Introduction to Creative Computing

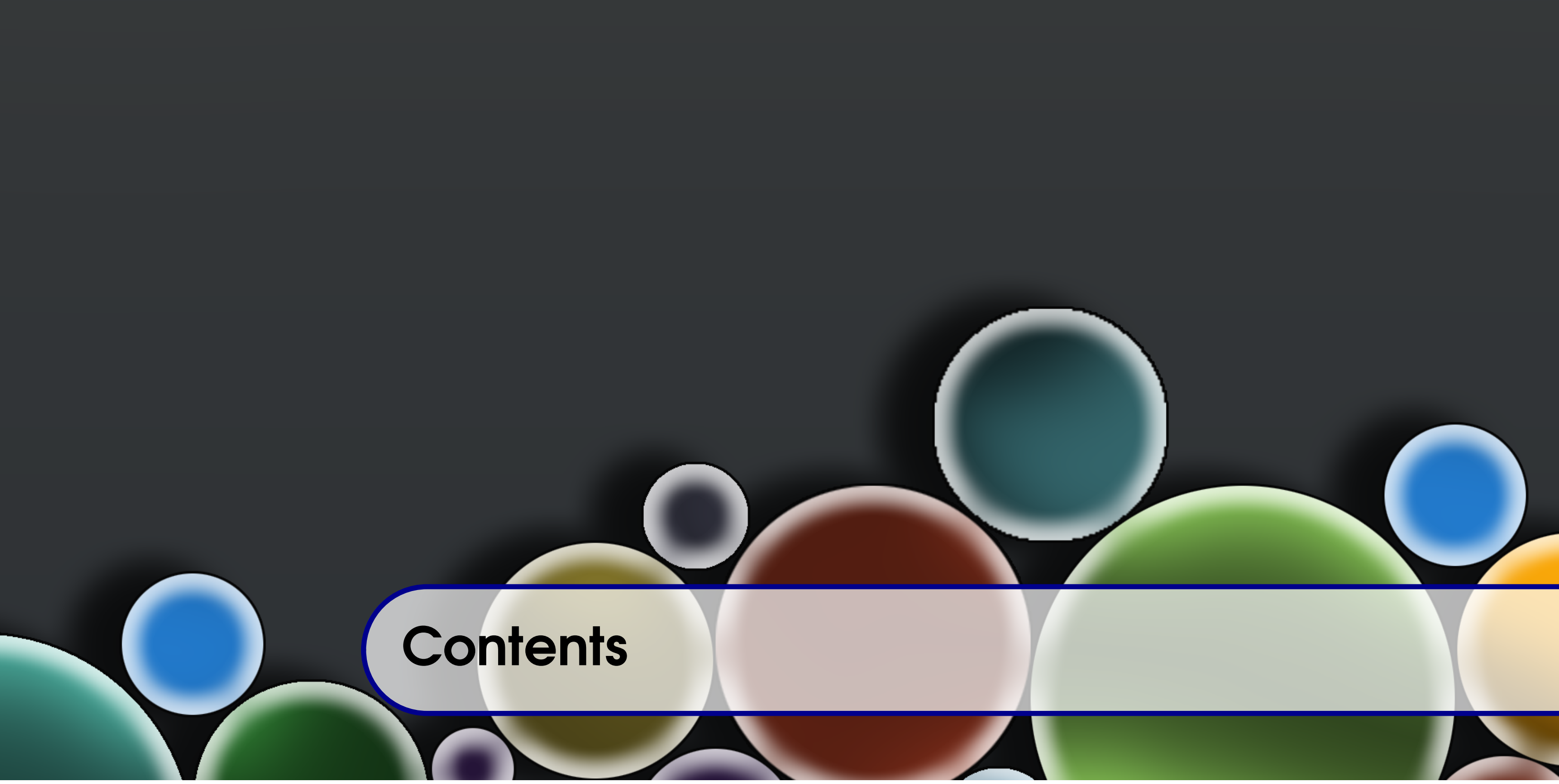
Jeff Long

Course readings for CMPT 140

PRODUCED BY THE AUTHORS FOR STUDENTS IN CMPT 140.

CS.USASK.CA

First Edition, September 2016



Contents

Part I Programming in Python

Part I

Programming in Python

1 — Algorithms

Learning Objectives

After studying this chapter, a student should be able to:

- describe what an algorithm is
- provide examples of algorithms
- distinguish between actions and control statements in algorithms
- identify appropriate algorithms for humans versus computers
- distinguish between problems and algorithms

1.1 Algorithms

An *algorithm* is an ordered list of actions that describe how to perform a task or solve a problem. Algorithms are an important concept in the study of computer science, but they are broadly applicable. A recipe for making bread is an algorithm, even though making bread has nothing to do with computers. The recipe describes what actions you must take, and the order in which you must take them, if you want to end up with something that looks and tastes like bread. If you deviate from the algorithm, there's a good chance you end up with something quite un-bread-like. Other examples of algorithms are:

- instructions for assembling a bookshelf
- steps to operate a coffee maker
- a list of things to do in case of a fire

One note of caution, however: not everything that looks like a list or or a set of instructions meets the standard for being an algorithm. For example, a recipe that consists only of a list of ingredients is not an algorithm, because it does not specify what should be done with those ingredients. Another way that a set of instructions can fall short of being an algorithm is if the actions are not explicitly ordered. There are many real-world problems that have subtasks that can be performed in any order.

