider Listing 4.11.

Listing 4.11: Function definition for **`piglatin`**

```
1 def piglatin(english):
2     if english[0] in "aeiou":
3         return english + "-way"
4     else:
5         return english[1:] + "-" + english[0] + "ay"
```

Whenever a **`return`** statement is encountered in a function's body, the function stops immediately and returns to where the function was called. The value that is returned from the function is whatever value is given to the return statement. Maybe you already understand this, but it bears repeating because it is very important to understand this concept if you are going to be able to understand functions.

In Listing 4.11, we have a function named **`piglatin`**. This example comes from the exercises in Section 3.7 from Chapter 3. We can pass to **`piglatin`** an English word, and for simplicity's sake, suppose the word always consists solely of lowercase letters. We are only handling two cases for now: 1.) the word starts with a vowel, or 2.) the word starts with a single consonant letter.

A function may have zero, one, or many return statements, but as soon as the program encounters a return statement, the function ends and the value is returned immediately.

How might we call this function? There are several ways we could do it.

```
engword = "apple"
pigword = piglatin(engword)
print("The pig latin of apple is %s." % pigword)

print("The pig latin of catalog is %s." % piglatin("catalog"))
```

Since **piglatin** returns a string, we can call **piglatin** anywhere we would normally put a string.

It's hard for new programmers to learn to write their own functions. It takes practice. It's helpful to see how to define a lot of example functions, and it's also helpful to try writing your own functions and seeing what problems you run into. Let's take a look at a bunch of example functions, and at some point, you should go back and try to define these functions on your own without looking at the code.

Listing 4.12 is a function definition that, given the radius of a circle, will return the area of the circle.

Listing 4.12: Function definition for **get_area**

```
1 import math
2
3 def get_area(radius):
4     return math.pi * radius * radius
```

Listing 4.13 is a function definition that simulates rolling a 6-sided die.

Listing 4.13: Function definition for **roll**

```
1 import random
2
3 def roll():
4     return random.randint(1, 6)
```

Listing 4.14 defines a function named **posintsum** that prints the sum of the first **n** positive integers. The listing also shows how you might call this function. For example, **posintsum(3)** would calculate the result of **1+2+3+4** as **10**. As another example, **posintsum(10)** would calculate the result of

1+2+3+4+5+6+7+8+9+10 as **55**. Note how we use a **for** loop and a variable named **total** to accomplish this before returning the result in **total**.

Listing 4.14: Function definition and sample function calls for **posintsum**

```
def posintsum(n):
    total = 0
    for i in range(1, n+1):
        total += i
    return total

print("1+2+3+4 = %d" % posintsum(4))
print("1+2+3+4+5+6+7+8+9+10 = %d" % posintsum(10))
```

What if we had forgotten the return statement at the end of the function? In other words, what if our function definition looked like Listing 4.15?

Listing 4.15: Forgotten **return** statement in **posintsum**

```
def posintsum(n):
    total = 0
    for i in range(1, n+1):
        total += i
    # Oh no! I need to remember to actually return total.
```

As it turns out, all functions in Python return something no matter what. If a function does not have a **return** statement, the function will return a special value known as **None**. The type of the value **None** is **NoneType**. You may have already seen **None** if you got confused on the difference between **print** and **input**, as many new programmers do. Suppose we did the following.

```
x = print("Enter a number: ")
# Oops.  x == None
```

Clearly, we meant to use **input** rather than **print**. **print** returns **None**.

4.4 Modularizing Code with Functions

Now that we can make "mini-programs" using functions, we can begin composing full programs made out of mini-programs. We will use functions to help organize our code so that our code is easier to read and easier to maintain. This will also make it easier to cope with errors, particularly user error, as we will see in the following example. Let's take an idea from the previous section and suppose that we want to write a simple program that asks the user for a positive integer **n**. Our program should print out the sum of all positive integers up through that **n**. We will also want to only accept valid input, so we will need to re-ask for input if what the user types is incorrect. Examples of bad input might be **23r5** (since there is a **r** in the number) or **-2** (since **-2** is negative).

It is often difficult for novice programmers to know where to begin in writing a program. One very effective approach is to start by writing comments in English that describe, step-by-step, what our program needs to do. So, we might do this.

```
# Ask for a positive whole number 'n' (i.e., a positive integer).

# Compute the sum of 1 + 2 + ... + n and print it.
```

That's a good start. Now, let's become "lazy" programmers. Laziness is a good thing in some situations, and this is one of those situations. In fact, a famous programmer named Larry Wall once stated that laziness is one of three "virtues" that programmers should seek (look it up on the Web).

In order to be "lazy," we should pretend that someone has already defined functions for us, and by calling these functions, our program will be done!

```
# Ask for a positive whole number 'n' (i.e., a positive integer).
n = getposint()

# Compute the sum of 1 + 2 + ... + n and print it.
total = getsum(n)

print("The sum of the first %d positive whole numbers is %d."
      % (n, total))
```

Hooray! We've written our program in three lines of actual code! Well, we haven't really, but now we can focus on the "pieces" of the program by defining the functions that will actually make this program work. Let's define **getposint** first (see Listing 4.16).

Listing 4.16: Function definition for **getposint**

```
1 def getposint():
2     s = input("Enter a positive whole number: ")
3     while not s.isdigit() and s != "0":
4         print("Sorry, that does not look like a positive whole
5                number.  Please try again.")
6         s = input("Enter a positive whole number: ")
7     return int(s)
```

Now, let's try to define **getsum** in Listing 4.17. Keep in mind that **getsum** has a single argument, so it needs a single parameter, too.

Listing 4.17: Function definition for **getsum**

```
1 def getsum(n):
2     total = 0
3     for i in range(1, n+1):
4         total += i
5     return total
```

Note that in line 3 of Listing 4.17 the second expression in **range** is **n+1**. Since we want to add up all the values from **1** to **n**, the value that will make us leave the look is one more than **n**, that is, **n+1**.

Writing functions takes practice. The more you try, the better you will get at it. A program with everything defined is shown in Listing 4.18.

Listing 4.18: Code composed with functions

```
1 def getposint():
2     s = input("Enter a positive whole number: ")
3     while not s.isdigit() and s != "0":
4         print("Sorry, that does not look like a positive whole
5                number.  Please try again.")
```

```
        s = input("Enter a positive whole number: ")
    return int(s)

def getsum(n):
    total = 0
    for i in range(1, n+1):
        total += i
    return total


# Ask for a positive whole number 'n' (i.e., a positive integer).
n = getposint()

# Compute the sum of 1 + 2 + ... + n and print it.
total = getsum(n)

print("The sum of the first %d positive whole numbers is %d."
      % (n, total))
```

Sometimes, programmers even like to put their main program code into its own function. That way, all of the code exists in some function. Listing 4.19 shows how we would do this.

Listing 4.19: Code composed with functions, using **`main`**

```
def getposint():
    s = input("Enter a positive whole number: ")
    while not s.isdigit() and s != "0":
        print("Sorry, that does not look like a positive whole
              number.  Please try again.")
        s = input("Enter a positive whole number: ")
    return int(s)

def getsum(n):
    total = 0
    for i in range(1, n+1):
        total += i
    return total

def main():
    # Ask for a positive whole number 'n' (i.e., a positive integer).
    n = getposint()

    # Compute the sum of 1 + 2 + ... + n and print it.
    total = getsum(n)

    print("The sum of the first %d positive whole numbers is %d."
```

```
23              % (n, total))
24
25  # Run the program by calling main().
26  main()
```

Some other programming languages like C++ and Java actually require programs to start by calling a function named **`main`**. If we always put our main program code into a function named **`main`**, it will ease our transition from Python to other languages.

4.5 Variable Scope

You may have noticed in the previous section that our programs are starting to get longer in terms of number of lines of code (LOC). As our programs get longer, we will start to define more variables to help our programs do their work. Eventually, we will find that sometimes we can't access a variable in one part of the program that is defined in another part of the program. This is due to *variable scope*. The *scope* of a variable is the part of the code where the variable can be accessed. In terms of accessing a variable, we need to make the distinction between *reading* a variable's value and *writing* to a variable's value. Reading a variable means using–but not modifying–its current value. Writing to a variable means modifying its value. Consider the following code.

```
y = x + 1
```

In this code, we are *reading* from **`x`**. We are *writing* to **`y`**. When we say *access* a variable's value, we mean the same thing as *reading* the variable.

Now, let's explore the notion of variable scope. Consider the (non-working) code in Listing 4.20.

Listing 4.20: Sum of squares, with **`NameError`**

```
1  for k in range(1, 11):
2      square = k * k
```

```
3     total = total + square
4
5 print(total)
```

The code in Listing 4.20 is supposed to compute the sum of squares, that is, $1^2 + 2^2 + 3^2 + \cdots + 10^2$. It computes each square's value and adds it to a variable named **total**. However, when we run the code, we get the following error.

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NameError: name 'total' is not defined
```

Let's take a closer look at line 3.

```
total = total + square
```

On the right-hand side (RHS) of the **=** in line 3, we are *reading* from both **total** and **square**, and then adding their values together. Since we did not previously define **total**, it is not yet *in scope*. The error produced in Python is called a **NameError** because Python doesn't have information about a variable named **total**. To correct this problem, we must *initialize* **total** with a starting value. Initializing a variable means to create the variable by giving it a starting value. Listing 4.21 shows the corrected code. Note that **total** is created prior to the start of the loop.

Listing 4.21: Sum of squares, corrected

```
1 total = 0
2 for k in range(1, 11):
3     square = k * k
4     total = total + square
5
6 print(total)
```

Always remember to initialize your variables.

Say it again: a variable's scope is the part of the program where the variable may be accessed (i.e., "read").

We are bringing up the topic of variable scope in this chapter because variable scope has particular relevance in functions. Suppose we were to write the code in Listing 4.22.

Listing 4.22

```
1  import math
2
3  def main():
4      people = 10
5      slices_per_pizza = 12
6      slices_per_person = 3
7      pizzas = number_of_pizzas()
8      print("You need to order %d pizzas." % pizzas)
9
10 def number_of_pizzas():
11     return math.ceil(people * slices_per_person / slices_per_pizza)
12
13 main()
```

Take a gander at Listing 4.22 and try to guess what might happen? Well, what do you think? Try to come up with a few possibilities. On one hand, it might set the variables **`people`**, **`slices_per_pizza`**, and **`slices_per_person`**, and then use them to calculate the number of pizzas one needs to order. Another possibility is that the program crashes hideously… but why?

Try running this code on your own. Of course, it ends up crashing. This code is a hot mess! The error reads as follows.

```
Traceback (most recent call last):
  File "test.py", line 13, in <module>
    main()
  File "test.py", line 7, in main
    pizzas = number_of_pizzas()
  File "test.py", line 11, in number_of_pizzas
    return math.ceil(people * slices_per_person / slices_per_pizza)
NameError: name 'people' is not defined
```

When the code reaches the **number_of_pizzas** function, it does not remember that there is a variable named **people** that was defined in the **main** function. That's interesting. This suggests that when we define a variable within a function's body, that variable is only visible within that function.

Think about it this way. Functions are like little houses. What happens in your house is your business. Other people shouldn't be able to see in your house from within their house.

So, if we define a variable in one function, it only "lives" in that function. This is a good thing! It means that if you define a variable somewhere in your program, a function you call can't mess with the variables you've already defined. If a function could mess with your variables without you being aware of it, then your program could have unintended side effects. If functions had surprising side effects, it would be very difficult to reason through programs and predict their behavior and output. One of the great things about computers is that they are logical and predictable.

A variable that is defined inside a function is called a *local variable*. If we define a variable in the body of a function, we say that the variable is *local* to that function. In this case, the variables **people**, **slices_per_pizza**, and **slices_per_person** are local variables in **main**.

All right, let's fix the problem. The best thing to do is to pass these values to the **number_of_pizzas** function, which means we'll need **number_of_pizzas** to have three parameters in its definition, as we can see in the updated code found in Listing 4.23.

Listing 4.23

```
import math

def main():
    pizzas = number_of_pizzas(10, 12, 3)
    print("You need to order %d pizzas." % pizzas)

def number_of_pizzas(people, slices_per_pizza, slices_per_person):
    return math.ceil(people * slices_per_person / slices_per_pizza)

main()
```

Hey, this works splendidly!

Let's go back to the "houses" metaphor we presented a few paragraphs ago. Recall that I asked you to think of functions as being like little houses. What do you think happens when you define a variable outside of all the "houses." In other words, what if I define a variable in what is normally the main part of the program that exists outside of any function's body? Consider Listing 4.24, where you'll notice we've done away with a **main** function entirely.

Listing 4.24

```
balance = 10000

def withdraw(amount):
    if balance >= amount:
        balance =  balance - amount

withdraw(1000)
print(balance)
```

If you run Listing 4.24, you'll see there is a problem. We'll resolve the problem shortly, however let's consider the intent of the code. The main program consists of three lines of code. They are (in order):

```
balance = 10000
withdraw(1000)
print(balance)
```

After we have defined **balance**, we define a function named **withdraw** that can be used to subtract an amount of money from **balance**, but only if there is sufficient money available in **balance**.

Since **balance** is not defined in the body of any function, it is known as a *global variable*. Global variables are accessible anywhere in the program, including inside functions. Consider the following code.

```
gl = 7

def f(x):
    return x + gl

print(f(3))
```

As we would expect, this code prints **10**.

Although global variables are accessible (i.e., they may be "read") from within functions, they cannot be modified from within a function unless we explicitly tell Python that's what we intend to do. Do you remember a little bit ago when I said that if function could mess with your variables without you being aware of it, then your program could have unintended side effects, and that side effects are bad? Well, here we are again. If you ran the code in Listing 4.24, you would see this:

```
Traceback (most recent call last):
  File "variable_scope.py", line 38, in <module>
    withdraw(1000)
  File "variable_scope.py", line 35, in withdraw
    if balance >= amount:
UnboundLocalError: local variable 'balance' referenced before assignment
```

Python assumes that any variable you wish to *write* to inside a function is a local variable, even if there is a global variable of the same name. Python does allow us to write to global variables, but we have to tell Python that we intend to do so and that we know the potential consequences! We can use the *global* statement to allow a function to write to a global variable; see Listing 4.25.

Listing 4.25

```
balance = 10000

def withdraw(amount):
    global balance
    if balance >= amount:
        balance =  balance - amount

withdraw(1000)
print(balance)
```

The output of Listing 4.25 is now **9000**, as we would expect.

global should be used sparingly. It is generally better to pass values to functions rather than storing those values in global variables, though in some cases we will manage common information in global variables, especially when we program video games. Stay tuned!

4.6 Exercises

1. Determine the output of the following code.

```
def f1():
    print("f1 called")

def f2():
    print("f2 called")

print("Main 1")
f2()
print("Main 2")
f1()
```

2. Determine the output of the following code.

```
def f1():
    print("f1 called")
    f2()

def f2():
    print("f2 called")


print("Main 1")
f2()
print("Main 2")
f1()
```

3. Determine the output of the following code.

```
def f1(x, y):
    z = x + y
    return z

print("Answer: %d" % f1(3, 2))
print("Answer: %d" % f1(2, 3))
```

4. Determine the output of the following code.

```
def f1(x, y):
    z = 2 * x + y
    return z

print("Answer: %d" % f1(3, 2))
print("Answer: %d" % f1(2, 3))
```

5. Determine the output of the following code.

```
def umkay(message):
    return message + ", umkay?"

print(umkay("Do your homework"))
```

6. Define a function named **`circum`** that takes the radius of a circle as a parameter and returns the circumference of that circle.

7. Define a function named **`smallest`** that takes three float parameters and returns the smallest value of the three.

8. Define a function named **`find_vowel`** that takes a string as a parameter and returns the index of the first vowel in the string.

9. Define a function named **`count_words`** that takes a string as a parameter and returns the number of words in the string (where words are separated by a single space).

Chapter 5

Lists

5.1 List Operations

Yay! Finally, we get to talk about *lists*. Lists are very useful and powerful structures in Python, and their use is critical when we start working with real-world data. So, let's get cracking!

Let's motivate the need for lists. Suppose I wanted to write software that keeps track of students' names and test scores. Let's store their names first. Given what we know so far, we would need a separate variable for each student name.

```
1 student1 = "John Smith"
2 student2 = "Judy Adams"
3 student3 = "Frank Johnson"
```

If we had another student enroll in the class, we'd need to add another variable.

```
1 student1 = "John Smith"
2 student2 = "Judy Adams"
3 student3 = "Frank Johnson"
4 student4 = "Paige McConnell"
```

How many variables do we need? Well, I guess we'd need as many as we think we might possibly need. If we were using this software at a large public

university, we might have 500 students in a Computer Science 1 class, so our code might look like this:

```
student1 = "John Smith"
student2 = "Judy Adams"
student3 = "Frank Johnson"
# ...
# ...
# ... definitions for student4 through student498 omitted ...
# ...
# ...
student499 = "Sally Schmidt"
student500 = "Maxwell Masterson"
```

That's a lot of variables, and this is getting ridiculous.

What if instead of a whole bunch of variables to store all of our students, we could have just one variable to store all of the students. We can! Contrast the code we've presented so far with the code in Listing 5.1.

Listing 5.1

```
students = ["John Smith", "Judy Adams", "Frank Johnson", "Paige McConnell"]
```

The **`students`** variable holds a *list* of values. In this case, each value is a single string value, and there are four strings stored in this list. We use the square bracket symbols **`[`** and **`]`** when we want to create a list.

In Listing 5.1, whenever we want to store more students, we don't need to create more variables. We can just dump their names into this **`students`** variable. Cool, huh? Actually, we haven't even scratched the surface of how useful this is going to be.

So, what can we do with a list once we've made one? Table 5.1 shows several handy list expressions and list functions. To make sense of the table, suppose we have the following variables.

```
Let L be a list,
    L1 be a sub-list of L,
    v be a value we wish to store in a list,
```

Table 5.1: List Operations

Expression	Explanation
`v = L[i]`	The indexing operator: gets the i^{th} item from `L` and places it in the variable `v`.
`L1 = L[i:j]`	The slicing operator: makes a new list `L1`, which is a sub-list of `L` consisting of `L`'s ite
`L.append(v)`	Adds the value `v` to the end of the list `L`.
`L.insert(i, v)`	Inserts the value `v` into the list `L` at position `i`.
`L.remove(v)`	Removes the first instance of the value `v` in the list `L`. If there are more instances of `v` i
`v = L.pop(i)`	Removes the value in `L` at index `i` and places it in the variable `v`. `i` may be omitted, in
`L = s.split()`	Uses the parts of a string to make a list. Does so by splitting a string up into pieces sep
`L = s.split(p)`	Uses the parts of a string to make a list. Does so by splitting a string up into pieces sep
`s = p.join(L)`	Joins the items in the list `L` to form a new string (stored in `s`) by joining the items toge

```
    i be an index integer,
    j be another index integer,
    s be a string
    p be a string
```

You'll notice in Table 5.1 that some of the same operations that can be performed on strings can also be performed on lists.

The more you program, the easier it will become to read tables like Table 5.1. For most beginning programmers, however, it is easier to learn by looking at examples. Listing 5.2 demonstrates a few "toy" examples of how to use these list operations. The result of most of the operations is printed, and the effect of each statement is shown in a comment to the right of the statement.

Listing 5.2

```
cheeses = ["cheddar", "swiss", "feta", "gouda", "parmesan"]
line = "red,green,blue,teal,cyan"

print(cheeses[1])           # prints swiss
print(cheeses[0:2])         # prints ["cheddar", "swiss"]
print(cheeses[-1])          # prints parmesan
print(cheeses[3:])          # prints ["gouda", "parmesan"]
print(cheeses[:2])          # prints ["cheddar", "swiss"]

cheeses.remove("parmesan")  # cheeses is now ["cheddar", "swiss", "feta", "gouda"]
cheeses.pop(-1)             # cheeses is now ["cheddar", "swiss", "feta"]
```

```
cheeses.pop()                # cheeses is now ["cheddar", "swiss"]
cheeses.append("jack")       # cheeses is now ["cheddar", "swiss", "jack"]
cheeses.insert(0, "brie")    # cheeses is now ["brie", "cheddar", "swiss", "jack"]

colors = line.split(",")     # colors is now ["red","green","blue","teal","cyan"]
print(len(colors))           # prints 5
print(", ".join(colors))     # prints red, green, blue, teal, cyan
print(" -> ".join(colors))   # prints red -> green -> blue -> teal -> cyan
```

Okay, let's test our understanding by trying to apply what we've learned. Suppose we have a list named **`lst`** defined as:

```
lst = ["Maria", "Sheryl", "Barbara", "Shafi"]
```

Can we write code to modify **`lst`** so that the first item is moved to the end of the list? In other words, the end result should look like this:

```
lst = ["Sheryl", "Barbara", "Shafi", "Maria"]
```

We can use any number of statements that we want. Try it. Try mocking up a solution using comments first, and then write your code beneath each of the comments.

Here's one approach. First, let's start with comments.

```
# Remove the first item and store it.

# Put the stored item on the end of the list.
```

Now, let's fill in the details of how we accomplish these steps.

```
# Remove the first item and store it.
value = lst.pop(0)

# Put the stored item on the end of the list.
lst.append(value)
```

Good! This is a suitable way to accomplish this task, and more importantly, it is easy to read what the code does even if there are no comments to explain the statements. To see what I mean, let's strip away the comments.

```
value = lst.pop(0)
lst.append(value)
```

We can clearly see that the first statement extracts the first item, and the second statement puts it back on the end of the list. We should always try to write our code in such a way that it is easily read and understood. This was best articulated in the preface of the book *Structure and Interpretation of Computer Programs* (a.k.a. "SICP") by Harold Abelson and Gerald Sussman:

> "Programs must be written for people to read, and only incidentally for machines to execute."

We could have also written this code as a one-liner, like this:

```
lst.append(lst.pop(0))
```

One could argue this is harder to read, at least initially.

Let's try another example. What if we wanted to "swap" the locations of the first and second items in the list? Consider the following definition of **`lst`**.

```
lst = ["Maria", "Sheryl", "Barbara", "Shafi"]
```

Can we write code that makes **`lst`** look like this?

```
lst = ["Sheryl", "Maria", "Barbara", "Shafi"]
```

Try to write code that does this. Make a good, honest attempt before reading ahead.

We know that we want the value of the expression **`lst[0]`** to be changed to the value of the expression **`lst[1]`** and vice versa. So, our first attempt might be Listing 5.3.

Listing 5.3: First attempt at swapping value in a list

```
lst[0] = lst[1]
lst[1] = lst[0]
```

Consider the effect of the first statement **`lst[0] = lst[1]`**, which is shown in Figure 5.1.

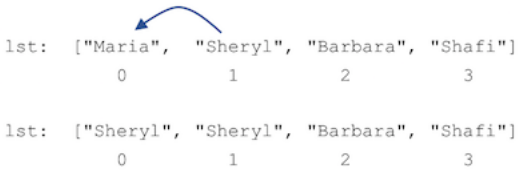


Figure 5.1

Uh oh! Our first statement overwrites "Maria" and she disappears forever! That is not good at all.

What we need to do is save "Maria" somewhere first so she doesn't get obliterated. So let's create a *temporary variable* – a one-use variable that will help us accomplish a task, and then we'll likely never use it again. Think of a temporary variable (or just "temp variable" for short) like a Post-It note. We may write something on a Post-It note to remind ourselves of something for a brief period of time. Consider our next attempt in Listing 5.4.

Listing 5.4: Swapping values in a list

```
temp = lst[0]
lst[0] = lst[1]
lst[1] = temp
```

Let's visualize how this works (see Figure 5.2).

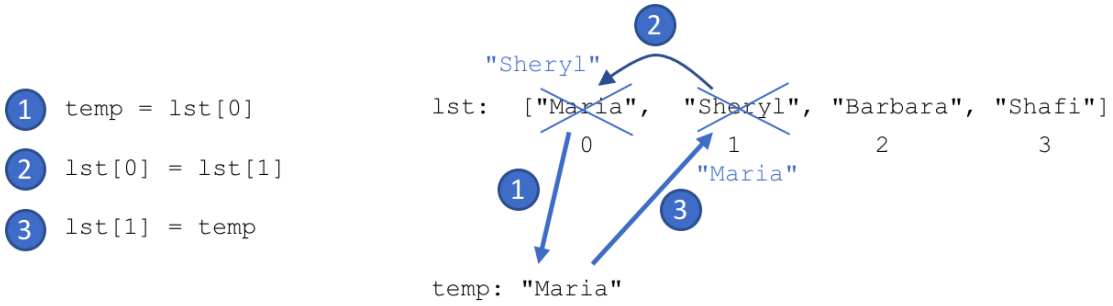


Figure 5.2

The statement labeled number **1** copies the value at index **0** (in this case, **"Maria"**) into the **temp** variable for safe keeping. Then, statement number **2** writes the value at index **1** over the top of the value at index **0**. At this point (after executing statement number **2**), there are two "Sheryl" strings in the list, but that's okay because we have saved **"Maria"** in our temp variable. Finally, in statement number **3**, we take the value saved in **temp** and we overwrite the value in index **1** with **temp**'s value. The list contents are now correctly **["Sheryl", "Maria", "Barbara", "Shafi"]**.

Index **0** and index **1** are hardcoded in Listing 5.4. If we wanted, we could invent two variables **i** and **j** to hold the indices whose values we wish to swap (see Listing 5.5).

Listing 5.5: Swapping values in a list, no hard-coded indices

```
i = 0  # or, whatever index we wish
j = 1  # or, whatever other index we wish
temp = lst[i]
lst[i] = lst[j]
lst[j] = temp
```

5.2 Assignment Statements Revisited: Multiple Assignment

Since this book is purely an introduction to programming more so than a definitive guide to the nuances of Python, we could stop our discussion right here. This same programming technique will work with most other programming language as you encounter them (e.g., C++, Java, etc.). However, Python has a really cool way of performing swaps called *multiple assignment* that your author (that's me!) simply can't help himself from showing you! We can replace the code in Listing 5.5 with Listing 5.6.

Listing 5.6: Swapping values in a list, no hard-coded indices

```
i = 0  # or, whatever index we wish
j = 1  # or, whatever other index we wish
lst[i], lst[j] = lst[j], lst[i]
```

Wizardry!
How does this work? Let's approach it visually as in Figure 5.3.

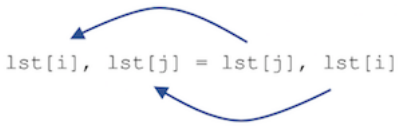


Figure 5.3

Multiple assignment is a Python feature known in the computer science community as *syntactic sugar*. Syntax is how you arrange a statement so that it can be understood. In the English language, sentences have a syntax consisting of a noun phrase followed by a verb phrase. That's how we start *parsing* (or "breaking apart and making sense of") a sentence. "Sugar" in this case means non-essential. So, syntactic sugar refers to language features that make something easier or more expressive but are non-essential. Usually, syntactic sugar

is converted behind the scenes to another form through a process called *desugaring*. If you continue on in studying computer science, this won't be the last time you hear about desugaring, hopefully.

Desugaring in this case first evaluates the expressions on the RHS, and then converts this

```
lst[i], lst[j] = "Sheryl", "Maria"
```

behind the scenes to this

```
lst[i] = "Sheryl"
lst[j] = "Maria"
```

5.3 List Variables: References Versus Values

Here is some fairly simple code that creates variables, changes one of them, and then prints their values.

Listing 5.7: Simple Assignment Statements

```
x = 3
y = x
x = x + 1
print(x)   # prints 4
print(y)   # prints 3
```

Let's do something similar, only this time we'll use lists of numbers.

Listing 5.8: Assignment with List Variables

```
xlist = [2, 4, 6]
ylist = xlist
xlist.append(8)
print(xlist)
print(ylist)
```

You may have expected the first print statement to yield `[2, 4, 6, 8]` and the second one to yield `[2, 4, 6]`, but that isn't what happens at all! Instead, the output is:

```
[2, 4, 6, 8]
[2, 4, 6, 8]
```

How is it that changing `xlist` has the side-effect of changing `ylist`? What is this sorcery?!

The good news is that the reason is really fairly simple, especially when we explain it with pictures. Consider the first example in Listing 5.7 where we had integer variables `x` and `y` (see picture in Figure 5.4).

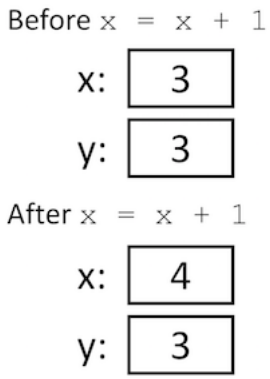


Figure 5.4

No surprises there. Now, let's see what happens in the example with `xlist` and `ylist`. It turns out that list variables don't actually contain lists themselves. Instead, they contain something called a *reference* to a list. The list is actually stored somewhere else. Think of a reference as being like a "pointer" or an arrow to somewhere else in the computer's memory. With this in mind, we can represent the code in Listing 5.8 visually in Figure 5.5.

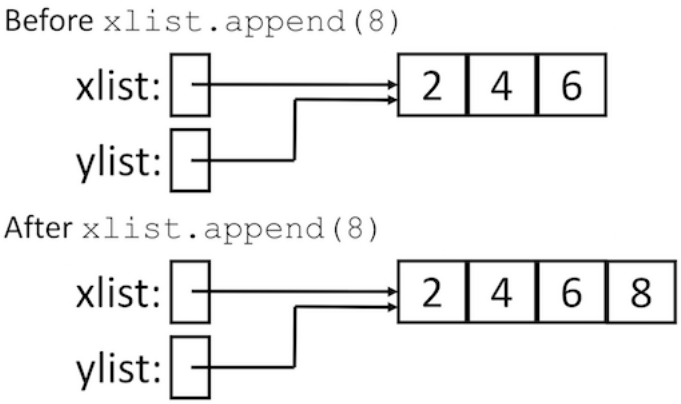


Figure 5.5

If we attempt to modify **xlist** or **ylist**, either will affect the same list. You might ask why this happens in Python, and there is a perfectly good reason that we will explore in Section 5.4. For now, recognize that we want to know how to make **ylist** be a copy of **xlist**. We could do the following.

```
ylist = xlist[:]    # the right way to make a copy of a list
```

This code assigns a slice of **xlist** to the new variable **ylist**. Why no integers before or after the colon (**:**)? Recall that the integers around the colon are optional. If no integers are given, it selects the entire list. The end result is shown in Figure 5.6.

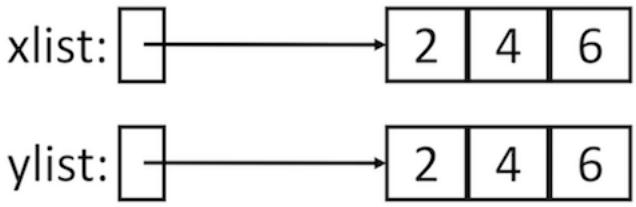


Figure 5.6

Now if we perform **xlist.append(8)** this time, we end up with the result in Figure 5.7.

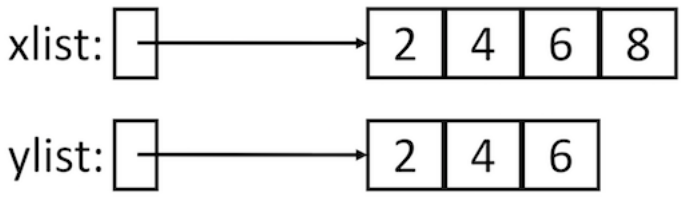


Figure 5.7

5.4 Functions with Lists

In the previous section, we learned that list variables do not contain the lists themselves. Rather, they contain a *reference* to a list. There are a number of perfectly good reasons for this, the first of which has to do with how functions interact with lists.

Let's consider an example. Suppose we have a list of strings that happen to be integers, like `["1", "2", "3", "4"]`. We wish to change the list so that the values are actually integers rather than strings, that is, `[1, 2, 3, 4]`. We would need to pass the list to the function as an argument.

```
def make_int_list(lst):
    # function body goes here


# main section of code
numbers = [ """Somehow we get a bunch of strings here.""" ]
make_int_list(numbers)
# Now 'numbers' has been changed from a list of strings
# to a list of integers.
```

We have named the parameter `lst`, but you can name it whatever seems appropriate. We named it `lst` rather than `list` since the word `list` is already used in Python as the name of a type.

As we learned in Section 4.2, Python passes arguments to function parameters by value. That is, a copy of the value in the main program is sent to the function. Imagine if list variables contained the lists themselves. In the above

example, a copy of the numbers list would be sent to **make_int_list**, so **lst** would be a complete copy of numbers (see Figure 5.8).

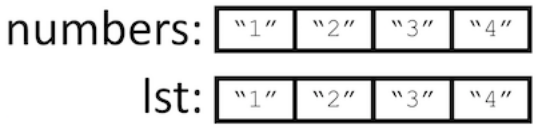


Figure 5.8

If we program properly, **make_int_list** would convert all the strings in the list to integers so the end result would be as shown in Figure 5.9.

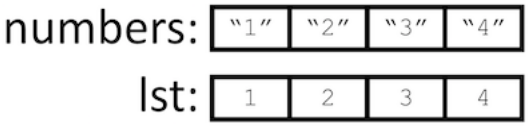


Figure 5.9

But, the original list that we wanted to change, namely **numbers**, hasn't changed! Now, consider what happens since list variables contain references to lists. The reference is passed by value, so **lst** now "points" at the same list as **numbers** (see Figure 5.10).

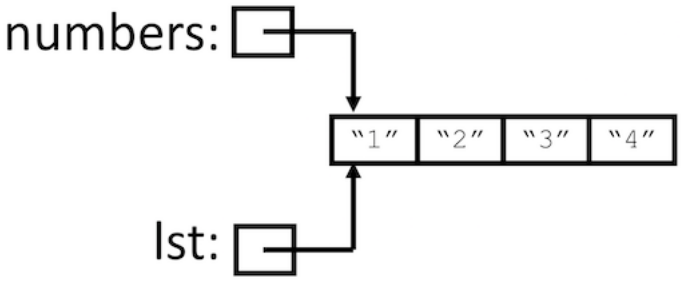


Figure 5.10

Now when **make_int_list** has returned, all changes made to **lst** are also made to **numbers** because both **lst** and **numbers** refer to the same list (see Figure 5.11)!

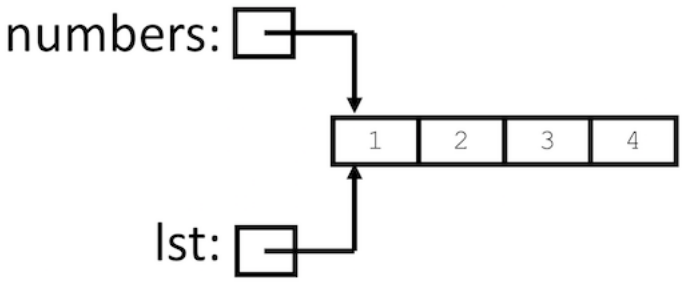


Figure 5.11

We should probably go ahead and write the function's body. We use a loop to walk through each index. At each index, we can convert the string to an **int**.