For instance, if you are assembling a table, at some point you have to attach each of the four legs, but which leg you start with doesn't matter. Humans are often pretty good at figuring out when this is the case, and therefore might omit an ordering; but if they do, then the result is not an algorithm. Note that this isn't a bad thing! Recipes that list only ingredients and instructions that leave out irrelevant ordering can be perfectly useful. They're just not algorithms.

1.2 An Algorithm in Detail

Let's take a look at a complete, concrete example of an algorithm. The following is an algorithm to make ramen noodles:

```
Algorithm MakeRamen:

boil water
add noodles to water
wait 6 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

This algorithm consists of six actions to solve the problem of making ramen noodles. The actions are taken in the order given, and the end result, or *output*, of the algorithm is a prepared bowl of steaming hot noodles, ready to eat. The important thing to remember about algorithms is that the given actions must be taken in the given order, otherwise we are not following the algorithm and there is no guarantee that we will get the desired output.

1.3 Actions in Algorithms

We have said that an algorithm is made up of a sequence of *actions*, so we should say a little more on what an action actually is. For our purposes, an action should:

- be declarative: it is a command to *do* something
- be feasible: the algorithm's recipient has the ability to perform it
- be self-explanatory: the algorithm's recipient knows how to perform it without further elaboration

From the criteria above, it should be apparent that an algorithm's recipient - that is, the person or thing that we expect will be following the algorithm - is of critical importance. For example, `raise your right hand` is a perfectly reasonable action for most adult humans. However, for a toddler who does not yet know right from left, it is not an appropriate action, because it is not self-explanatory. For a computer, it is not even a feasible action, since computers do not typically have hands at all!

1.4 Control Flow of Algorithms

Algorithms can contain more than just actions to take; they can also contain information on *when* to perform a given action, or even *how many times* to perform an action. Consider the following (very simple) algorithm for tightening a bolt.

```
Algorithm TightenBolt:

clamp wrench to bolt
while the bolt is not tight:
    rotate the wrench clockwise
unclamp and put away wrench
```

Notice that the line `while the bolt is not tight` is not in fact an instruction to do something; rather, it is indicating the condition under which we should continue to perform the action on the following line, namely, rotating the wrench. We will call lines such as the former a *control statement*. Control statements are very powerful, for they allow us to create algorithms that take only a few lines of text to write yet can describe complex behaviour.

Notice also that in the algorithm above, the line `rotate the wrench clockwise` is indented. This is to visually indicate that it is this action that should be repeated so long as the bolt is not tight - it is the line to which the preceding control statement applies. By contrast, the line `unclamp and put away wrench` is not indented. This is to show that this action should only be done once (when we are done tightening the bolt), and that this action has nothing to do with the earlier control statement. With algorithms written for humans, this sort of thing is often obvious from context; for example, a human intuitively knows that putting away a wrench multiple times would be rather silly. Once we start dealing with computers, however, we have to be very precise about the scope of control statements so that the computer knows which action(s) are covered by the control statement and which are not.

We can, of course, have multiple statements under the scope of the same control statement, as shown in the following example:

```
Algorithm MakeHamburgers:

read customer's order
check how many hamburgers are required
while more hamburgers are needed:
    slice hamburger bun
    spread ketchup on bun
    put meat on bun
    put hamburger in bag
give bag of hamburgers to customer
```

Here we have multiple statements at the same level of indentation and subject to the same control statement. We call a collection of statements like this a *block*, and it's the indentation that tells us which statements belong in the block. As a result, when reading algorithms, we should be aware that indentation is probably not arbitrary or accidental, but rather conveys important meaning.

1.5 Methods of Writing Algorithms

Algorithms can be written in different forms. So far we've seen a few algorithms that are written in words. Algorithms written in words are called *pseudocode*. Pseudocode can look like the "code" we would write in a programming language, but is much more flexible because its syntax and form is not rigidly specified like that of a programming language.

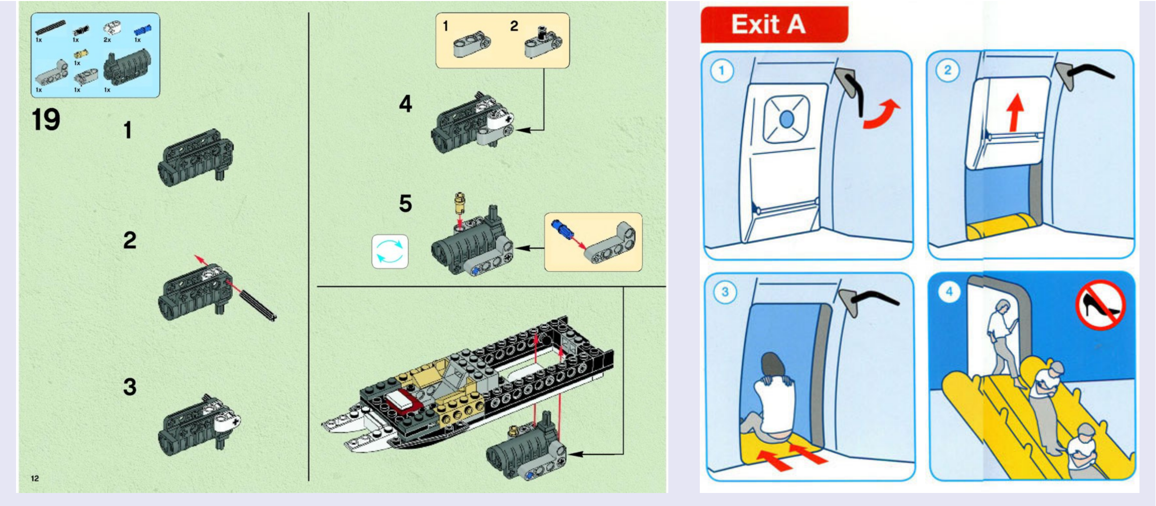


Figure 1.1: Examples of algorithms written using pictures. Left: LEGO instructions; right: aircraft safety procedures card.

The LEGO instructions and the airline safety card pictured in Figure 1.1 are examples of algorithms written using pictures. They indicate, in a step-by-step manner, what to do to complete the tasks of building the LEGO model and escape the aircraft in an emergency.

Words and pictures are normally how we write algorithms that are to be understood and/or carried out by humans. When we write algorithms for computers we have to use a language that a computer can understand. Computer programs are written in a *programming language*. A programming language provides instructions to the computer in a way that it can both understand them and carry them out unambiguously. For this reason, programming languages are more restrictive in the syntax and style we can use to write algorithms. Many hundreds of programming languages exist, but for this course, we will just be learning one (a language called *Python*).

If a computer program doesn't do what its programmer wants it to, it's not the computer's fault! The computer can only do exactly what the program tells it to do whether or not it is what the programmer intended. This is why it is vital that a programmer understand the algorithm that they are trying to write. If a programmer doesn't understand the algorithm, the chance that they'll be able to tell the computer how to perform the algorithm correctly is slim to none. This is why it is advantageous for programmers to write algorithms using pseudocode first. It helps them to ensure they understand what they are about to program without having to worry about the details of the programming language syntax. Once understanding is reached via pseudocode, a programmer is much more easily able to get the details right when writing the algorithm in the precise syntax required by the programming language. Moreover, they are ready to implement the algorithm in **any** programming language that they know!

1.6 Problems vs. Algorithms

We'll conclude this chapter by making a distinction between problems and algorithms. A *problem* is a task to be carried out. An *algorithm* is a specific set of steps for **how** to carry out a task. A problem may have more than one algorithm for solving it. A given algorithm, however, solves only one problem. This might seem like an easy distinction at this point, but during an introductory course like this one, it can be easy to forget. Because this is a course for novices, much of the work you will be given to practice your programming basics will be so simple that the problem statement and the algorithm for solving it are more or less one and the same. It's therefore important to remember that for most real-world problems, this is rarely the case.

As an example, let's consider the problem of picking up a pile of playing cards that you were just dealt, and putting them in rank-order. Some people pick up their cards one at a time and place each card into their hand at the correct position as they do so. Other people pick up all of the cards at once, find the smallest one and move that card to the left-most position, then find the next smallest and put it next to the smallest card, and so on. These are two different algorithms for solving the problem of putting a hand of cards in order. Both achieve the same result, but the algorithms themselves are fundamentally different processes.

2 — Abstraction

Learning Objectives

After studying this chapter, a student should be able to:

- define abstraction, refinement and encapsulation
- write algorithms at different levels of abstraction
- describe how input and output relate to encapsulation
- employ encapsulation to decompose algorithms written in pseudocode

2.1 Abstraction

Abstraction is the process of strategically removing details that are not relevant with regard to a particular goal. Abstraction, like the algorithms we discussed in the previous chapter, can be found everywhere, not just in computer science. A building floor plan is an abstraction: it omits details such as the materials the walls are made of and the location of wires and plumbing because these things are not relevant to the purpose of the floor plan, which is to help people navigate the building.

In computer science, the ability to write an algorithm at different levels of abstraction is a key skill - perhaps even the single most important skill. To illustrate this concept, let's look again at our `MakeRamen` algorithm from Chapter 1.

```
Algorithm MakeRamen:

boil water
add noodles to water
wait 6 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

The first action in this pseudocode algorithm is `boil water`. This is a great example of abstraction, because the action `boil water` glosses over all of the details of **how** to boil water. For humans, these details are pretty unimportant, because almost all adult humans know how to boil water. But imagine that this algorithm has to be carried out by a humanoid cooking robot. The robot does not intuitively know how to boil water. The action `boil water` is too *abstract* for it. It needs more details.

2.2 Refinement

Refinement is the process of adding more details to an algorithm. It is the inverse process to abstraction. For example, we might *refine* the `boil water` action by replacing it with a sequence of actions (shown in red text) that describe how to boil water in more detail:

```
Algorithm MakeRamen:

get pot from cupboard
place pot under faucet
add water to pot
place pot on stove
turn on burner
wait until water boils
add noodles to water
wait 6 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

Listing 2.1: `MakeRamen` algorithm with **refinement** of the `boil water` action.

Each of these new actions describes an action that partially carries out the original `boil water` action. But, depending on the sophistication of our cooking robot, even these actions may not be detailed enough for it to carry out the task. At some point, the robot needs to know exactly where, and for how long to position its legs and arms to carry out these actions. This would require that we further refine the actions `place pot under faucet`, `add water to pot`, etc. to the level of detail where we tell the robot exactly where and how to move its limbs by replacing each of these actions with sequences of even more detailed actions. This is called *stepwise refinement*. We repeatedly replace actions that are too abstract with a sequence of less abstract, more detailed actions until we reach a level of detail such that the actions are both *feasible* and *self-explanatory* for the algorithm's audience (in this case, our cooking robot).

The ability to think at different levels of abstraction and move between them is critical to success as a programmer and a computer scientist. We abstract away details when they are not important, and refine abstractions later when we are ready for the details. It's not an easy skill and it takes practice.

2.3 Encapsulation

Encapsulation is a process in which several related actions are grouped together and given a name. It is a tool that helps us break up tasks and temprorarily hide detail, thus enabling effective abstraction.

Without good encapsulation, it can be hard to grasp the high-level structure of an algorithm without getting lost in the details. For example, consider the following algorithm for making lunch:

```
Algorithm MakeLunch:

boil water
add noodles to water
wait 6 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
open fridge
get milk carton from fridge
pour milk in glass
return milk carton to fridge
open cookie jar
put cookie on plate
```

Listing 2.2: Making lunch without **encapsulation**

It should be apparent that the first several actions in this algorithm come directly from our `MakeRamen` algorithm that we've seen before. Furthermore, the steps after making the ramen, while important to a delicious and low cost student lunch, have nothing to do with making ramen. In other words, there is a natural break between high-level concepts in this algorithm. Employing encapsulation would allow us to see the overall organization of this lunch-making algorithm much more clearly, as follows:

```
Algorithm MakeLunch:

MakeRamen
PourMilk
GetCookie
```

Listing 2.3: Making lunch with the details encapsulated

If you've been paying particularly close attention, you may have noticed that we've always given names to all of our sample algorithms so far. This hasn't been accidental - now we can use that name to refer to the entire algorithm. In essence, we can think of the `MakeLunch` algorithm as consisting of three other encapsulated algorithms. The first of these is the `MakeRamen` algorithm that we've seen many times. The `PourMilk` and `GetCookie` algorithms are constructed simply by taking the details from our original `MakeLunch` algorithm and encapsulating them with their own names.

```
Algorithm PourMilk:

get milk carton from fridge
pour milk in glass
return milk carton to fridge
```

```
Algorithm GetCookie:

open cookie jar
put cookie on plate
```

Encapsulation can be useful when writing algorithms of any length, but it's especially crucial for computer programs. Large-scale software can consist of thousands or even millions of lines of code.

There's no way anybody could make sense of it all if the code wasn't encapsulated to break it up into manageable chunks.

2.3.1 Input and Output

An algorithm's *input* specifies resources or information that need to exist in advance before the algorithm can be carried out. An algorithm's *output* specifies the results that must have occurred by the time the algorithm is finished. Input and output are crucial to the concept of encapsulation because together they define the boundaries of the encapsulated algorithm. You can think of this process as something like a business contract: the encapsulated algorithm says "if you give me these things (the inputs), then I will make for you these other things (the outputs)".

Often, we include these inputs and outputs as part of the description of an algorithm, right after the algorithm's name. For our running ramen example[1], this would look as follows:

```
Algorithm MakeRamen:
Inputs: One ramen package
Output: Hot bowl of cooked ramen

boil water
add noodles to water
wait 6 minutes
drain the noodles
stir in contents of flavour packet
place cooked noodles in bowl
```

Notice that the algorithm says nothing whatsoever about how the initial package of ramen was acquired. From the perspective of the algorithm, how the ramen package was obtained doesn't matter. As long as you had such a package available, you could follow the algorithm to cook the ramen. Notice too that the algorithm might also produce things that weren't specified by the *output*. In this case, the algorithm will very likely result in a dirty pot in addition to the hot bowl of noodles. The output is not supposed to be an exhaustive list of everything that will result from following the algorithm. Instead, it is supposed to list the goal or purpose of following the algorithm. Any other results are incidental. In other words, the input and output are defined by the **problem** that the algorithm is solving, not by the algorithm itself.

In fact, it light of this last insight, we can present the following (very abstract) algorithm for writing algorithms!