```
def make_int_list(lst):
    for i in range(0, len(lst)):
        lst[i] = int(lst[i])
```

That's it!

There is another advantage to passing a list to a function given that lists are stored by reference rather than by value. Suppose we had a very large list of information (say, 500 million strings). If the entire list was passed to the function rather than just a reference to the list, Python would have to make a copy of the entire list, all 500 million items. That takes precious time, and even though computers are fast, there are limits even to what computers can do (more on this in a later chapter). It also uses an unnecessary amount of memory, because now instead of having one list containing 500 million strings, we now have two lists containing a cumulative total of 1 billion strings. There are some computer languages that let you pick whether you pass arguments to functions by value or by reference no matter what the type of the variable is. It is common practice in those languages to pass large lists by reference for that reason alone.

Let's look at another way we might use functions on lists. We won't always define functions that modify lists directly. Sometimes, we will pass a list to a function, and then the function will *return* a value or an entirely new list based on the original list. For example, there is a function named **sum** that takes a list of numbers and returns the sum of all the numbers in the list. We could use it like this.

```
money = [30.50, 72.25, 10.00]
grand_total = sum(money)
print("You have $%.2f." % grand_total)
```

sum takes a list as a parameter and returns a **float**. It does not modify the list in any way. How did the person who wrote the **sum** function do it? Let's find out by writing our own, only we need to give ours a different name. Let's call it **total**. Once we define our function named **total**, we will be able to do this.

```
money = [30.50, 72.25, 10.00]
grand_total = total(money)
print("You have $%.2f." % grand_total)
```

Let's begin. We know that we need the name of the function to be **total**, and it needs to accept a single list parameter, so we can write:

```
def total(numbers):
    # function body goes here
```

Now, the function body will need to loop through the list of numbers and add each of them to a variable. The variable will keep track of the running total. So, let's create the variable (we'll call it **running_total**) and then write the loop.

Listing 5.9: Function definition for `total`

```
def total(numbers):
    running_total = 0.0
    for i in range(0, len(numbers)):
        running_total += numbers[i]
    return running_total
```

This should work. Go ahead and test it out. The key ideas here are:

1. We're passing a list to a function.
2. We're using a loop to walk through the list's items.
3. We're creating a variable that we will update in each iteration of the loop.

These key ideas will show up a lot when we write functions that operate on lists.

Let's try out a different example, and let's see if we can apply some of the same ideas mentioned in the prior paragraph. Suppose we have a list of numbers. We want to know the smallest value in that list of numbers. Let's write function that determines this for us.

First, let's pick a name for our function: `smallest`.

Next, let's write the function signature.

```
def smallest(numbers):
    # FIXME
```

Notice how I have a comment with the all-caps word `FIXME` in place of the function's body? This is a common idiom that programmers use to remind them that they need to add code to a particular part of a file. I suppose it's silly to do this since we're going to replace the function body very shortly, but it's worth bringing up since 1.) many programmers do this, and 2.) many programs for editing code will keep track of your `FIXME`'s and then help you find them before you test out your code.

Okay, now let's use a loop to walk through the list's items as we described above. We'll use a variable to keep track of the smallest value we've seen so far with each step of the loop.

Listing 5.10: Function definition for `smallest`, first attempt

```
1 def smallest(numbers):
2     small = 0.0
3     for i in range(0, len(numbers)):
4         if numbers[i] < small:
5             small = numbers[i]
6     return small
```

It's kind of crazy how similar `smallest` is to `total`, isn't it? In computer science, we take a whole bunch of problems (many of them very complicated) and rearrange them so we can apply simple solutions to them. That's one of the cool things about computer science; it teaches us to define and organize our problems so that we can manage the complexity of the problem. Yay computer science!

Okay, let's not pat ourselves on the back too much here. We haven't tested this code yet. Let's try it out.

```
print(smallest([5, -5, 2, 3]))
print(smallest([10, 30, 20]))
```

The output of these tests is:

```
-5
0.0
```

The first line is correct. The second line is BOGUS! What went wrong? Can you figure it out on your own? (Sure you can. Take a deep breath and debug it.)

The problem is on line 2 of Listing 5.10. We needed to give the variable `small` a good first value, but we weren't sure what a good first value was at the time. We picked `0.0`. Unfortunately, nothing in `numbers` is going to be

smaller than `0.0` when all the numbers are greater than zero, so `smallest` returns `0.0` even though there is no zero in `numbers`.

So, what would be a better starting choice for `small` than zero? Some students might say, "How about a really big number!" That's not a bad idea. We could do something like Listing 5.11.

Listing 5.11: Function definition for `smallest`, second attempt

```
def smallest(numbers):
    small = 1000000000.0
    for i in range(0, len(numbers)):
        if numbers[i] < small:
            small = numbers[i]
    return small
```

This will work really well… until we have all numbers that are bigger than 1 billion.

Let's try one more strategy. Let's let the first smallest number be the first number in the list. Suppose `numbers[0]` is `10.0`. Then, the first value of `small` can just be `10.0`. This should work nicely. If the first number in the list is the smallest, `small` will stay set to that number. If there is a smaller number down the line in the list, eventually it will get changed. This should work, and our final function definition for `smallest` is shown in Listing 5.12. Naturally, we'll want to test it, however.

Listing 5.12: Function definition for `smallest`, corrected

```
def smallest(numbers):
    small = numbers[0]
    for i in range(0, len(numbers)):
        if numbers[i] < small:
            small = numbers[i]
    return small
```

This final code works… as long as there is at least one item in the list. If the list is empty, then calling `smallest` doesn't make much sense in the first place. After all, a list with no items does not have a smallest item, per se.

Are these examples starting to help? How about if we do a few more examples?

Here's a different one. Suppose we have a list of student exam percentages. Such a list might look like this (we'll call it **scores**):

```
scores = [82.5, 77.0, 95.0, 56.0, 74.5, 46.0, 97.5]
```

In this example, there are seven student scores. We could think of other examples where there could be far more scores. Suppose we're interested in the number of students who passed the exam, i.e., the number whose score is **60** or above. Let us define a function named **number_passed** that returns the number of passing students.

First, write the function signature (Listing 5.13).

Listing 5.13: Function signature for **number_passed**

```
def number_passed(scores):
    # FIXME
    pass
```

I've written **pass** for the body of the function. We'll delete it very shortly, but I thought now would be a good time to introduce **pass**. What does **pass** do exactly? Python expects there to be an indented block of statements after a function signature. If there is no indented statement or block of statements, our code will crash. Comments don't count, so naturally **FIXME** comments don't count either. This way I can still run any other code in my file even if I haven't finished the definition of **number_passed**.

Okay, let's delete the **pass** statement and the **FIXME** comment and replace them with an actual function body. What are we trying to accomplish again? We want to scan the list and keep track of how many students passed. Any time we need to scan a list, we should immediately think of using a loop. We've used **for** loops mostly, but we can use **while** loops just as easily. It's up to you as the programmer to express your code however you want. Let's write some code that scans through the list. Listing 5.14 shows a loop with a lot of pseudocode surrounding it.

Listing 5.14: Partial function definition for **number_passed**, attempt 1

```
def number_passed(scores):
    # Probably use a variable to keep track of something.
    for i in range(0, len(scores)):
        # Do something with scores[i]
    # Return a thing.
```

This function definition follows the same pattern as **total** and **smallest**. Since we wish to keep track of the number of students who've passed, we'll use a variable and that variable's value will be what this function returns. Let's update those lines (Listing 5.15).

Listing 5.15: Partial function definition for **number_passed**, attempt 2

```
1 def number_passed(scores):
2     passed = 0
3     for i in range(0, len(scores)):
4         # Do something with scores[i]
5     return passed
```

Notice how we're writing the code non-linearly. Often, it's best to write what you can at the time, use comments when you're not ready to fill in the details, and then go back later and fill in the details.

Lastly, we need to increment **passed** whenever we encounter a passing score, i.e., **scores[i]**. Checking **scores[i]** involves an **if** statement (Listing5.16).

Listing 5.16: Function definition for **number_passed**

```
1 def number_passed(scores):
2     passed = 0
3     for i in range(0, len(scores)):
4         if scores[i] >= 60.0:
5             passed += 1
6     return passed
```

This looks simply grand! Now, you test it!

Write some test cases. Use the examples of test code that we're written in previous examples. Go ahead. I'll wait. I could use a break from writing this book. Writing a book is hard work, you know.

Okay, I'm back. Did you come up with something like this?

```
print(number_passed([5, 100, 2, 70]))    # should print 2
print(number_passed([90, 80, 70, 60]))   # should print 4
print(number_passed([1, 2, 3]))          # should print 0
print(number_passed([]))                 # should print 0
```

5.5 Using Several Lists to Store Related Data

We've been using lists thus far to store groups of related information. In Section 5.4, for example, we kept a series of exam scores in a list variable we named **scores**. We might store a series of names in a list called **names**.

But, what if we want to keep track of both names and scores. That is, we want to associate students' names with their scores. In this way, we can know that Michelle got a 90% while Susan got a 76%. There are a number of ways to accomplish this, and in fact we will explore some rather slick ways to do this in Chapters 7 and 9, respectively. However, there is a very good way to do this by simply using more than one list. Observe Listing 5.17.

Listing 5.17: Using several lists to store related data

```
1 names = ["Aubrey Bell", "Adam Jamison", "Susan Stratford", "Michelle Wang"]
2 scores = [82.5, 87.0, 76.0, 90.0]
```

Notice that **names** and **scores** contain the same number of items. Let us assume that the positions of each name matches the position of the score in each list. For example, **"Aubrey Bell"** got an **82.5** since both values exist at index **0** in both lists. Likewise, **Susan Stratford** got a **76.0** because they are both at index **2** in both lists.

In the example found in Listing 5.17, we are using different lists to store different attributes of people. Each list has the same number of items. One

attribute is the person's name, which is stored in its own list. Another attribute is a person's score, which is stored in its own list as well. Each person is uniquely identified by an index. Figure 5.12 shows this relationship appears visually, using indices to associate names and scores. (Note: not all items and indices are shown in the figure to keep the image smaller).

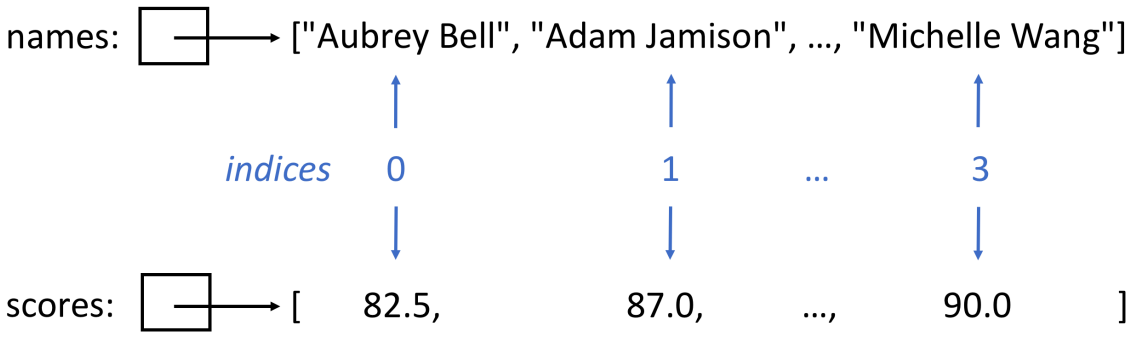


Figure 5.12

The definitions in Listing 5.17 allows us to write code like the following (see Listing 5.18).

Listing 5.18: Iterating through related lists

```
for index in range(0, len(names)):
    print("The score for %s is %.2f%%." % (names[index], scores[index]))
```

The output of the example code in Listing 5.18 is:

```
The score for Aubrey Bell is 82.50%.
The score for Adam Jamison is 87.00%.
The score for Susan Stratford is 76.00%.
The score for Michelle Wang is 90.00%.
```

5.6 Coding Conventions for Lists Spanning Multiple Lines

Sometimes the lists we define will have a whole lot of items, and it will be cumbersome to manage their contents on a single line. We can "spread out" the definition of such a list using the convention shown in Listing 5.19.

Listing 5.19: List definition spanning multiple lines

```
cheeses = [
    "cheddar",
    "provolone",
    "colby jack",
    "gorgonzola",
    "brief",
    "pepper jack"
]
```

This is the preferred way to format a list that spans multiple lines as defined by the PEP8 coding standard. PEP8 is a document available on the Web that tells Python programmers how best to format their code so that any programmer can more easily read and modify code written by others. The PEP8 document tells us this is one of the two ways that are best for defining multi-line lists. This particular way is the one we will use through the remainder of this book.

Note that we are still using commas to separate the items in the list. If we wanted to add another cheese to the end of this list, we would need to add a comma after **`"pepper jack"`** and then add the last cheese, like this:

```
cheeses = [
    "cheddar",
    "provolone",
    "colby jack",
    "gorgonzola",
    "brief",
    "pepper jack",
    "mozzarella"
]
```

This is cumbersome, so the people who invented Python decided we should be able to leave a trailing comma at the end of any list (either single line or multiple line) in case we want to add more values. This allows us to change the look of our definition to Listing 5.20 (note line 7).

Listing 5.20: List definition spanning multiple lines

```
1 cheeses = [
2     "cheddar",
3     "provolone",
4     "colby jack",
5     "gorgonzola",
6     "brief",
7     "pepper jack",
8 ]
```

Now if we want to add `"mozzarella"`, we can just hit the **Enter** or **Return** key and start typing. This is handy if we want to copy and paste a bunch of lines.

```
1 cheeses = [
2     "cheddar",
3     "provolone",
4     "colby jack",
5     "gorgonzola",
6     "brief",
7     "pepper jack",
8     "mozzarella",
9 ]
```

5.7 Lists of Lists

We've been using lists to store a series of items in a row. Each of these items can have any type: string, integer, float, … or even another list! This might sound kind of weird and scary at first, but you may be surprised to know that a list containing lists is nothing more than a matrix, a grid, a table, a spreadsheet, or whatever other term you might use to describe rows and columns of data.

Watch this:

```
votes = [
    [25, 18,  2],
    [19, 17,  1],
    [25,  4,  5],
    [ 5, 27, 20],
    [40, 30, 29],
]
```

Hey, it's a grid! Let's try to type some expressions using **votes** to see what we can do with it.

```
print(votes[0])      # prints [25, 18, 2]
print(votes[-1])     # prints [40, 30, 29]
print(votes[0][1])   # prints 18
print(votes[1][0])   # prints 19
print(votes[2][2])   # prints 5
print(votes[20][20]) # Kablooey!  IndexError!  The max index of votes is 4.
```

How does an expression like **votes[0][1]** work? Well, the first part of the expression **votes[0]** gives us the list **[25, 18, 2]**. Essentially, **votes[0][1]** becomes **[25, 18, 2][1]**. Then, the **[1]** gives us the value of the list at index **1**, so **[25, 18, 2][1]** becomes **18**.

When we use two sets of square brackets, the first square bracketed value selects the row number and the second bracketed value selects the column, as in **votes[row][column]**.

Now, let's see a practical example. I've named this list of lists **votes** for a reason. Suppose we have an election and we want to keep track of the votes for each candidate. Suppose we also want to be more detailed in tracking votes, so we want not just total votes but votes by county in a particular state. Let's let the columns represent the candidates. Since we have three columns, that must mean we have three candidates! Then, let's let the rows represent the counties. We have five rows, so we should assume we have five counties we are tracking.

So, let's see who won the election. No Electoral College here folks; we'll count raw votes. Let's total them up. Only, let's do it like real programmers. We could write **total0 = votes[0][0] + votes[1][0] + votes[2][0] + votes[3][0] + votes[4][0]**, but that's really lame and we'd have to

change our totaling code if we ever add more counties. So, let's do this instead (Listing 5.21).

Listing 5.21: Total votes for one candidate

```
total = 0
for countyindex in range(0, len(votes)):
    total = total + votes[countyindex][0]

print("Total votes for candidate 0: %d" % total)
```

Then, we could copy and paste this code for candidates **1** and **2**, or we could put all of this code in another loop and "loop through" the candidates (Listing 5.22).

Listing 5.22: Total votes for all candidates

```
for candindex in range(0, len(votes[0])):
    total = 0
    for countyindex in range(0, len(votes)):
        total = total + votes[countyindex][candindex]

    print("Total votes for candidate %d: %d" % (candindex, total))
```

Let's stop referring to the candidates as "Candidate 0", "Candidate 1", etc. Let's give them actual names and store them in code somewhere. How about a using a list?

Listing 5.23: A list for candidates

```
candidates = ["Ronald Rump", "Billary Blimpton", "A Giant Meteor"]
```

Similarities of these names to candidates in recent U.S. presidential elections is purely coincidental… or not.

Now, check this out. See how we can use a loop with this new **candidates** list to label the columns of our **votes** grid when totaling up the results (see Listing 5.24).

Listing 5.24: Total votes for all candidates, with names

```
votes = [
    [25, 18,  2],
    [19, 17,  1],
    [25,  4,  5],
    [ 5, 27, 20],
    [40, 30, 29],
]
candidates = ["Ronald Rump", "Billary Blimpton", "A Giant Meteor"]
for candindex in range(0, len(votes[0])):
    total = 0
    for countyindex in range(0, len(votes)):
        total = total + votes[countyindex][candindex]

    print("Total votes for %s: %d" % (candidates[candindex], total))
```

The output of Listing 5.24 is:

```
Total votes for Ronald Rump: 114
Total votes for Billary Blimpton: 96
Total votes for A Giant Meteor: 57
```

This code is pretty slick. This code is also a good example of where we should be heading as programmers. We should be getting pretty comfortable with iterating through a list using a loop.

What makes this code slick? Well, if we change the structure of the **votes** grid, the code still works perfectly. In other words, we could add or subtract candidates, or we could add or subtract counties, and the code still works because we have not hard-coded the number of candidates or counties.

To verify this, let's redefine **votes** (and **candidates**, so that its length matches the number of columns in **votes**) and then try to run our code again. A new, full listing can be found in Listing 5.25.

Listing 5.25: Total votes for all candidates, again

```
votes = [
    [25, 18,  4,  2],
    [19, 17,  8,  1],
    [25,  4,  1,  5],
```

```
    [ 5, 27, 21, 20],
    [40, 30,  5, 29],
    [10, 12,  1,  1],
]
candidates = ["Ronald Rump", "Billary Blimpton",
              "Johnny Third-Party", "A Giant Meteor"]
for candindex in range(0, len(votes[0])):
    total = 0
    for countyindex in range(0, len(votes)):
        total = total + votes[countyindex][candindex]

    print("Total votes for %s: %d" % (candidates[candindex], total))
```

Because we have not hard-coded the lengths of the lists, and because the number of **votes** columns matches the length of **candidates**, the code works as shown below.