```
Algorithm WriteAlgorithm:

identify the problem
define the solution to the problem (output)
decide starting conditions of the problem (inputs)
design an algorithm that produces the output given the input(s)
```

Listing 2.4: A correct algorithm for writing algorithms

Contrast this with the following algorithm, which is a much worse way of designing algorithms. We highly advise against it.

[1] Hopefully, you are not getting sick of ramen by now. As a student, you may have to get used to it.

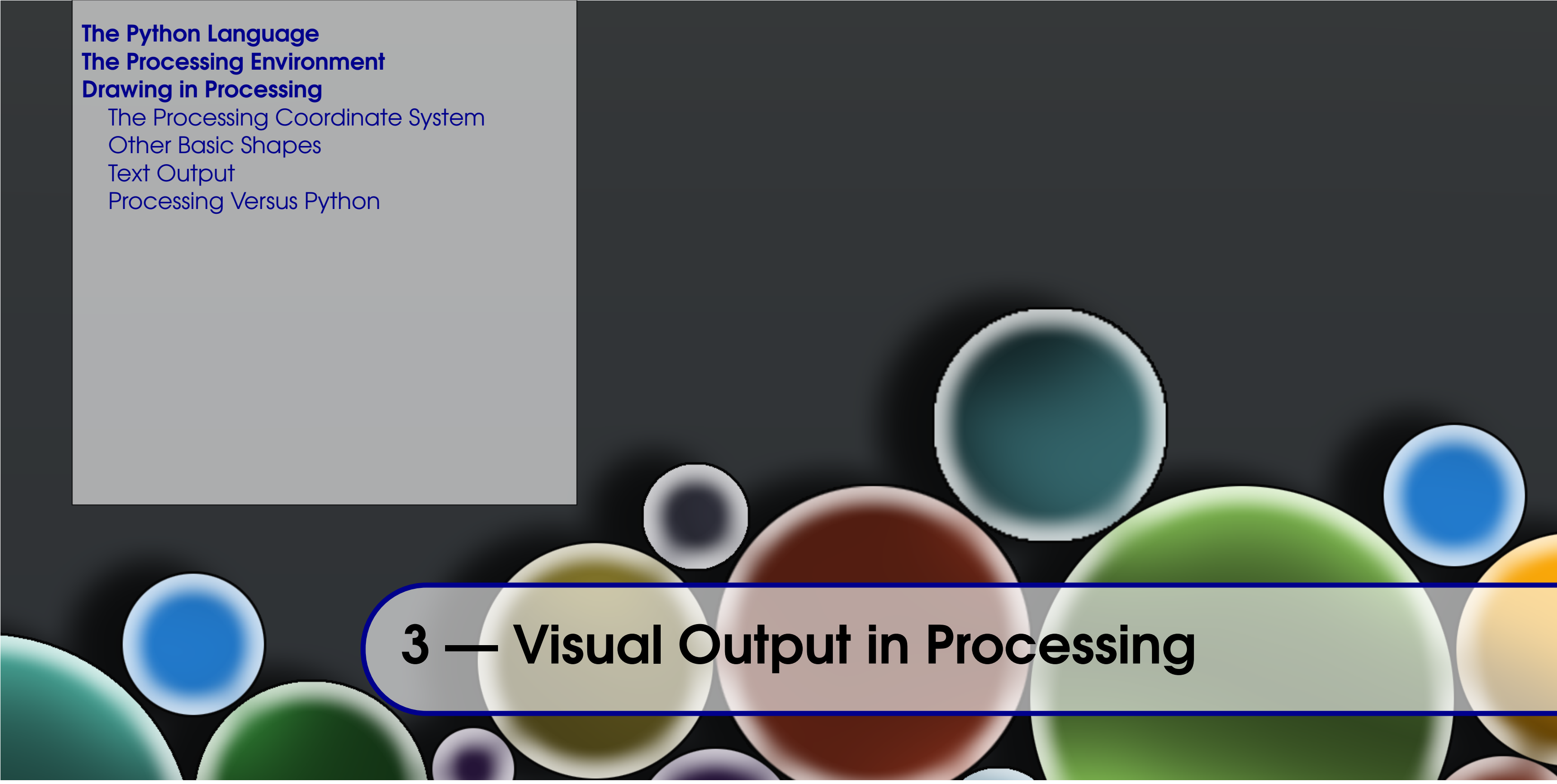
```
Algorithm WriteAlgorithm:

identify the problem
write an algorithm that has something to do with the problem
figure out what the algorithm does: call these outputs
figure out what the algorithm needs: call these inputs
```

Listing 2.5: A less effective algorithm for writing algorithms

2.4 Why Abstraction is Important

With regard to how computers work, you may have heard something along the lines of "computers only really understand 1s and 0s". This is more or less true. However, very few, if any, modern computer scientists write their computer code as enormous streams of 1s and 0s. Instead, we mostly write code in a so-called "high-level" language, like the Python language you will be learning in this class. Languages like Python are more precise that the pseudocode algorithms we've seen so far, but are still fairly easy for humans to read (all things considered, of course!). When the algorithm that we write in Python is actually run on a computer, there are multiple pieces of software and hardware that automatically translate the code into the infamous sequences of 1s and 0s. There are several steps to this translation process, and each one *refines* the original algorithm into a lower and lower level of abstraction. In other words, without abstraction, modern computers wouldn't work at all even for the simplest tasks!

To summarize, abstraction allows us to think about performing higher-level, more complex actions without worrying about **how** they are performed. Abstraction doesn't mean that the lower-level details of how an abstracted algorithm is carried out don't exist or never have to be written at some point. It is just a mechanism that allows us to ignore such details when it is convenient or until they are needed - a mechanism that is central to the entire field of computer science.



3 — Visual Output in Processing

Learning Objectives

After studying this chapter, a student should be able to:

- distinguish between a programming language and a development environment
- describe the visual coordinate system used in Processing
- author Processing programs with simple visual output such as lines and shapes

3.1 The Python Language

A *programming language* is a language that human programmers use to give specific instructions to a computer. In some ways, programming languages are similar to natural languages, such as English, French, Russian, Punjabi, and many others. Both types of languages have *vocabularies*, or words that mean specific things. Both types of languages have a grammar or *syntax*, which specifies how words in the language can be combined with each other and with punctuation. A big difference is that with a natural language, syntax doesn't have to be perfect in order to be understood by an audience. With a programming language, you do not have this luxury; if you make a single mistake in your syntax, the computer will not understand you at all. Another difference is that natural language can be ambiguous: the same word can have multiple meanings, or might be interpreted differently by a different audience. With a programming language, there can be no ambiguity; every word or statement has one and only one meaning.

For this class, we will use a programming language called **Python**. A Python program consists of one or more valid Python statements, each on its own line. Line breaks (i.e. the thing you get when you hit the 'return' or 'enter' key on most keyboards) are very important for separating statements in Python (as is indentation and other forms of "whitespace", as we will see a little later). The following is an extremely simple, but complete, Python program:

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

The only effect of this program is to print the phrase `Hello world!` to the computer's monitor. The word `print` is part of Python's built-in vocabulary. It is a command that displays text to the screen. It's important to note that Python, as a language, is case-sensitive, so `print` and `Print` are **not** the same thing. In general, if you're trying out any of the examples you see in these readings, you should type them **exactly** as they appear here, including case and punctuation. Don't worry too much about what the brackets or quotation marks mean yet, we'll get to why you need those later. For now, they're just part of the proper Python syntax for the `print` command.

The following is an example of an incorrect Python program:

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

Although on their own, both `print("Hello world!")` and `print("Goodbye!")` are valid Python statements, the syntax rules of Python insist that you can't put them on the same line. If you try to ask a computer to run this Python program, you'll get an error. Don't expect the computer to suggest the obvious correction of just moving the second statement to a new line either. Computers, as a general rule, aren't very good at telling you *why* your syntax is incorrect, which can be understandably frustrating for novice programmers. Just remember, the computer isn't actually trying to make things difficult for you, it simply isn't anywhere near as smart as you are!

3.2 The Processing Environment

A *development environment* is a piece of software that is designed to help programmers write computer code. For this class, we will be using a development environment called **Processing**. Processing makes our lives as programmers easier in a variety of ways. For instance, just as a document editor like Microsoft Word can recognize spelling mistakes and bring them to your attention, Processing recognizes Python keywords and other programming constructs and can highlight them in different colors, making our code easier to read and write. Telling the computer to run your code in Processing is as simple as pressing the "Run" button that appears directly above the coding window. Processing also gives us a *console* where your program can print any text-based output, as well as a separate window called the *canvas* for displaying simple graphics or pictures that your program might draw. All of these things are just conveniences for us. There's no reason we couldn't write our Python code in a plain text editor like Notepad [1] and then later tell the computer to run it. But by using Processing, we have everything we need all in one place.

Processing actually goes a little beyond other development environments in that it expands the Python language a bit by adding some new commands that don't normally exist in plain Python. For instance, Processing allows us to create simple visual output, such as lines and shapes, often with only a single command. Without the Processing environment, doing these sorts of things using Python is still possible, but more complicated. The downside to using the Processing environment is that it's a bit tricky for novices to tell where basic Python stops and the extra functionality provided by Processing begins. Throughout these readings, we'll try to point out the distinction where we can, but ultimately it's not all that important for the moment anyway. Our goal here is to learn the fundamentals of programming, not to learn every possible detail of a specific language or tool.

[1] Never use a document editor like Microsoft Word to write code. Document editors often sneakily insert invisible formatting commands into your document, which will mess up your code when the computer tries to run it.

3.3 Drawing in Processing

Processing makes it easy to create nice-looking visual output quickly, so we're going to take advantage of that to write our very first basic Python programs. We'll start off by using the `line` command to draw a line on the screen, like so:

```
line(0, 0, 10, 10)
```

The program above draws a rather small, diagonal line on the canvas that Processing automatically pops up every time you run a program. You have likely noticed that we included four numbers, separated by commas, inside the brackets next to the `line` command. Those numbers are to tell Processing where to draw the line on the canvas. This illustrates a very important concept in computer science, namely, the *parameterization* of commands. The people who created Processing could have designed the `line` command such that it always drew the line in exactly the same place, but that wouldn't have been very useful. Instead, by specifying four numeric parameters for the command, the designers created a single command that allows programmers to draw a line anywhere they like (well, anywhere on Processing's canvas).

In order to effectively use the `line` command, we need to know the meaning of those four numbers. In this case, it's easy: the numbers specify the (x,y) coordinates of the end points of the line. In other words, in English, we would express the command above as ‘‘`draw a line from coordinate (0,0) to coordinate (10, 10)`’’. The ordering of the numbers is how Processing knows which number is which; thus, the first number is always the x coordinate of the first point, and so on.

3.3.1 The Processing Coordinate System

Processing places any lines or shapes that it draws on its canvas. The canvas uses an (x,y) coordinate system to specify locations. You might recall from math class that coordinate $(0,0)$ (the origin) is normally located in the center of the standard Cartesian coordinate system. That system has four quadrants (some of which use negative coordinates for x and y). The coordinate system used by Processing is a little different from the Cartesian one. In Processing, there are no quadrants, coordinates are never negative, and the coordinate $(0,0)$ is always the **top left corner** of the canvas. This means that when we're talking about locations on the canvas, increasing the x coordinate moves us to the right (which you're used to in math class) and increasing the y coordinate moves us down (which is the **opposite** of what you're used to in math class).

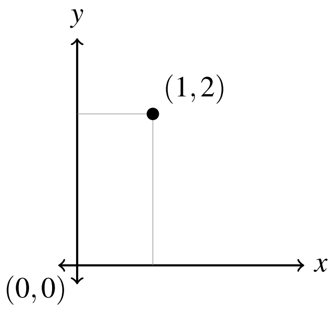


Figure 3.1: Cartesian Coordinate System

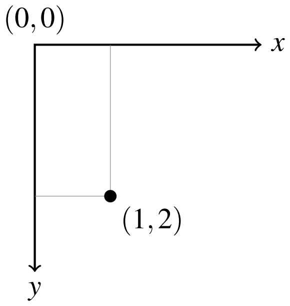


Figure 3.2: Processing Coordinate System

The size of the canvas is measured in *pixels*, and by default, the Processing canvas is 100x100 pixels in size. We can change that with the `size` command, like so:

```
size(200, 200)
line(0, 0, 150, 150)
```

Here, the two numbers in the `size` command specify the width and height (in that order) of the canvas in pixels. You'll have to play around with Processing to get a sense of how big a pixel is, but it's fairly small; you won't see much of a difference between `line(0, 0, 50, 50)` and `line(0, 0, 51, 51)`, for example.

3.3.2 Other Basic Shapes

Processing provides several other commands for drawing two-dimensional shapes. These are just a few examples.

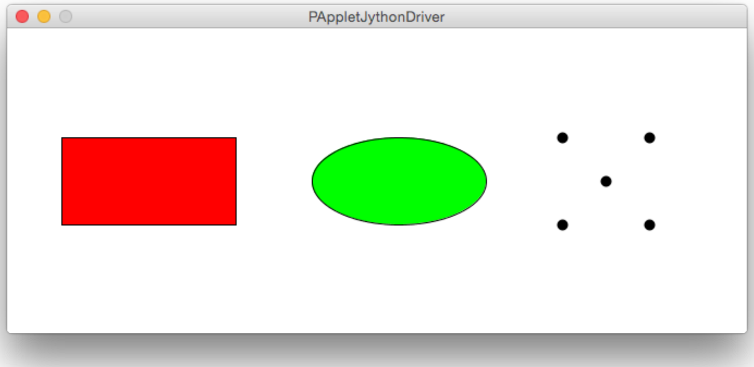


Figure 3.3: Rectangle, Ellipse, and Points drawn in Processing

Rectangles

```
rect(10, 10, 20, 30)
```

The `rect` command, as you might expect, draws a rectangle on the canvas. Again, Processing needs to know some more details about the rectangle, such as where to draw it and how big it should be, and that's what the numbers are for. The first two numbers specify the (x,y) coordinates of the **upper left corner** of the rectangle, and the second two numbers specify the width and height of the rectangle, in that order [2].

Ellipses

```
ellipse(50, 40, 20, 30)
```

We can draw ellipses as well. In this case, the first two numbers specify the (x,y) coordinates of the **center** of the ellipse, and the next two numbers specify its width and height respectively. If the width and height are equal, then we get a circle.

[2] It might occur to you that there are other ways to describe the size and location of a rectangle, and indeed, Processing lets us use some of these other ways if we really want.

Points

```
point(42, 17)
```

The point command draws a single point on the canvas. Since the point's size is fixed at one pixel, we only need two numbers to describe the point, and these are its (x,y) coordinates.

3.3.3 Text Output

Processing is a great tool for drawing shapes and pictures, but sometimes we might want to display less flashy but still useful textual output as well. There are two ways we can do this. We've already briefly seen the `print` command at the start of this chapter. As a reminder, it looks like this:

```
print("Here is some text.")
```

The `print` command displays the text that you provide between the parantheses to the *console*, which is the black area underneath your coding window in Processing. Notice that the text we want to display is enclosed in double-quotes (“ and ”), and those quotes **don't** get displayed along with the text when the `print` command is executed by the computer. In Python, any time you're working with words or sentences that you want to be displayed “as is”, you need to enclose them in double quotes. For instance, this `print` statement will result in an error.