```
Total votes for Ronald Rump: 124
Total votes for Billary Blimpton: 108
Total votes for Johnny Third-Party: 40
Total votes for A Giant Meteor: 58
```

Poor Johnny Third-Party – the Giant Meteor got more votes than he did. Maybe Johnny should re-think his political career aspirations if he can't manage to beat an eschatological, extra-terrestrial projectile of doom!

Spend a little time thinking about how you might write code to do different things with **votes** and **candidates**. How many different ways can we slice-and-dice this data? In Section 5.10 (the Chapter 5 Exercises), you'll be asked to report the winner of each county, and you'll need to give each county a name. Be thinking about how you might accomplish that.

5.8 (Optional) Functional Programming with Lists

It's not uncommon to have a list that contains items that we wish to modify. Or, we may have a list, and we wish to create a new list based on the original list. You may recall an exercise in a previous chapter where we were asked to create a program that translated English into Pig Latin. If you do not recall this

exercise, it is sufficient to understand that an English word can be translated into the fictional "Pig Latin" language by taking the first consonant in the word, moving it to the end of the word, and then adding "ay" to the end. Thus, "cat" becomes "at-cay." There are other pertinent rules for forming Pig Latin words, but for the example that follows, this understanding is sufficient.

If we take a string containing an English sentence and we transform it into a list of words, creating a program that translates English to Pig Latin becomes as simple as creating a new list from our original list where each word has been translated into its Pig Latin equivalent. Written another way, suppose we have a list like this:

```
words = ["jim", "likes", "radishes"]
```

And we wish to make a new list that looks like this:

```
pig_words = ["im-jay", "ikes-lay", "adishes-ray"]
```

We could easily write a function to translate a single word at a time. Consider the following simple function.

```
def piglatin(english):
    return english[1:] + "-" + english[0] + "ay"
```

Then, we could construct a new list by using this **`piglatin`** function on each item in the original list. Observe:

```
sentence = input("Enter a sentence: ")
# sentence is now something like "jim likes radishes"

words = sentence.split()
# words is now something like ["jim", "likes", "radishes"]

pig_words = []
for word in words:
    pig_word = piglatin(word)
    pig_words.append(pig_word)
```

There is a far more concise way to do this in Python using something called *functional programming*. Observe that all we wish to do is make a new list from an original list by applying a function to each item in the list. The original list is **words**. The new list is **pig_words**. The function is named **piglatin**. We can compress the above code into the following code.

```
sentence = input("Enter a sentence: ")
# sentence is now something like "jim likes radishes"

words = sentence.split()
# words is now something like ["jim", "likes", "radishes"]

pig_words = list(map(piglatin, words))
```

Look at the last line of the above code. This statement tells Python to "map" the function **piglatin** onto all of the items in **words**. That is, it tells Python to pass each of the items in **words** to **piglatin**, and then take all the return values and make a new list out of them.

The expression **map(piglatin, words)** creates something that is like a **list**. We then must convert its return value to a **list**. Notice that **map** takes two arguments. The second is a list, but the first is actually a function. We can pass function as parameters to other functions. Python allows us to do this because we can store a reference to a function in any variable, including parameter variables.

To understand how to pass a function to another function, consider the following "toy" example.

```
def call_function(func, val):
    return func(val)

call_function(print, "Hi there")
# prints "Hi there"

def square(x):
    return x * x

y = call_function(square, 3)
print(y)
# prints 9
```

call_function takes the first parameter **func** and calls it. It uses **val** as the argument to **func**.

Consider (roughly) how **map** works.

```
def map(func, inlist):
    outlist = []
    for item in inlist:
        outlist.append(func(item))
    return outlist
```

We say this is how **map** works "roughly" because **map** does not actually return a **list**; it returns something like a **list** that we can convert to a **list**. The reason for this is beyond the scope of this book.

The **map** function is pretty slick! Consider the following examples.

```
xs = ["1", "2", "3"]
ys = list(map(int, xs))
# ys is now [1, 2, 3]

strings = ["cheese", "ate", "sand"]
lengths = list(map(len, strings))
# lengths is now [6, 3, 4]
```

This is really the tip of the iceberg when it comes to functional programming! This is just a taste, and most other programming languages support some form of functional programming. There are a lot more things you can do with functional programming that are super powerful and really concise in terms of coding. If you take future computer science classes and read more books, it is likely you'll encounter functional programming again. Functional programming is used frequently in "real world" code. In fact, Google's data processing pipeline is built around a functional programming paradigm known as "Map-Reduce." Google it!

5.9 (Optional) List Comprehensions

Based on what we know so far, we can construct lists in two ways. If we know the items we want in a list at the time we create the list, we can do so as follows.

```
mylist = [1, 2, 3, 4, 5]
```

Alternatively, we could create an empty list and then add items to it later, like this.

```
mylist = []
# Then, later on...
mylist.append(1)
mylist.append(2)
mylist.append(3)
mylist.append(4)
mylist.append(5)
```

There is an additional way to create lists that is somewhat unique to Python. This method is called a *list comprehension*. Consider the following code.

```
squares = []
for num in range(0, 10):
    squares.append(num**2)
# squares is now [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The list comprehension-way to write this is:

```
squares = [num**2 for num in range(0, 10)]
```

A list comprehension consists of square brackets containing an expression followed by a loop that gives the expression a sequence of values. The loop can contain additional loops or even if statements. Consider this:

```
evens = [x for x in range(0, 16) if x % 2 == 0]
# evens is now [0, 2, 4, 6, 8, 10, 12, 14]
```

The value for each item in **evens** is **x**, but only if **x** is divisible by **2**. The **if** expression can be used to "filter" items in the list.

List comprehensions can be handy if you have one list and you wish to make another new list based the values in the first list. For example:

```
sentence = "Sheamus Shaughnessy is HUNGRY."
words = [w.lower().replace(".","") for w in sentence.split()]
# words is now:
#     ['sheamus', 'shaughnessy', 'is', 'hungry']
```

Another example:

```
nonprimes = [y for x in range(2, 8) for y in range(x*2, 50, x)]
primes = [x for x in range(2, 50) if x not in nonprimes]
# primes is now:
#     [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Neat-o burrito!

(Mmm… burritos. Those who happen to be in Storm Lake, Iowa, may want to take a break from studying and head to La Juanitas for some exceptional, authentic Mexican food.)

5.10 Exercises

1. Suppose we define a list as follows:

   ```
   mylist = [1, 3, 5, 7, 9, 11]
   ```

   What is the type and value of each of the following expressions?

   ```
   mylist[1]
   mylist[0]
   mylist[-1]
   mylist[len(mylist)-1]
   mylist[1:3]
   mylist[3:]
   mylist[:3]
   7 in mylist
   8 in mylist
   len(mylist)
   sum(mylist)
   min(mylist)
   ```

```
max(mylist)
mylist[mylist[1]]
str(mylist[2])
",".join(["cat", "dog"])
```

2. What is the output of the following code?

```
xs = [1, 3, 5]
ys = xs
xs.append(7)
ys.append(9)
print(xs)
print(ys)
ys = xs[:]
ys.remove(9)
ys.pop(0)
print(xs)
print(ys)
```

3. Define a function named **`biggest`** that takes a **`list`** of numbers as a parameter and returns the biggest number in the list.

4. Define a function named **`average`** that takes a **`list`** of numbers as a parameter and returns the mean/average of the numbers as a **`float`**.

5. Define a function named **`contains_chuck`** that takes a **`list`** of names and returns **`True`**/**`False`** whether **`"Chuck Norris"`** or **`"Carlos Norris"`** is in the list.

6. Define a function named **`count_chucks`** that takes a **`list`** of names and returns an integer representing the number of names that start with **`"Chuck"`** in the list.

7. Define a function named **`cull_cullens`** that takes a **`list`** of names and returns a new **`list`** based on the original list, only in the new list any name ending in **`"Cullen"`** has been removed. For example:

```
in = ["Sarah James", "Edward Cullen", "Mary Smith",
      "Bella Cullen"]
out = cull_cullens(in)
# out contains only ["Sarah James", "Mary Smith"]
```

8. Define a function named **replace** that takes a list of values, an "old" value, and a "new" value. The function should modify the list of values by replacing any "old" values with the "new" value. For example:

```
game = ["duck", "duck", "grey duck", "duck", "grey duck"]
replace(game, "grey duck", "goose")
print(game)
# This should print ["duck", "duck", "goose", "duck", "goose"]
```

9. Define a function named **shuffle_list** that takes a **list** as a parameter and returns a **list** of the same size only with the items in the parameter list in different positions. That is, the returned list should have the items randomly assigned to new positions; granted, it is possible that some of the items may still be in their original positions. For example:

```
cards = ["4D", "AS", "3C", "JH", "5C"]
new_cards = shuffle_list(cards)
# new_cards will be different than cards, for example:
#    ["JH", "3C", "4D", "5C", "AS"]
```

10. Define a function **shuffle_this** just like the function **shuffle_list** in the previous problem, only this time change the original list rather returning a new list. For example:

```
cards = ["4D", "AS", "3C", "JH", "5C"]
shuffle_this(cards)
# cards will now contain something like:
#    ["JH", "3C", "4D", "5C", "AS"]
```

11. Define a function named **merge** that takes two lists and returns a new single list that is the result of interleaving the two original lists. If one

list is exhausted before the other, the remaining items are tacked on to the end of the resulting list. To perform the merge, the function should select the first item from the first list, the first item from the second list, the second item from the first list, the second item from the second list, etc. For example:

```
new_list = merge([1, 3, 5, 7], [2, 4, 6])
# new_list should be [1, 2, 3, 4, 5, 6, 7]
```

12. Suppose we have a list of lists (a "grid") representing exam grades in a class. The rows are individual students' scores. The columns are the exam scores for each student. Each score is based on a possible score of 100 points.

    Define a function named **`print_averages`** that takes a grades "grid" and a names **`list`** as parameters, and then it should print the exam average for each student along with the student's name. The function does not need to return anything.

    NOTE: you cannot assume that there will only be three scores or three students. The code you write should still work correctly even if we were to add columns or rows. The code below serves only as an example.

```
grades = [ [100.0, 98.0, 99.0],
           [ 68.5, 79.0, 85.5],
           [ 88.5, 99.0, 87.5] ]
names = ["Samantha Beeson", "Reginald Ronald", "Dani Smith"]
```

13. Recall the **`votes`** grid example in the section on lists of lists (i.e., Section 5.7). Define a function named **`county_winners`** that prints the name of the candidate who wins each county. The output should display a line for each county. Each output line should be of the form **`"CANDIDATE has won COUNTY."`** where **`CANDIDATE`** is the candidate's name and **`COUNTY`** is the name of the county. To define this function, you will need to create a list that stores the names of the counties in addition to defining the function body itself.

An example of needed list definitions are:

```
votes = [
    [25, 18,  2],
    [19, 20,  1],
    [25,  4,  5],
]
candidates = ["Ronald Rump", "Billary Blimpton", "A Giant Meteor"]
counties = ["Poweshiek", "Winneshiek", "Ironsheik"]
```

Sample output:

```
Ronald Rump has won Poweshiek county.
Billary Blimpton has won Winneshiek county.
Ronald Rump has won Ironsheik county.
```

Chapter 6

Files and Exceptions

6.1 Opening and Reading files

Of the time people spend using computers, a very large proportion of the time is spent on managing files. People take photos, organize them, and sometimes edit them on their laptops. People open Word documents, edit them, and save their changes. People may make copies of files or move files from one folder to another. In some cases, people manage files directly. In other situations, people may be blissfully unaware that programs are managing files on their behalf. Suppose you are playing a video game. When you, the user, save your game, the game program most likely stores the level you were on in a file. The game would then open and read information from this file when you start the game again so that you can start where you left off. In this chapter, we will learn how to create files, read from files, and write to files using Python.

As we often do in this book, we will begin with an example. In Thonny, create a new code file named **`show-menu.py`**. Do not type anything into it just yet, but leave the new code file open. Now create a second file and name it **`menu.txt`**. Take note that this file ends in **`.txt`** rather than **`.py`**. Be sure to save the file **`menu.txt`** in the same folder as you save the file **`show-menu.py`**. This **`menu.txt`** file will not contain Python code. Instead, it should contain the following lines. Go ahead and type them into the file and save it.

```
roast duck
ribeye steak
wood-fired pizza
plank-seared salmon
caesar salad
```

Now, return to the window for **show-menu.py**. Still, do not type anything into it, but go ahead and run the empty code file. This should shift the focus back to the Python Shell window. In the Python Shell window type the following.

```
open("menu.txt", "r")
```

What appears when you type this? Now, type this:

```
f = open("menu.txt", "r")
type(f)
```

Even though the type of **f** is reported as having the type **_io.TextIOWrapper**, in general we will refer to **f** as a **file** type variable. When we use the **open** function, we pass it the name of the file we wish to open, and we also tell it what we intend to do with the file. In this case, the **"r"** indicates that we wish to "read" from the file.

Now try this in the Python Shell window:

```
f.readlines()
```

Interesting. This function returns a list that looks like **['roast duck\n', 'ribeye steak\n', 'wood-fired pizza\n', 'plank-seared salmon\n', 'caesar salad\n']**. Each item in the list is a single line in the file. The **"\n"** is the newline character. A newline can be found at the end of every line.

Try reading the lines again by typing:

```
f.readlines()
```

Oh no! Where did the lines go? Has our file been deleted? Is the file still there but its contents have been deleted?

The good news is the file is still there and none of the lines have been harmed in the process. For evidence, try the following.

```
f.seek(0)
f.readlines()
```

Hey, they're back! What do you think **`f.seek(0)`** does? (We'll explain exactly what it does in a bit.)

Okay, now let's try these lines in the Python Shell window.

```
f.seek(0)
f.readline()
f.readline()
f.seek(0)
f.readline()
f.readline()
```

The singular **`f.readline()`** works differently than the plural **`f.readlines()`**. **`readline`** simply returns a single line as a string. **`readlines`** returns all of the lines as a **`list`** of strings.

We can also do this:

```
f.seek(0)
f.read()
```

The function call **`f.read()`** returns the entire file as a single string.

Finally, we can do this.

```
f.close()
```

This closes the file we opened earlier. If we try to **seek** or **readline**, the program will encounter an error. We can check to see whether a file is closed by typing:

```
f.closed    # type is bool
```

Let's make sense of how this all works. When we open a new file, imagine that we are setting up a "pipe" in front of our file. Through the pipe, we can pull lines out of a file and into our Python program, line by line. At any point, the end of the pipe is pointing at the next line to be retrieved.

Suppose we type the following into the Python Shell window.

```
line = ""
f = open("menu.txt", "r")
line = f.readline()
print(line)
line = f.readline()
print(line)
```

With each call to **readline()**, the next line will be read through the file's pipe, and then the pipe will be moved to the location in the file to be read next. To help visualize this process, the first and second **readline** calls are shown in Figure 6.1 and Figure 6.2, respectively. Note how after the first **readline**, the pipe is moved down in Figure 6.2, ready to read the next line.

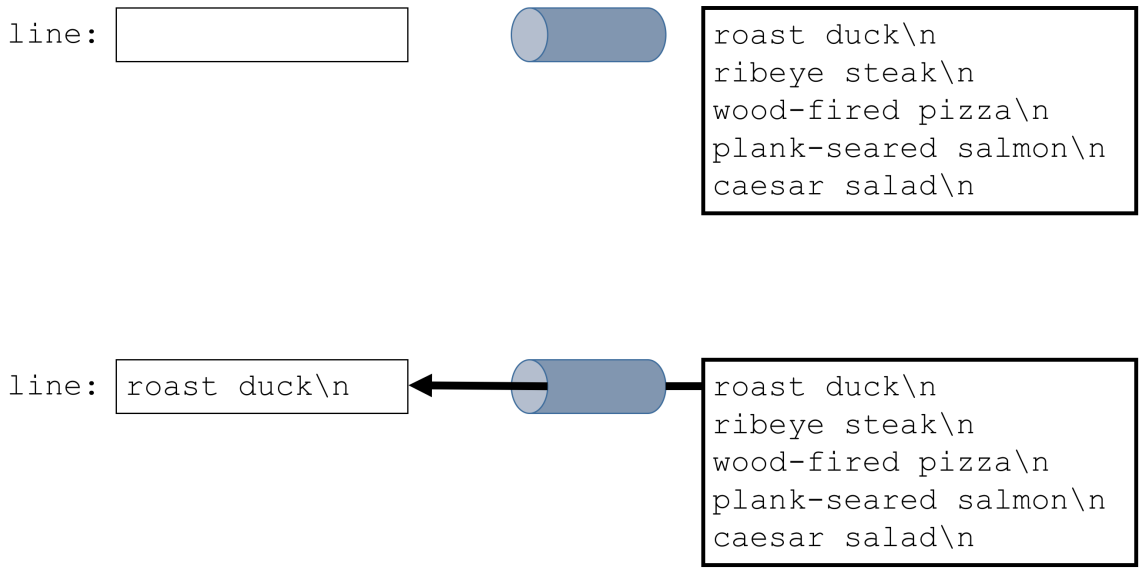


Figure 6.1: The effect of the first `f.readline()`

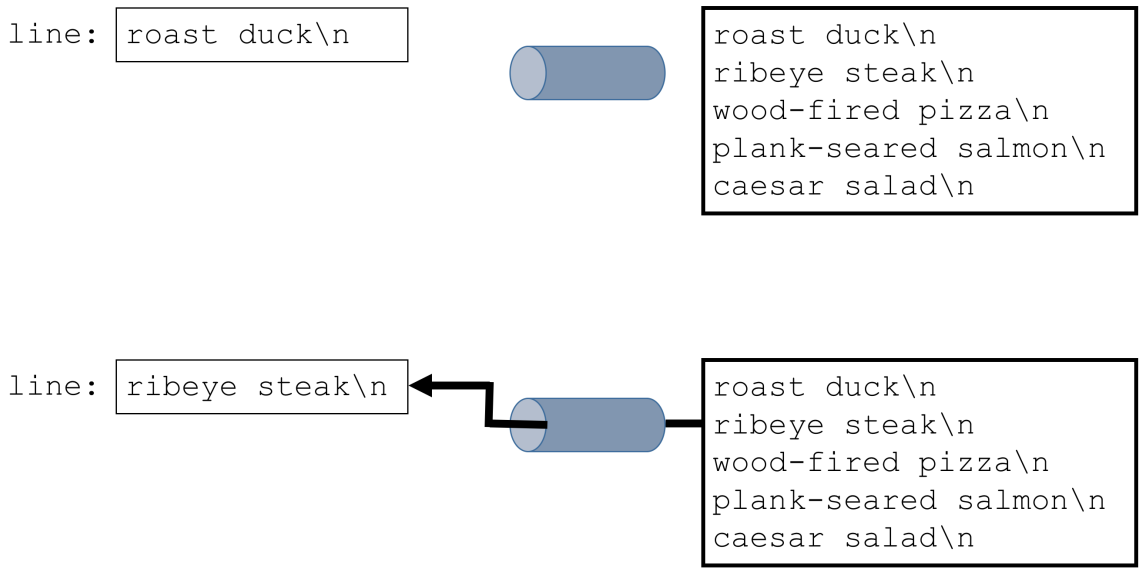


Figure 6.2: The effect of the second `f.readline()`

When we write `f.seek(0)`, we are telling the file to reset the pipe to the

beginning of the file.

Now, we're finally ready to write some code in **show-menu.py** (see Listing 6.1).

Listing 6.1

```
f = open("menu.txt", "r")
lines = f.readlines()
for food in lines:
    print("They have %s." % food)
    if food == "roast duck":
        print("Ooo, they have duck!  Yummy!")
f.close()
```

The code in Listing 6.1 opens the file and retrieves all of the lines as a list of strings. Then, we use a **for** loop to print each one of the lines; in the loop we call each line **food**. If the food happens to be **"roast duck"** we print an especially excited message. At the end of the code, we close the file. It is always good practice to remember to close any files you open.

Save your code file and run it. Does the output look like what you would expect (I would guess the answer is "no")? Let's look at the output.

```
They have roast duck
.
They have ribeye steak
.
They have wood-fired pizza
.
They have plank-seared salmon
.
They have caesar salad
.
```

First of all, we don't see our excited message at all. Second, the period at the end of each **print** statement is on the next line.

Look again at the list value returned by **f.readlines()**, which is **['roast duck\n', 'ribeye steak\n', 'wood-fired pizza\n', 'plank-seared salmon\n', 'caesar salad\n']**. Every string value within the list includes a **"\n"** at the end. The **"\n"** is the newline that separates the text on

each line. Since **print** normally inserts a newline after it prints its string, and since the string has a newline, too, there are two newlines that get printed: one before the period and one after the period.

This also causes our **if** statement to fail. When we compare **"roast duck"** to **"roast duck\n"**, these are technically different strings because the second one has a **"\n"** character at the end. That's why we never see our excited message. The expression **"roast duck" == "roast duck\n"** is **False**.

Let's fix this code; see Listing 6.2.

Listing 6.2

```
f = open("menu.txt", "r")
lines = f.readlines()
for food in lines:
    food = food.rstrip()
    print("They have %s." % food)
    if food == "roast duck":
        print("Ooo, they have duck!  Yummy!")
f.close()
```

The **rstrip** function removes any "whitespace" characters from the end of the string. Whitespace characters include spaces, tabs, and newlines. Make sure you remember to call the **rstrip** function using parentheses. It's a common mistake to forget. It is perfectly "legal" in Python to do this:

```
food = food.rstrip     # This is INCORRECT.
```

Instead of this

```
food = food.rstrip()   # This is correct.
```

The second one is correct. The first one is wrong but does not cause an error. Variables can store *references* to functions, so the first statement would change the variable food from a string to a function. There are reasons why we would want to put function references in variables (which you may have encountered

if you read "optional" Section 5.8), but this is not one of those instances. So, be sure to call the function by using parentheses.

Another way we could have accomplished this is to use the singular **readline** instead of **readlines**. Consider Listing 6.3.

Listing 6.3

```
f = open("menu.txt", "r")
food = f.readline()
while food != "":
    food = food.rstrip()
    print("They have %s." % food)
    if food == "roast duck":
        print("Ooo, they have duck!  Yummy!")
    food = f.readline()
f.close()
```

We retrieve the first food using **f.readline()**. As long as the food is not an empty string, we can display it using the code in the body of the **while** loop. At the end of the **while** loop, we need to remember to retrieve the next line from the file. When we have reached the end of the file, **f.readline()** returns an empty string (that is, **""**).

6.2 Reading File Records

In the previous section, we had a file that contained a list of food items. What if we wanted to store more information about each food item beyond simply its name? Suppose we wanted to also track how many calories and carbohydrates the food item contains? One way we might do this is to store each piece of information on a separate line in a new file (let's name it **new-menu.txt**), like this.

```
roast duck
1284
0
ribeye steak
847
```

```
0
wood-fired pizza
1680
200
```

The attributes of each item are now listed on three separate lines. The first line is the name of the food item. The second line is the calorie count. The third line is the number of carbohydrates ("carbs") in grams.

Let's write code to show all of the items and their attributes, and at the end of the program, let's announce which food has the lowest calorie count. Let's focus on the first part initially (see Listing 6.4), and then we'll come back and add to the code to print out the item with the lowest calorie count.

Listing 6.4: Showing all records in a file

```
infile = open("new-menu.txt", "r")
item = infile.readline()
while item != "":
    item = item.rstrip()
    calories = int(infile.readline().rstrip())
    carbs = int(infile.readline().rstrip())
    print("%s contains %d cal and %d carbs." \
              % (item, calories, carbs))
    item = infile.readline()
infile.close()
```

In Listing 6.4, we attempt to read lines in sets of three's since each item now consists of three lines of attributes, the item name, the calorie count, and the carbohydrate count. If we ever get an empty string for the first line in the next set of three lines, we know we have reached the end of the file.

Now, let's add the code to track which food as the lowest calorie count. For now, if there is a tie, we will announce only the first food with the lowest calorie count (see Listing 6.5).

Listing 6.5: Finding lowest calorie count in a set of file records

```
lowcal = 100000   # a really big number!
lowitem = ""
```

```
3
4  infile = open("new-menu.txt", "r")
5  item = infile.readline()
6  while item != "":
7      item = item.rstrip()
8      calories = int(infile.readline().rstrip())
9      carbs = int(infile.readline().rstrip())
10     print("%s contains %d cal and %d carbs." \
11               % (item, calories, carbs))
12
13     if lowcal > calories:
14         lowcal = calories
15         lowitem = item
16
17     item = infile.readline()
18
19 infile.close()
20
21 print("The food with the least calories is %s with %d cal." % \
22       (lowitem, lowcal))
```

In Listing 6.5, we make two variables before the loop starts. **`lowitem`** keeps track of the name of the lowest calorie item and **`lowcal`** keeps track of how many calories it was. Then, inside the loop, we check each item to see if its calorie count is lower than what we've seen so far in the file. Once the loop is finished and we've examined all items, we print our findings.

Storing related values on separate lines is not uncommon, though real-world data is often stored differently. It is very common to find data stored in files using the comma-separated values (CSV) format. Let's explore how to read CSV files. To do this, let's make yet another new file and name it **`menu.csv`** (note the **`.csv`** file extension). The contents of this file should appear as follows.

```
#item,cal,carb
roast duck,1284,0
ribeye steak,847,0
wood-fired pizza,1680,200
```

Now, the information for each item is stored on a single line, and each "attribute" is separated by commas. It is fairly common to have the first line in the file not contain actual data, but rather it contains a description of the order of attributes in each subsequent line. This first line is often called the *header*

line or *header row* of information. In the code we will write, we will need to remember to read that header line, but then simply ignore its contents.

Our CSV-related code becomes Listing 6.6.

Listing 6.6: Finding lowest calorie count in CSV file

```
lowcal = 100000   # a really big number!
lowitem = ""

infile = open("menu.csv", "r")

# Read the header and throw it away.
line = infile.readline()

# Get the actual first line.
line = infile.readline()

while line != "":
    attributes = line.rstrip().split(",")
    item = attributes[0]
    calories = int(attributes[1])
    carbs = int(attributes[2])
    print("%s contains %d cal and %d carbs." \
              % (item, calories, carbs))

    if lowcal > calories:
        lowcal = calories
        lowitem = item

    line = infile.readline()
infile.close()

print("The food with the least calories is %s with %d cal." % \
      (lowitem, lowcal))
```

We use `.split(",")` to split the line into its three pieces: the item, the calories, and the carbs. Then, we proceed as we did previously.

6.3 Writing Information to Files

So far in this chapter, we have written code to read information from existing files. Let's learn how to write code that writes to files. That is, we may want to create new files, or we may wish to update existing files.

We shall continue with another food-related example, and we'll hope that you're not reading this book close to meal time, otherwise it may be hard to concentrate!

Suppose we want to create a program that allows us to make a menu for a meal that we would then save for later use. Let's save this code into a file named **write_menu.py**. Let's ask for each menu item, one after another, until the user enters a bare enter/return (i.e., an empty string). Then, we'll write each menu item line to a file that we will name **menu.txt**. Listing 6.7 shows the code to accomplish this in all its wonderous Python-y glory.

Listing 6.7: Creating **menu.txt**

```
1 f = open("menu.txt", "w")
2 food = input("What do you want to eat? (Enter nothing to quit.) ")
3 while food != "":
4     f.write(food + "\n")
5     food = input("What do you want to eat? (Enter nothing to quit.) ")
6 f.close()
```

Give this code a try. After the program ends, a file named **menu.txt** should "magically" appear in the same folder as your code file **write_menu.py**. Go ahead and open **menu.txt** and verify that all the items you typed when you first ran the program do in fact appear in the file, each on a separate line.

Take note of line 1 in Listing 6.7. See how we changed the second parameter of the function call to **open**? Instead of **"r"**, which means "read", we typed **"w"**, which means "write."

Also, consider line 4 in Listing 6.7. The file variable **f** has a function named **write**. Unlike **print**, which always inserts a newline (**\n**) at the end of output, **write** does not automatically add a newline. Thus, we must manually insert the newline by passing the string expression **food + "\n"** to our call to **f.write**. If we omitted the **"\n"**, all of the food items you entered would be squished together on one unreadable line.

There is a downside to this program. Whenever we run this code, **open("menu.txt", "w")** obliterates any pre-existing **menu.txt** file. Any menu that we have previously created is destroyed. That's maybe not such a good thing. It might be

a good idea to show users the menu we already have, if there is one, and then ask if they want to create a new menu.

Okay, here we go! Check out Listing 6.8.

Listing 6.8: Show and re-create **menu.txt**

```
1  infile = open("menu.txt", "r")
2  print("The current menu on file is:")
3  for menuitem in infile:
4      print(menuitem.rstrip())
5  infile.close()
6  print()
7
8  create_new = input("Would you like to throw away this menu " +
9                     "and create a new one? (y/n): ")
10
11 if create_new == "y":
12     outfile = open("menu.txt", "w")
13     food = input("What do you want to eat? (Enter nothing to quit.) ")
14     while food != "":
15         outfile.write(food + "\n")
16         food = input("What do you want to eat? (Enter nothing to quit.) ")
17     outfile.close()
```

Run the code in Listing 6.8. It should show you the contents of **menu.txt** and then ask you if you want to create a new menu. Enter **n** and press enter/return. Re-run the code. Your menu items should still be intact! Now run the program again. When it asks you if you want to create a new menu, enter **y**. You will be able to enter several new menu items. When you are finished entering items, you should be able to see those menu items whenever you re-run the program.

Our code does suffer from at least one problem, however. Line 1 of Listing 6.8 assumes there is a file named **menu.txt**. What if there isn't? In other words, if we were to run this program for the first time without creating a **menu.txt** ahead of time, what will happen. Well, let's try that. Delete the file **menu.txt**. Re-run the program. Kablooey!

```
Traceback (most recent call last):
  File "t.py", line 1, in <module>
    infile = open("menu.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'menu.txt'
```

Our code is brittle. We will fix this in Section 6.4.

6.4 Exceptions

Let's not delay in fixing Listing 6.8. The code, again, for the sake of convenience, is:

```
1  infile = open("menu.txt", "r")
2  print("The current menu on file is:")
3  for menuitem in infile:
4      print(menuitem.rstrip())
5  infile.close()
6  print()
7
8  create_new = input("Would you like to throw away this menu " +
9                     "and create a new one? (y/n): ")
10
11 if create_new == "y":
12     outfile = open("menu.txt", "w")
13     food = input("What do you want to eat? (Enter nothing to quit.) ")
14     while food != "":
15         outfile.write(food + "\n")
16         food = input("What do you want to eat? (Enter nothing to quit.) ")
17     outfile.close()
```

To recap, the problem lies in line 1, which assumes that we already have a file named **`menu.txt`** that resides in the same folder as our code. If that file does not exist, then our code will crash at line 1. Furthermore, lines 2 through 9 also rely on this assumption, namely, that we already have a menu saved. So, our first instinct might be to "protect" lines 1 through 9 with an **`if`** statement. The Python standard libraries has a package named **`os`** that has functions that let us look at the contents of the current folder. We can use it as shown in Listing 6.9.

Listing 6.9: Show and create menu with error checking

```
1  import os
2
3  create_new = "y"
4
5  if os.path.isfile("menu.txt"):
```

```
6       infile = open("menu.txt", "r")
7       print("The current menu on file is:")
8       for menuitem in infile:
9           print(menuitem.rstrip())
10      infile.close()
11      print()
12
13      create_new = input("Would you like to throw away this menu " +
14                         "and create a new one? (y/n): ")
15
16  if create_new == "y":
17      outfile = open("menu.txt", "w")
18      food = input("What do you want to eat? (Enter nothing to quit.) ")
19      while food != "":
20          outfile.write(food + "\n")
21          food = input("What do you want to eat? (Enter nothing to quit.) ")
22      outfile.close()
```

In Listing 6.9, we've indented lines 6 through 14 from our previous code (Listing 6.8). We are now importing the **os** module and using it in line 5 to ensure that **menu.txt** exists before we try to open it and read from it. We also need to initialize the variable **create_new** since we only ask the question of whether to create a new menu if we already have a menu created (see line 3).

Try to break this code. Remove **menu.txt** and run the code. What happens?

It looks like we've fixed everything, so congratulations! Give yourself a pat on the back. If that is difficult, you may want to consider stretching your shoulders more often (heh!).

The strategy we just used is one we've used before in this book, specifically, we've used an **if** statement to "guard" a block of other statements. That's not the only way to solve this problem, however. There is another way to do it, and it is one that might make other problems easier to solve in the future as well. The other way is to utilize *exceptions*.

You've seen exceptions before in code. In fact, we saw one at the end of the last section.

```
Traceback (most recent call last):
  File "t.py", line 1, in <module>
    infile = open("menu.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'menu.txt'
```

FileNotFoundError is an exception! An exception is an "exceptional condition" that, if unhandled, can cause the program to crash. We've seen others of these earlier in the book as well: **NameError**, **TypeError**, **IndexError**, etc.

Notice what we just said about exceptions: they *can* cause a program to crash. We can prevent them from causing a crash, however. We can *catch* exceptions when they happen, and then we can respond appropriately to them.

Before we go back to Listing 6.9 and try to solve the problem a different way, let's look at a more simple example that uses exceptions. Let's start with the following code.

```
age = int(input("How old are you? "))
if age >= 18:
    print("You are old enough to vote.")
else:
    print("You are not old enough to vote yet.")
```

This code is all fine and dandy as long at the users properly enter an integer as their input. If the user types something other than an integer (say, "cheetos"), the call to **int** will produce an exception named **ValueError**.

```
Traceback (most recent call last):
  File "s.py", line 1, in <module>
    age = int(input("How old are you? "))
ValueError: invalid literal for int() with base 10: 'cheetos'
```

Instead, we can "try" a certain bit of code, and then also write code that should be executed only in the event we receive an exception, like this:

```
1  try:
2      age = int(input("How old are you? "))
3      if age >= 18:
4          print("You are old enough to vote.")
5      else:
6          print("You are not old enough to vote yet.")
7  except ValueError:
8      print("That does not look like a valid age.")
```

We "try" lines 2 through 6. If we encounter a **ValueError**, we immediately jump down to line 8 and execute that code. If we didn't have **try** and **except**, we would have to write code like this:

```
age = input("How old are you? ")
if age.isdigit():
    age = int(age)
    if age >= 18:
        print("You are old enough to vote.")
    else:
        print("You are not old enough to vote yet.")
else:
    print("That does not look like a valid age.")
```

One of the next things about **try**/**except** is we can focus on "normal" conditions in our code and then handle "edge" cases in our **except** block.

For the sake of completeness in our terminology, when a function generates an exception, we sometimes say that the function *throws* or *raises* an exception. The **except** block is responsible for *catching* or *handling* an exception.

A **try** block can be followed by more than one **except** block. You might think of each **except** block as being like an **elif** in that we can have different blocks of code to be executed depending on the type of exception we catch. Consider Listing 6.10.

Listing 6.10: Show and create menu with error checking

```
1  # Try commenting out the different lines in the try block to see what
2  # exceptions get caught and handled.
3  try:
4      float("cheetos")
5      ["a","b"][7]
6      x
7  except ValueError:
8      print("ValueError")
9  except IndexError:
10     print("IndexError")
11 except:
12     print("Error")
```

Line 4 of Listing 6.10 generates a **ValueError**. If we comment out line 4, then line 5 throws an **IndexError**. If we comment out lines 4 and 5, then

line 6 throws a **NameError** since **x** is not defined. The last **except** that starts on line 11 catches all other exceptions, including **NameError**. Thus, we do not necessarily need to state the type of exception to catch.

Now, let's return to our file reading/writing example involving menus. We can use exceptions to deal with the situation where there is no **menu.txt** file. Consider Listing 6.11.

Listing 6.11: Show and create menu with exception handling

```
1  create_new = "y"
2
3  try:
4      infile = open("menu.txt", "r")
5      print("The current menu on file is:")
6      for menuitem in infile:
7          print(menuitem.rstrip())
8      infile.close()
9      print()
10     create_new = input("Would you like to throw away this menu " +
11                        "and create a new one? (y/n): ")
12 except:
13     pass
14
15 if create_new == "y":
16     outfile = open("menu.txt", "w")
17     food = input("What do you want to eat? (Enter nothing to quit.) ")
18     while food != "":
19         outfile.write(food + "\n")
20         food = input("What do you want to eat? (Enter nothing to quit.) ")
21     outfile.close()
```

The **except** block in Listing 6.11 handles any exceptions encountered in lines 4 through 11. Since **except** must have a body and there's really nothing more to do once the exception is caught, we simply write **pass**. **pass** is just an empty statement in Python that doesn't do anything other than serve as a placeholder.

So how do exceptions get raised in the first place? Consider the following example code (see Listing 6.12).

Listing 6.12: Raising an exception

```
try:
    print("hello")
    raise Exception
    print("goodbye")
except Exception:
    print("Not so fast!")
```

The output of the code in Listing 6.12 is:

```
hello
Not so fast!
```

Raising an exception can be a good way in code to say "something's not right and we need the programmer to handle it appropriately."

6.5 Machine Representation (a.k.a. Bits, Bytes, and Nybbles)

The files our programs have accessed so far in this chapter have all been *text files*. We have appended the file extension **`.txt`** to the file's name to remind us that the files contain plain text. Since they contain only plain text, it is easy to read their contents in any text editor, including Thonny.

Not all files are text files. In fact, many files are not text files. Image files (which typically end with **`.jpg`**, **`.png`**, or **`.bmp`**, among others) are not easily examined by just opening the file in a text editor and examining the contents. If you do, the file will look like gibberish. Other programs such as image viewers or Web browsers must be used for us to view the contents of an image file.

Image files belong to a class of files known as *binary files*. Rather than consisting of lines of strings of characters, binary files contain *bytes*. (Actually, text files really consist of bytes, too, but don't concern yourself with that now. We'll come back to that later.)

We can read bytes from files and we can write bytes to files if we want programs that create or modify images, manipulate music files, or edit video

files. Before we get into that, we really need to understand what the heck a byte is in the first place.

Okay, so here's a byte:

$0110\ 0101_2$ What?

Why is there a subscripted 2 after a bunch of zeroes and ones? What is this supposed to mean to us?

In short, this byte is a number, but it doesn't look quite like we expect numbers to look. To understand what this means, we have to digress a bit. What follows is actually kind of cool.

Consider something that looks more like a number. How about 16? Say the name of this number aloud. If you said "sixteen", then you're correct! But, what does this number *mean*? It depends. Maybe it conveys how many years you had been alive before you got your driver's license. Maybe it's the number of Mountain Dews in your refrigerator. Numbers represent a *magnitude*, or the *amount* of something.

Even more importantly, we can look at a number and get a sense of just *how big* it is. Why?

1. A number is just a string of symbols.

2. The order of the symbols conveys magnitude.

3. The symbols themselves are convey magnitude.

There is a rule for what symbols we are allowed to use. Each symbol can be a 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Ever wondered why?

Well, let's go back to the number sixteen. Here's what the string of symbols that makes up the number sixteen (i.e., the "1" and "6") tell us.

$$\begin{aligned} 16 &= 1 \times 10^1 + 6 \times 10^0 \\ &= 10 + 6 \end{aligned}$$

The "1" and the "6" tell us we have one ten and six ones. Tens is a larger magnitude than ones.

Suppose we have the number 9. What happens when we add 1 to 9?

$$9 + 1 = 10$$

We run out of "room" in one digit to hold the magnitude of the number, so we add 1 to the next digit to the left. We now have a "ten," not just ones.

The numbers we use on a daily basis are called *decimal* numbers. The position of each digit represents a power of ten, hence the prefix *deci-*. Valid symbols in each digit position are 0 through 9.

This may all seem very elementary, but it sets the table for what comes next, and it's important to realize that numbers are actually strings whose pieces have meaning.

With all this in mind, can you successfully write other numbers in the expanded form we used above? In other words, if I give you 16, can you write $1 \times 10^1 + 6 \times 10^0$? Let's find out.

Write 216 and 1000 in expanded form.

Did you come up with the following?

$$\begin{aligned} 216 &= 2 \times 10^2 + 1 \times 10^1 + 6 \times 10^0 \\ &= 200 + 10 + 6 \end{aligned}$$

$$\begin{aligned} 1000 &= 1 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 0 \times 10^0 \\ &= 1000 + 0 + 0 + 0 \end{aligned}$$

We will also refer to this expanded form as *sum of products* since the positions of the digits in a number tell us its value when we add up the values of the digits, each of which involve multiplication with a power of ten.

When we think about computers, we can envision them as electrical machines. At their simplest level, they consist of lots of very tiny electrical wires where each wire is either "on" or "off." Because of this, there's no good way to store numbers in computers using the digits 0 through 9. We can only have 0 and 1 where we think of 0 as "off" and 1 as "on." (For a more comprehensive discussion of why it would be hard to create decimal computer, your author

strongly encourages you to take an introductory physics course that covers electromagnetism!)

Since we only have 0 and 1 to work with as symbols, we can no longer work in base-10, namely, having digit positions represent powers of ten. Since digits 0 - 9 allow us to work with powers of ten, digits 0 - 1 only allow us to represent powers of two. We said that numbers whose symbols convey powers of ten are called *decimal* numbers. Numbers whose symbols convey powers of two are called *binary numbers*.

Let us consider the binary number 101. It's sum-of-products expanded form would be:

$$\begin{aligned} 101 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 4 + 0 + 1 \\ &= 5 \end{aligned}$$

We took a lot of liberty with how we write numbers, because it looks like I just wrote $101 = 5$, which doesn't make any sense! We need a way to let people know if 101 is a binary number or a decimal number. We will write the *base* of the number (base-2 binary or base-10 decimal) as a subscript following the number. Thus:

$$101_2 = 5_{10}$$

In other words, "101" in binary means the same as "5" in decimal.

To more exactly show sum-of-products in the prior example, we would write:

$$\begin{aligned} 101_2 &= (1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)_{10} \\ &= (4 + 0 + 1)_{10} \\ &= 5_{10} \end{aligned}$$

The subscript 10 following the parentheses says "the stuff in the parentheses is in base-10."

Okay, let's see if this is making sense. What is the decimal value of 11011_2 in binary? Write sum-of-products expansion to figure it out. Try it, and try not to look ahead until you have an answer.

Here's the answer, and we'll show a few more steps this time so everything is more clear.

$$\begin{aligned} 11011_2 &= (1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)_{10} \\ &= (16 + 8 + 0 + 2 + 1)_{10} \\ &= 27_{10} \end{aligned}$$

Try another: 11111_2.

$$\begin{aligned} 11111_2 &= (1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)_{10} \\ &= (16 + 8 + 4 + 2 + 1)_{10} \\ &= 31_{10} \end{aligned}$$

The value of each digit in a binary number is twice the amount of the digit to its right. This is similar to how each digit in a decimal number is ten times the amount of the digit to its right.

It's worth seeing what it looks like to count in binary versus how we traditionally count in decimal. Table 6.2 shows a side-by-side comparison of counting in these two bases. Note that we have done a few things to make the binary numbers easier to read and understand. First, we have placed leading zeroes in front of the binary numbers. This is something that is commonly done with binary numbers so that it is easier to see where the 0's are and where the 1's are. Also, we have placed a space between each group of four binary digits. This also helps to tell which digit places are occupied by which 0's and 1's. This is similar to the use of commas in decimal numbers when we have larger numbers (e.g., 157,246,398).

Did you notice some of the patterns that emerge in Table 6.2 when we count in binary? The right-most binary digit flips between 0 and 1 with each new number we count. Why? Well, think about when we count in decimal. When

Table 6.1: Counting in decimal versus binary

Decimal	Binary	Decimal	Binary
0	0000 0000	10	0000 1010
1	0000 0001	11	0000 1011
2	0000 0010	12	0000 1100
3	0000 0011	13	0000 1101
4	0000 0100	14	0000 1110
5	0000 0101	15	0000 1111
6	0000 0110	16	0001 0000
7	0000 0111	17	0001 0001
8	0000 1000	18	0001 0010
9	0000 1001	19	0001 0011

we add 1 repeatedly in decimal, it takes us until we reach 9 and then add 1 before we run out of "room" in the right-most digit, so we add one to the tens digit. In binary, we run out of "room" in the right-most digit every other time we add a 1. Thus, that digit flips every time. For the same reason, each binary digit "flips" half as often as the digit to its right.

Let's introduce some handy terminology. The word *bit* is used to describe a **bi**nary **di**git. A bit is either a 0 or a 1. If a bit is a 1, we say the bit is *set*. Bits are typically grouped into collections of 8. 8 bits is a *byte*. We typically separate bits in a byte into groups of 4 bits for the sake of readability. A group of 4 bits is called a *nybble* (I am not making this up!).

Suppose we jumped ahead in our counting, and I told you that

$$0011\ 1111_2 = 63_{10}.$$

Can you use your powers of deduction to find x in

$$0100\ 0000_2 = x_{10}?$$

Think about it.

What happens when we add 1 to a string of all 1's with binary numbers? Look again at Table 6.2. Can you see how adding the 1 causes all of the digits

Table 6.2: Counting in decimal versus binary

Character	Decimal ASCII Code	Binary ASCI Code
J	`65`	`0100 0001`
a	`97`	`0110 0001`
s	`115`	`0111 0011`
o	`111`	`0110 1111`
n	`110`	`0110 1110`

to run out of "room" again. Thus, adding a 1 to binary number consisting solely of 1's results in all of the bits being "flipped." Therefore,

$$0011\ 1111_2 = 63_{10}$$
$$0100\ 0000_2 = 64_{10}$$

Also, we should expect $0100\ 0000_2$ to be a power of two since only a single bit is set.

It's easy to tell if a binary number is even or odd. If the right-most bit is set, it's odd. Otherwise, it's even.

All information in a computer is stored as bytes. Since bytes are numbers, you might wonder how things like character strings are stored in your computer. Each number corresponds to a character using something called a *character encoding*. For example, the number `65` (`0100 0001` in binary) corresponds to the capital letter `A`. There are different character encodings that computers use. The most basic one is ASCII. More comprehensive ones exist that include characters from different languages, such as Unicode. Unicode even includes encodings for emojis!

If you look at the ASCII codes (use the link above), you can translate your name into the bytes that would represent it in computer memory. For example, your author's first name would reside in computer memory as:

The `J` would be stored first as `0100 0001`, followed by the `a` as `0110 0001`, and so forth.

6.6 Exercises

For each of these exercises, your code should properly handle situations where files that should exist fail to exist. That is, rather than just allowing the program to crash, you should handle the exceptional condition and print a helpful error message, if possible.

1. Write code that reads lines from a file named `input.txt`. It should then print each line to the screen, but it should prefix the line's contents with the line number on which it appears.

   For example, if the contents of `input.txt` are:

   ```
    Haikus are easy
   But sometimes they don't make sense
   Refrigerator
   ```

   Then the code should print to the screen:

   ```
    1. Haikus are easy
   2. But sometimes they don't make sense
   3. Refrigerator
   ```

2. Write code that reads from a file named `input.txt`. The code should print to the screen the *even-numbered* lines in the file. The lines should be prefixed with the line number on which they appear in the file (much like the previous problem).

3. Repeat the previous problem, only this time the program should prompt the user for a file name rather than assuming `input.txt` should be used.

4. Write code that writes the first 20 positive even numbers to a file named `evens.txt` such that each number is on a separate line in the file.

5. Suppose we have a text file named `temps.txt` that contains a series of temperatures. Each temperature is a `float` on a single line in the file. Write code that prints the *average* temperature given all the temperatures in the file.

6. Write code that replaces each line in a file with its reverse. The program should prompt the user for a file name first.

   For example, if the contents of the file are:

   ```
    The quick brown fox jumps over the lazy dog.
   Pack my box with five dozen liquor jugs.
   ```

   Then the new contents should become:

   ```
    .god yzal eht revo spmuj xof nworb kciuq ehT
   .sguj rouqil nezod evif htiw xob ym kcaP
   ```

7. Repeat the previous exercise, only this time reverse the *order* of the lines in addition to the lines themselves.

   For example, if the contents of the file are:

   ```
    The quick brown fox jumps over the lazy dog.
   Pack my box with five dozen liquor jugs. (These are all
   pangrams.)
   ```

   Then the new contents should become:

   ```
    ).smargnap lla era esehT(
   .sguj rouqil nezod evif htiw xob ym kcaP .god yzal eht
   revo spmuj xof nworb kciuq ehT
   ```

8. Suppose we have the following code.

   ```
   f = open("input.txt", "r")
   for line in f:
       line = f.readline()
       print(line)
   f.close()
   ```

   And, suppose **input.txt** contains each capital letter of the English alphabet on a single line, like this:

   ```
    A
   B
   ```

```
C
D
E
F
... and so on through 'Z'
```

What is the output of this code? Explain why.

9. Convert the following twos-complement 8-bit binary numbers into decimal.

   $0001\ 0000_2$

   $0000\ 1111_2$

   $0001\ 0100_2$

   $0010\ 1000_2$

   $0011\ 1111_2$

   $1011\ 1111_2$

   $1001\ 1001_2$

   $1000\ 0001_2$

10. Convert the following decimal numbers to twos-complement 8-bit binary numbers (use leading zeroes if necessary).

    15_{10}

    85_{10}

    102_{10}

    -102_{10}

    -128_{10}

    -127_{10}

11. Add the following binary numbers using binary arithmetic. Show your work, especially anytime you carry a one.

0101 0101_2

0001 1101_2

12. Consider the following code. What output does it produce? Use asciitable.com to figure it out.

```
print("%s%s%s" % (chr(77), chr(86), chr(80)))
```

Chapter 7

Dictionaries

7.1 Creating and using dictionaries

We can do a lot of things in code using what we know so far. Some of the things we can do in code is kind of, well, "clunky." Consider the following example. Suppose we want to keep track of a group of students and their major field of study. The only way we know how to do that currently is with lists.

```
names = ["Shayla Succotash", "Amy Avocado", "Larry Linguini", "Kevin Kumquat"]
majors = ["Comp Sci", "Accounting", "Math", "Comp Sci"]
```

We are using lists to store corresponding attributes of each student. Each student has a unique index for their attributes. Suppose we ran the code that follows.

```
print(names[1])
print(majors[1])
print()
print(names[3])
print(majors[3])
```

The resulting output of this code is:

```
Amy Avocado
Accounting

Kevin Kumquat
Comp Sci
```

This is all fine, but it's also a bit awkward. If we want to know a student's major, we have to find their location in the first list, and then use their index to look up their major in the second list, like in Listing 7.1.

Listing 7.1

```
x = "Kevin Kumquat"
found = False

for i in range(0, len(names)):
    if names[i] == x:
        print(majors[i])

if not found:
    print("Couldn't find a major for %s." % x)
```

Yuck. What we really want to do is something like this:

```
x = "Kevin Kumquat"
m = get_major(x)
# m is now "Comp Sci"
```

We could easily define a function like **`get_major`**, but fortunately we don't have to.

To make this easier, we will forgo lists (for now) and use a *dictionary*. At first glance, a dictionary is just like an English dictionary. With an English dictionary, we look up a word (which is a string) and next to the word is its definition (also a string). Python dictionaries can be used on more than just strings, but we'll focus on strings for now. We can apply this concept of an English dictionary to, say, look up **`"Amy Avocado"`** and receive **`Accounting`**.

Let's create a dictionary to *associate* names with majors. Here is how we do it (see Listing 7.2).

Listing 7.2

```
majors = {
    "Shayla Succotash" : "Comp Sci",
    "Amy Avocado" : "Accounting",
    "Larry Linguini" : "Math",
    "Kevin Kumquat" : "Comp Sci",
}
```

Now, we can *look up* majors by typing this:

```
print(majors["Shayla Succotash"])
print(majors["Larry Linguini"])
```

This produces:

```
Comp Sci
Math
```

Notice how we *create* a dictionary. We use curly braces to denote the dictionary type (which is called **`dict`** kind of like other types such as **`str`** and **`int`**). We use colons **`:`** to separate the name from the major. Also, notice how we use dictionaries to *look up* values. We use square brackets, which looks a lot like how we used lists.

Terminology time! The values on the left side of the **`:`** in our dictionary definition (e.g., **`"Shayla Succotash"`**, etc.) are called *keys*. The values on the right (e.g., **`"Comp Sci"`**, etc.) are simply called *values*. Each entry in the dictionary that associates a name with a major is called a *key-value pair*. **`"Larry Linguini"`** and **`"Math"`** is an example of a key-value pair.

Okay, so now what? Well, in this book we often ask questions that start with "What happens if…" or "What happens when…?" to enhance our understanding of a topic. Here's one: What happens if we try to look up something that doesn't exist in the dictionary? Can we create an example of this in code? You try it without looking at the next paragraph.

Were you able to come up with an example? If so, did it look like Listing 7.3?

Listing 7.3

```
print(majors["Brooklyn Broccolini"])
```

Clearly, there is no key for **"Brooklyn Broccolini"** defined in **majors** in Listing 7.2. What happens when we run the Listing 7.3 code?

Did you try it? What did you find out?

If we try to specify a key that does not exist in the dictionary, the code produces a **KeyError**. We could handle **KeyError** if we so chose, like the code example found in Listing 7.4.

Listing 7.4

```
try:
    student = "Brooklyn Broccolini"
    print(majors[student])
except KeyError:
    print("There is no major listed for %s." % student)
```

If we didn't want to rely on handling **KeyError**s, we could check first to see if a certain key exists in a dictionary (see Listing 7.5).

Listing 7.5

```
1 if "Brooklyn Broccolini" in majors:
2     print(majors["Brooklyn Broccolini"])
3 else:
4     print("No major listed for that student.")
```

Even though line 2 could technically raise a **KeyError**, the **if** statement in line 1 checks first to make sure line 2 won't raise a **KeyError**.

To see a different example, let's return to the idea of a Python dictionary as being similar to an English dictionary. In an English dictionary, we can look up a word and find its definition. The word is a key. The definition is the word's value.

We will now write a program that creates a very small English dictionary that allows the user to look up definitions of words (see Listing 7.6).

Listing 7.6

```
d = {
    "alpaca" : "A domesticated camelid",
    "bow" : "A weapon that shoots arrows",
    "rose" : "A prickly bush that produces fragrant flowers",
}

word = input("Enter a word to lookup (enter nothing to quit): ")
while word != "":
    if word in d:
        print("Definition:", d[word])
    else:
        print("There is no definition for '%s'" % word)

    word = input("Enter a word to lookup (enter nothing to quit): ")
```

This is lovely, but English dictionaries often have more than one definition for a word. In Listing 7.6, the key **"rose"** is a homophone. The word "rose" could also mean "the past tense of rise."

It appears that we want a key to have multiple values, potentially. If we wanted to store multiple values in the past, we have relied on **list**s. So, we shall use **list**s once again. Let us modify the definition of **d** in Listing 7.6 so that the values are no longer strings, but rather lists of strings (that is, a **list** of **str**).

```
d = {
    "alpaca" : ["A domesticated camelid"],
    "bow" : ["A weapon that shoots arrows",
             "To bend at the waist",
             ],
    "rose" : ["A prickly bush that produces fragrant flowers",
              "The past tense of rise",
              ],
}
```

The expression **d["rose"]** now yields a **list** of two items: **"A prickly bush ..."** and **"The past ..."**. Because the expression **d["rose"]** produces a **list**, I can do **list**-things to it. For example, the expression **len(d["rose"])** produces the **int** value **2**, and the expression **d["rose"][1]** produces the **str** value **"The past tense of rise"**.

When we look up a word the user enters, we can use a **for** loop to show each of the definitions. See how we have modified Listing 7.6 to create Listing 7.7.

Listing 7.7

```
1  d = {
2      "alpaca" : ["A domesticated camelid"],
3      "bow" : ["A weapon that shoots arrows",
4               "To bend at the waist",
5               ],
6      "rose" : ["A prickly bush that produces fragrant flowers",
7                "The past tense of rise",
8                ],
9  }
10
11 word = input("Enter a word to lookup (enter nothing to quit): ")
12 while word != "":
13     if word in d:
14         for defn in d[word]:
15             print("Definition:", defn)
16     else:
17         print("There is no definition for '%s'" % word)
18
19     word = input("Enter a word to lookup (enter nothing to quit): ")
```

There's an interesting tidbit in Listing 7.7 you may have missed. Note lines 14 and 15. My initial choice for a variable name was **def**. However, **def** is a reserved keyword in Python, which we know is used to define functions. So, I had to chose another variable name. Instead, I chose **defn** to stand for "definition."

What if we want to remove a key-value pair from a dictionary? We can use the **del** keyword to accomplish this. Consider Listing 7.8, specifically line 13.

Listing 7.8

```
1  d = {
2      "alpaca" : ["A domesticated camelid"],
3      "bow" : ["A weapon that shoots arrows",
4               "To bend at the waist",
5               ],
6      "rose" : ["A prickly bush that produces fragrant flowers",
7                "The past tense of rise",
8                ],
```

```
}

word = input("Enter a word to remove from the dictionary: ")
if word in d:
    del d[word]
else:
    print("Sorry, %s doesn't seem to be defined in our dictionary." % word)
```

If the user were to type **alpaca**, the pair for the key **"alpaca"** would be removed. Only the pairs for the keys **"bow"** and **"rose"** would remain. (Readers will note that removing alpacas from the dictionary would be a terrible calamity because they are adorable and charming. Llamas, on the other hand, are jerks.)

7.2 Iterating through dictionaries

There may be instances where we wish to iterate through a **dict** similar to how we might traverse a **list**. Suppose have the following dictionary definition.

```
calories = {
    "egg" : 80.0,
    "milk" : 80.0,
    "cheerios" : 100.0,
    "blueberry" : 0.78,
    "strawberry" : 4.0,
}
```

calories is a **dict** that acts as a food database. It allows programmers to look up the calories for a given food. Thus, **calories["strawberry"]** gives us **4.0**. Suppose we want to see all the foods in our database. We might write the following code.

```
for food in calories:
    print(food)
```

When a **for** loop operates on a dictionary, the value assigned to the variable **food** is actually the *key*. The output of this code is each of the keys (the food names), one on each line.

Now, run this code. Run it several times. What do you notice? In all likelihood, the ordering changes for the food names. We cannot rely on the keys to be kept in order when we use a dictionary. If we want the keys to be sorted alphabetically, we would need to do something like what we see in Listing 7.9.

Listing 7.9

```
for food in sorted(calories.keys()):
    print(food)
```

`keys()` creates a `list` of the keys in `calories`. `sorted` then returns a new `list` with the keys in alphabetical order.

If you're curious why the keys are ordered differently when you iterate through them using a loop, you'll learn more when you take a class on data structures and learn about *hash tables*. Hash tables are used to make dictionaries. (Very curious readers should look on YouTube for a video from PyCon 2010 titled "Mighty Dictionary." This will give you an idea how hash tables work and how they organize the internals of a dictionary.)

7.3 Exercises

1. Given the following `dict` definition

   ```
   d = {"a" : 1, "b" : 2, "c" : 3}
   ```

   write the type and value of each of the following expressions. If there is an error, state the error instead.

   a. `d`

   b. `d["b"]`

   c. `d[2]`

   d. `d["x"]`

   e. `"c" in d`

   f. `3 in d`

2. Given the following `dict` definition

   ```
   d = {"a" : 1, "b" : 2, "c" : 3}
   ```

   write code that adds a new key `"d"` and gives it the value `4`, and then modifies the key `"c"` and gives it the value `"radish"`.

3. Write a function that stores a `dict` to a file. The function's name should be `dict_to_file` and it should have two parameters. The first parameter should be the `dict`. The second parameter should be the new file's name. Assume that each key is a `str` and each value is also a `str`. The format of the file should be each key on a line followed by its value on the next line.

4. Write a function that creates a `dict` and fills its contents using lines from a file. The function's name should be `file_to_dict` and it should have one parameter: the name of the file. The function should return the `dict` it creates. Assume the file's format consists of a key on a single line followed by the key's value on the line after it. Also assume both the key and the value are strings.

5. Write a function that stores a `dict` that maps strings to lists into a file. The function's name should be `str2list_to_file` and it should have two parameters. The first parameter should be the `dict`. The second parameter should be the new file's name. Assume that each key is a `str` and each value is a `list`. The format of the file should be each key on a line followed by a series of lines, where each line is an item in the `list`. The lines that are keys should be prefixed with the string `"K:"`. This will tell us which lines are keys and which are values.

   For example, suppose we define

   ```
   names = { "Anderson" : ["Steve", "Tyler"], "Smith" : [Susie, Aaron] }
   ```

   calling `str2list_to_file(names, "names.txt")` should create a file named `names.txt` that has the contents

```
 K:Anderson
Steve
Tyler
K:Smith
Susie
Aaron
```

Chapter 8

Searching and Sorting

8.1 The Sorting Problem

Do you stream music online? Do you click on a folder on your desktop to find and organize your files? Have you ever used Microsoft Excel to keep information in a spreadsheet? If we think for a moment about these software programs, we might realize that they have some features in common. One we might notice is that they eventually allow a user to *sort* information. In a music streaming app, you might choose to order your songs by title or by artist to make certain songs easier to find. When you're looking at a folder full of files, you might order your files by file name or by the date it was last modified. In Excel, we can sort a column of data, which has the effect of rearranging the rows into a different order.

Sorting is a very common thing that programs do, and it is an interesting problem that can be tackled in a lot of different ways. In this introductory book you're reading right now, we will look at only one of those ways. In doing so, we'll also get better at walking through a problem logically in order to write a program that solves a problem. The end result will be that we improve our natural problem solving abilities, too. If you continue learning about computer science through other courses and other books, you'll encounter other ways to perform sorting, some of which have really interesting capabilities. Well, what are we waiting for?

8.2 Towards an Algorithm for Sorting

In this section, we will start developing an algorithm for sorting. An *algorithm* is a series of well defined steps for solving a problem that has an input, an output, and completes in a reasonable amount of time. (The meaning of "reasonable" in this context is discussed in greater detail in more advanced computer science texts.)

Suppose we have a list of numbers, defined as such.

```
numbers = [8, 4, 3, 9, 5]
```

If we were to sort this list into *ascending* order, it would look like this.

```
numbers = [3, 4, 5, 9, 8]
```

A thoughtful reader might ask what happens if we have duplicate numbers. In other words, what if **`numbers`** were defined like this?

```
numbers = [3, 2, 3, 1, 4]
```

Note that we have two 3's in our list. The sorted version would look like this.

```
numbers = [1, 2, 3, 3, 4]
```

The two 3's remain preserved in the sorted list. We certainly wouldn't want to lose items, so this makes sense. We used the word *ascending* to describe how we sorted the numbers, but that's not technically correct since the list does not "ascend" between the first and second 3. A more correct phrase would be to describe the sort as *monotonically increasing*. We should use that phrase from here forward because 1.) it's technically correct, and 2.) it sounds cooler.

So, how do we sort these numbers? In other words, how do we change the positions of the numbers in the list without losing any of the numbers?

Often, the best way to solve a problem is to think of a very simple instance of that problem, and then invent a simple solution. If we can find the smallest number in the list, where in the list should it go? The smallest number should always come first in the list, right?

Suppose we have the following list.

```
numbers = [8, 4, 3, 9, 5]
```

3 is the smallest number, so 3 should move to position 0, the start of the list. However, if we're going to move the 3 to position 0, we need to find a new home for the 8. So, we'll put 8 where the 3 was.

```
# Before
numbers = [8, 4, 3, 9, 5]
# After
numbers = [3, 4, 8, 9, 5]
```

None of the other numbers are touched in moving the 3 and the 8. Exchanging positions for the 3 and the 8 is called *swapping* the numbers. In fact, you learned how to swap values in a list back in Chapter 5. Now would be a good time to go back and view Listing 5.4 and Figure 5.2 in Section 5.1 again.

Now that the smallest item has been moved into position 0, we can do the same thing with the next smallest item. Where should the next smallest item go? In other words, what position should hold the next smallest item? If you said position 1, you're correct!

```
# Original list
numbers = [8, 4, 3, 9, 5]
# Moving the smallest to position 0 by swapping.
numbers = [3, 4, 8, 9, 5]
# Moving the next smallest to position 1 by swapping.
numbers = [3, 4, 8, 9, 5]
```

Oh dear! Nothing changed! Actually, nothing needed to change. The 4 was already at its desired position. The computer doesn't need to notice this. We

don't need to check to see if they swap needs to happen. We can always perform the swap, because we can swap a number with itself with no unfortunate side effects.

We can continue this procedure to move each subsequent number to the correct position in the list. When we have one final item remaining, we know that number is the biggest the number, and it should simply remain at the end of the list.

In each step `x`, we swap the next smallest value to position `x`. We scan across from position `x` to the end of the list, looking for and keeping track of the position of the smallest. When we are done scanning, we perform the swap. We can see what this procedure looks like in Figure 8.1. We've underlined the eventual position of smallest item that we have selected and swapped. In fact, this sort is named *selection sort* because of how we select the smallest item in each step.

```
numbers:  [8, 4, 3, 9, 5]
           0  1  2  3  4

numbers:  [3, 4, 8, 9, 5]
          [0] 1  2  3  4

numbers:  [3, 4, 8, 9, 5]
           0 [1] 2  3  4

numbers:  [3, 4, 5, 9, 8]
           0  1 [2] 3  4

numbers:  [3, 4, 5, 8, 9]
           0  1  2 [3] 4
```

Figure 8.1

There are a couple of things to note in the example shown that will help us write actual code. This is often how we come up with code that solves problems. We make up an example to help us think more concretely, and then we use that concrete example to help us see patterns that become our code. In Figure 8.1,

you can see that we are starting each scan to find the smallest item at positions 0, 1, 2, and finally 3. 3 is one less than the last index in the list. We will need a loop to perform the scan to select the smallest item, and we will need another loop (outside of the scan loop) to control where we start each scan.

Try to write the code yourself before going on to the next section. Don't cheat and look ahead. Make a focused attempt, hand-written on paper before you proceed.

8.3 The Selection Sort

Let's walk through the advice given at the end of Section 8.2. Rather than name our list **numbers**, we'll name it **L** (hey, fewer keystrokes!).

In each step of the selection sort, we scan across the list from beginning to end looking for the smallest item. We want to note the position/index where the smallest item sits (we'll call it **smallpos**).

```
1 smallpos = 0
2 for i in range(1, len(L)):
3     if L[i] < L[smallpos]:
4         smallpos = i
```

This code assumes we start looking for the smallest at index 0. We let index 0 be the first smallest, and then we scan across from index 1 onward. However, we don't want to start at index 0 every time. We want to start at index 0 to find the first smallest item, then we want to start at index 1 to find the next smallest, and so forth. So, the starting position needs to become a variable since the starting position needs to change each time. We'll named that variable **start**. Notice how the code changes on lines 1 and 2 of the code.

```
1 smallpos = start
2 for i in range(start+1, len(L)):
3     if L[i] < L[smallpos]:
4         smallpos = i
```

Now that we have a **start** variable to control the starting position of each scan through the list, we need a loop to update it in each step of the algorithm.

```
1 for start in range(...):
2     smallpos = start
3     for i in range(start+1, len(L)):
4         if L[i] < L[smallpos]:
5             smallpos = i
```

What should we put in **range** in line 1? The first value of **start** should clearly be 0. What about the last value? From looking at Figure 8.1, we want the last time through the loop to be the next to last index. Therefore, the expression that ends the loop should be **len(L)-1**, like in the following code.

```
1 for start in range(0, len(L)-1):
2     smallpos = start
3     for i in range(start+1, len(L)):
4         if L[i] < L[smallpos]:
5             smallpos = i
```

This code now does successive scans across the list, but it doesn't actually swap any values. Once we're finished with the scan, but before we do another scan, we need to swap values between the positions stored in **start** and **smallpos**. The final solution is shown in Listing 8.1.

Listing 8.1: Selection Sort

```
for start in range(0, len(L)-1):
    # Find the smallest value and keep track of its position.
    smallpos = start
    for index in range(start+1, len(L)):
        if L[index] < L[smallpos]:
            smallpos = index

    # Swap the smallest value to the correct location,
    # i.e., where we started our walk through the list.
    temp = L[start]
    L[start] = L[smallpos]
    L[smallpos] = temp
```

This code works for any list we name **L** that has values that can be compared with the **<** operator. If we fill **L** with numbers, this code works. If we fill **L** with strings, it also works (can you guess how it orders strings?).

We can take this one step further and wrap it into a function that we can call. See Listing 8.2.

Listing 8.2: Selection Sort as a Function

```
def selsort(L):
    for start in range(0, len(L)-1):
        # Find the smallest value and keep track of its position.
        smallpos = start
        for index in range(start+1, len(L)):
            if L[index] < L[smallpos]:
                smallpos = index

        # Swap the smallest value to the correct location,
        # i.e., where we started our walk through the list.
        temp = L[start]
        L[start] = L[smallpos]
        L[smallpos] = temp
```

Then, we could call the function like this.

```
numbers = [3, 4, 1, 5, 2]
names = ["Becky", "Allen", "Derek", "Casey"]
selsort(numbers)
selsort(names)
print(numbers)  # Output is [1, 2, 3, 4, 5]
print(names)    # Output is ["Allen", "Becky", "Casey", "Derek"]
```

8.4 Searching for Items in Sorted Data

Human beings can find things easier in a list that has been sorted because they can eliminate sections of the list as they quickly visually scan its contents. Computers, too, can more easily find items in a list that has been pre-sorted.

Consider a long, unordered list of names. In order to determine if the name "Ned" exists in the list using a computer program, we would need to linearly

scan the list from beginning to end. The number of comparisons we would need to do, in the worst case, would be equal to the number of items in the list.

Suppose, however, that the list is pre-sorted. We can reduce the number of comparisons between examining the middle item. If the name we seek comes before that middle item, we know that if the name exists in the list, it must exist in the first half of the list. Conversely, if the name we seek comes alphabetically after that middle item, we know that if the name exists in the list, it must exist in the latter half of the list. In that first step, we have eliminated *half* of all the items. This can speed up our search dramatically. Keep in mind that if the middle item is the name we were looking for in the first place, then we have found the name in only one comparison.

The algorithm we just described is called a *binary search*. To invent code that performs a binary search, let's take the a similar approach to what we did with selection sort. We will start with an example, and observe patterns that we can turn into code.

Suppose we have a pre-sorted list `L` defined in the following way.

```
L = [2, 4, 7, 10, 13, 18, 25]
```

The middle item in this list happens to be `10`, found at index `3`. We know that `3` is the middle index because `len(L) == 7` and `7 // 2 == 3` (recall that the `//` operator does integer division… it divides and then drops the fractional part).

Let's try to find `18` in `L`. We compare `18 > 10` to find that `18` is greater than the middle. This means that if the `18` is in our list, it must be in the right half of the list. So, we now only need to look at the sublist `[13, 18, 25]`. We've reduced the number of items left to check to half!

So, how do we proceed from here? How do we keep track of which part of the list we're currently examining. Well, as always, if we want to keep track of something or somethings, we use variables. Let's let `i` and `j` be index variables that hold the positions of the leftmost item and rightmost item respectively. Let's also create a variable `m` to calculate the index of the middle item. Figure 8.2 demonstrates the steps that will find `18` in `L`. Observe how `i`, `j`, and `m` change in each step.

```
      L:  [2,  4,  7, 10, 13, 18, 25]
           0   1   2   3   4   5   6
Step 1:    i           m           j
Step 2:                        i   m   j

                    Found 18 at m == 5!
```

Figure 8.2

If `18` is greater than the middle (that is, if `18 > L[m]`), then we move `i` to the right of `m` (that is, `i = m + 1`). What would be do if `18` were less than the middle? Then we would move `j` instead by moving it to the left of `m` (that is, `j = m - 1`).

We can see that in two steps, binary search is able to find the `18` in our example list. That's fast! With a normal scan/search, it would have taken us 6 comparisons.

Wait a minute. What happens if we apply these same steps to look for a number that doesn't exist in `L`. How about `17`? Figure 8.3 walks us through that scenario.

```
      L:  [2,  4,  7, 10, 13, 18, 25]
           0   1   2   3   4   5   6
Step 1:    i           m           j
Step 2:                    i   m   j
Step 3:                   ijm
Step 4:                    j   i

                    i and j have "crossed",
                    so 17 is not in L.
```

Figure 8.3

We are left with the idea that we will probably need to use a loop that con-

tinues to look for our target number, and the job of the loop's body will be to either move `i` or `j` in each step. Terminal conditions for the loop are:

- We found our target number, or
- `i` and `j` have "crossed" (that is `i > j`), so our target number cannot be found

Okay, now its your turn. Can you attempt to construct the code for a binary search? Once the code is done, `m` should be set to the index of the found target number. If we don't find the number, let's set `m` to `-1`. Remember what we just said. We need a loop whose job (the body of the loop) is to move `i` and `j` and calculate `m`. The terminal conditions for the loop are given above. There may not be one single right way to write this code, so don't worry about trying to do the one right way. Play around with it for a while on paper first. Try your best before looking ahead.

How far did you get? Where did you get stuck? You can learn a lot about how you solve problems and what types of things get you stuck by doing exercises like this.

Here is one possible solution.

```
x = 18   # x will be our target number
i = 0
j = len(L) - 1

while i <= j:
    # Find the middle index.
    m = (i + j) // 2
    if x < L[m]:
        # Look at the left half of L
        j = m - 1
    elif x > L[m]:
        # Look at the right half of L
        i = m + 1
    else:
        # Found it
        break

if i > j:
    m = -1
```

Note that `x` is not less than the middle and not greater than the middle, then it must be equal to the middle.

We can easily place this code into a function that we could call to find items in a list. We can pass to the function two values: 1.) the list `L`, and 2.) the target item `x`. The function should return the correct value of `m`, which should be `-1` if we didn't find the target item, or some integer `>= 0` if we did find it. See Listing 8.3.

Listing 8.3: Binary Search as a Function

```
1  def binsearch(L, x):
2      i = 0
3      j = len(L) - 1
4
5      while i <= j:
6          # Find the middle index.
7          m = (i + j) // 2
8          if x < L[m]:
9              # Look at the left half of L.
10             j = m - 1
11         elif x > L[m]:
12             # Look at the right half of L.
13             i = m + 1
14         else:
15             # Found it.
16             return m
17
18     # Didn't find x in the loop.
19     return -1
```

8.5 (Optional) Using Recursion to Search

This section will be added at a later date.

8.6 Exercises

1. Demonstrate how selection sort would order the following list:

   `[27, 34, 12, 9, 13]`

Show each step of the sort, even if that step does not move an item.

2. Demonstrate how selection sort would order the following list:

   `["carrots", "peas", "broccoli", "bok choy"]`

   Show each step of the sort, even if that step does not move an item.

3. Suppose we have two lists that represent students' names and their grade point averages (GPAs).

```
names = ["Jennifer", "Alfred", "Jack"]
gpas = [4.0, 3.1, 2.7]
```

   Also suppose that the names and GPAs line up with one another. In other words, Jennifer's GPA is 4.0, Alfred's GPA is 2.7, and Javier's GPA is 3.1. If we were to sort **`names`** without also reordering **`gpas`**, Alfred's GPA would become 4.0 (which I'm sure would make him happy, but it would be wrong).

   Define a function named **`sort_all`** that takes a list of lists. The function should sort the first list and then reorder all other lists in the same way as the first list.

```
# Example 1:
names = ["Jennifer", "Alfred", "Jack"]
gpas = [4.0, 3.1, 2.7]
sort_all([names, gpas])
print(names)  # Output is ["Alfred", "Jack", "Jennifer"]
print(gpas)   # Output is [3.1, 2.7, 4.0]

# Example 2:
names = ["Jennifer", "Alfred", "Jack"]
gpas = [4.0, 3.1, 2.7]
sort_all([gpas, names]) # we are sorting by GPA this time
print(names)  # Output is ["Jack", "Alfred", "Jennifer"]
print(gpas)   # Output is [2.7, 3.1, 4.0]
```

Chapter 9

Objects, Classes, and Interactive Graphics

9.1 Getting Started

You're still reading this book?! Hurray! Good for you. Now that you're here, you'll be pleased to know that this is the best chapter of the whole book, so pay attention!

Thus far, all of our programs have been text-only programs. In this chapter, we will make programs that have graphics. To do this, we'll use a Python code library named *Pygame*. Very shortly, we'll learn how to install Pygame on your computer. Then, we'll write some simple graphical programs and eventually animate the graphics to make them move around the screen.

Cool, huh?

We will reach a point in learning about graphics where things will get a little frustrating. We will want to detect when two graphical things collide, and that will seem kind of hard to manage, especially when we have lots of graphical things flying around the screen. So, we will need to learn about things called *objects* and *classes* that will help us tremendously, and will make programming graphical video games way more fun.

Let's get started by installing Pygame. Open Thonny and find the **Tools** menu. Under the **Tools** menu you will see an option named "Manage pack-

ages…" Click that option. A dialog window will be displayed to you. The dialog has a textbox and a button at the top of its window. Enter "pygame" into the textbox and press the button (which reads "Find package from PyPI"). Information about the **pygame** package should then be displayed in the main part of the dialog window, and beneath it should be a button labeled "Install". Click the "Install" button. After pygame has installed, click the "Close" button.

How do you know if the installation worked? In the Python shell window, you should be able to type **import pygame**. If nothing bad happens, it worked! If instead that statement makes Python puke red text, it did not work. At that point, you may wish to solicit help from the nearest knowledgeable, techie human.

Assuming everything worked, it's time to write code that uses Pygame!

9.2 Basic Pygame: Drawing Graffiti

Pygame can do a lot for us as programmers. Pygame can make a separate window to display our graphical program. Pygame allows us to draw graphics on that separate window. Pygame allows us to draw different types of graphics (shapes, colors, etc.), play sounds, and use a whole host of other features.

Because Pygame does a lot, the first Pygame program you see might look a bit daunting. So, let's dive in. You can see our first Pygame program in Listing 9.1, and if you'd like to avoid typing this code or using copy/paste, you can download the code directly from this link. In the paragraphs that follow, we will break down Listing 9.1 section by section.

Listing 9.1: A basic Pygame code template

```
1  import pygame
2  from pygame.locals import *
3
4  pygame.init()
5
6  white = (255, 255, 255)
7  black = (  0,   0,   0)
8  green = (  0, 255,   0)
9
10 screenWidth = 800
```

```
screenHeight = 600
screenSize = [screenWidth, screenHeight]
screen = pygame.display.set_mode(screenSize)
pygame.display.set_caption("WINDOW TITLE HERE")

clock = pygame.time.Clock()

done = False

while not done:
    # 1. Process events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    # 2. Program logic, change variables, etc.

    # 3. Draw stuff
    screen.fill(white)
    pygame.draw.line(screen, green, [100, 200], [150, 300], 3)
    pygame.draw.line(screen, green, [150, 300], [200, 200], 3)

    pygame.display.flip()
    clock.tick(20)

pygame.quit()
```

The result of the code in Listing 9.1 is the window shown in Figure 9.1.

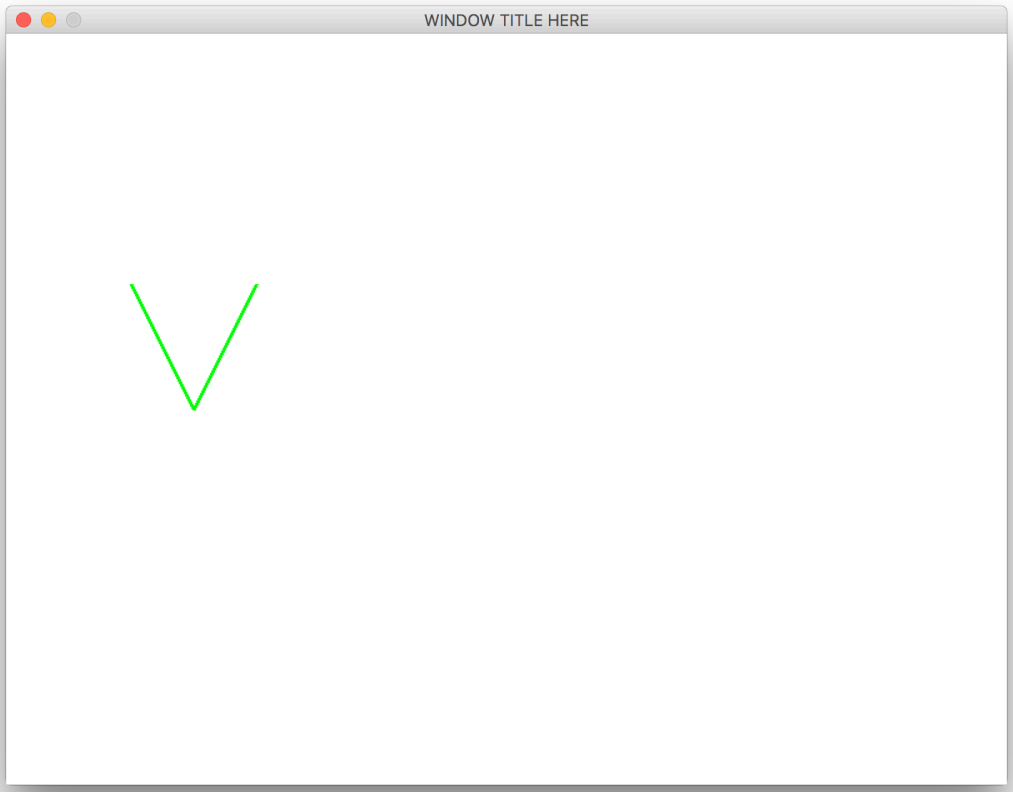


Figure 9.1: The window produced by the code in Listing 9.1

That's a lot of code to throw at someone, or at least it seems that way right now. It's actually not a huge deal. Let's take this code step by step. We'll look at a block of code and then we'll explain what it does.

```
import pygame
from pygame.locals import *

pygame.init()
```

The first two lines in this code snippet make it so we can use Pygame variables and functions in our code. The last line gets Pygame ready to start making

graphical windows.

```
white = (255, 255, 255)
black = (  0,   0,   0)
green = (  0, 255,   0)
```

In this code snippet, we define colors that we think we'll use in our program. Colors are "mixed" together using amounts of red, green, and blue (in that order, specifically). The amounts of each range from **0** to **255**. Thus, when we type **green = (0, 255, 0)**, we are telling Python to make a color that has **0** of red, lots of green (**255**, which is the maximum), and **0** of blue.

You might be wondering what the parentheses do. Note that this:

```
(0, 255, 0)
```

Kind of looks like this:

```
[0, 255, 0]
```

Notice all we did was change the type of braces – parentheses versus square brackets. We know that square brackets surround lists. If we use parentheses, it makes a list-like thing called a *tuple*. A tuple is a list that cannot be changed once it is created. We use tuples to make colors.

Sometimes students are bothered by the indentation I've used in the above code snippet. Here is the code again:

```
white = (255, 255, 255)
black = (  0,   0,   0)
green = (  0, 255,   0)
```

Notice the extra spacing around the zeroes. This extra spacing doesn't do anything special. It just makes the code look more readable. You're allowed (and encouraged) to be artistic in writing code. Make your code pretty!

```
screenWidth = 800
screenHeight = 600
screenSize = [screenWidth, screenHeight]
screen = pygame.display.set_mode(screenSize)
pygame.display.set_caption("WINDOW TITLE HERE")
```

This code actually creates a graphical window. It chooses a width and height and then uses the function **`pygame.display.set_mode`** to set up the window, which makes it appear on the screen for the first time. The part of the code that reads **`pygame.display`** is the name of a code module that contains the **`set_mode`** function. Much of the code in Pygame is stored in different modules that have descriptive names.

The last line changes the text in the title bar of the window. This text is called the window's *caption*.

```
clock = pygame.time.Clock()
```

This line of code creates a "clock" that "ticks." We'll talk more about what this **`clock`** variable is for in a little bit, but for now just remember that we created a variable named **`clock`**.

```
done = False

while not done:
```

You've seen loops a lot in this book. Often, we've used loops to repeatedly ask for a number, a string, or a value in a list. In this case, the loop is going to repeatedly draw things on the screen.

Computer programs that create graphics don't just draw those graphics on the screen and then wait. Computer monitors continually refresh their graphics on-screen, so programs need to be written to continually re-paint their shapes, colors, and images. This is the start of the loop that accomplishes this.

We create a Boolean variable named **`done`**. The loop will keep running until **`done`** is **`True`**. Presumably, something inside the loop will change **`done`** from **`False`** to **`True`**. We'll see that very shortly.

```
# 1. Process events

# 2. Program logic, change variables, etc.

# 3. Draw stuff
```

Okay, so I've removed some code so that you can focus on the three main comment sections of the loop. In fact, I've labeled them 1, 2, and 3. Each time through the loop, we're going to do three basic things.

First, we'll handle events. Events are things like the user pressing a keyboard key or moving the mouse.

Second, we'll perform program logic. If we want to change where graphics show up on the screen, or possibly check to see if there are any collisions between our graphics, we would do that in this section of code. We will use variables to keep track of where things should appear on the screen, and in this section we will update the values of those variables.

Third and finally, we'll use the variables modified in the second section of code to actually draw shapes on the screen.

It is a really good idea to stick to this organization in the loop. Consider if we decided to try to draw something on-screen in part 1. It is possible that whatever we drew would end up erased or overwritten in part 3. This would also make our code really hard to debug.

Now, let's look at the code under each of these comment sections.

```
# 1. Process events
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        done = True
```

The function **`pygame.event.get`** gives us a list of all the events that have happened to the program. This could be mouse clicks, key presses, etc. We are using a **`for`** loop to cycle through the events, one at a time. If the event happens to be a **`pygame.QUIT`** event, this tell us that the user has clicked on the "close" button typically found at the top corner of the window. We should respond by setting **`done`** to **`True`** so that the loop can exit at the top of the **`while`** loop and then the program can end.

In future Pygame programs, we will handle more types of events in different ways.

```
# 2. Program logic, change variables, etc.
```

There's no code in this section right now. This is because in this code sample, all we are doing in the next section (part 3) is drawing two lines at fixed locations. There is no real program logic.

```
# 3. Draw stuff
screen.fill(white)
pygame.draw.line(screen, green, [100, 200], [150, 300], 3)
pygame.draw.line(screen, green, [150, 300], [200, 200], 3)
```

Think of drawing on the screen in Pygame as being like painting on a wall or a canvas. When we start painting on a surface, we want to make sure the canvas is clear and uniform. If our canvas already had drawings on it and we wanted to start over, perhaps we could paint over the existing drawings using one background color. That is what we accomplish in the first line with **`screen.fill(white)`**.

Each of the next two lines of code draw a single line using the **`green`** color on the **`screen`** variable we created earlier. The **`pygame.draw.line`** function accepts five arguments. The first is the **`screen`**. The next is the color. The third and fourth arguments are lists, each of which represents a point on the screen. The points are the ends of the line segment we wish to draw. Each list is of the form **`[X, Y]`** where **`X`** is the x-coordinate and **`Y`** is the y-coordinate. However, the x's and y's work a little differently than what you might be used to in past mathematics courses or other experiences in life.

FIXME Figure here showing x's getting larger as we go down from the origin point

```
pygame.display.flip()
clock.tick(20)
```

```
pygame.quit()
```

9.3 Objects and Classes

9.4 Pygame Sprites