```
print(You can try this.  It won't work.)
```

As we've seen with the other commands in this chapter, you don't need the quotation marks when dealing with numbers. You only need them with text. In later chapters, we'll say more about why this is so.

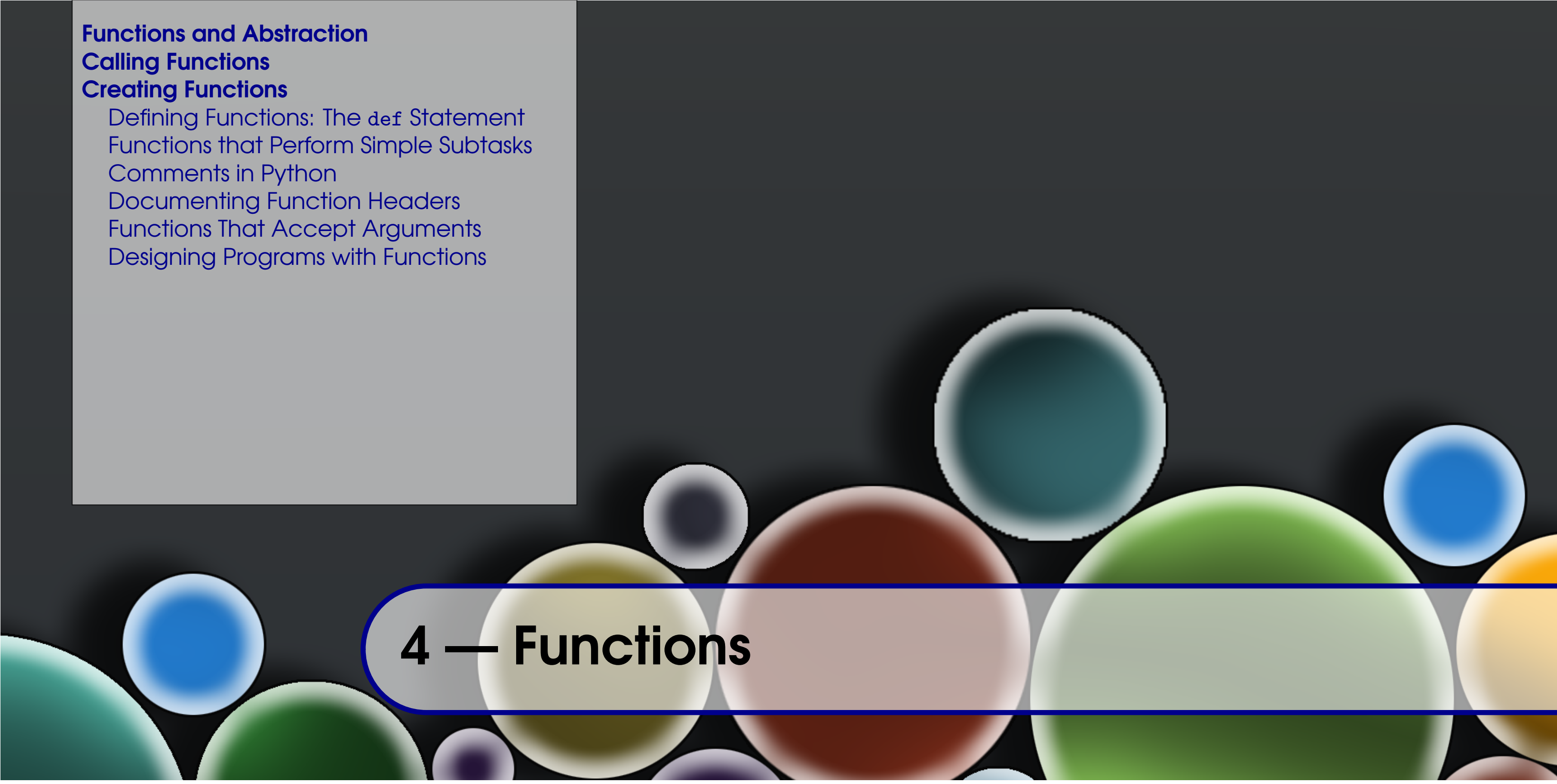
Processing also lets you display text on its canvas. For that, we use the `text` command, like so:

```
text("Hello there!", 0, 10)
```

You'll notice that the `text` command needs two extra numbers that `print` didn't. That's because since we're using the canvas now, the `text` command needs to know not only the text that you want to display, but also **where** on the canvas to put it. The two numbers give the (x,y) coordinates of where to place the text.

3.3.4 Processing Versus Python

With the exception of the `print` command, all of the commands we've described in this section are specific to the Processing environment and do not exist in plain Python. In fact, as a general rule, if a command has anything to do with drawing on the canvas (including commands like `size` for re-sizing the canvas), then it's one of the ‘extra’ commands that Processing provides. Of course, these commands still follow standard Python *syntax* — the commas and the parantheses are part of that — but they're not in Python's default *vocabulary*. Again, this distinction isn't going to be terribly important for the purposes of this course. Just don't be confused if you find the `line` command doesn't work in plain Python.



4 — Functions

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a function is
- describe how to call a function
- explain what the arguments of a function are
- compose functions in Python that perform a subtask
- compose functions in Python that accept arguments as input
- describe the role of a function's parameters
- distinguish between arguments and parameters
- author appropriately descriptive comments to document a function

In Chapter 2 we talked about encapsulation, which is the idea of giving a name to an algorithm and allowing us to refer to all of the instructions in that algorithm simply by referencing its name. Python allows us to do this through a mechanism called *functions*. Functions allow us to give a name to a block of Python code. If an algorithm is implemented as a function in Python, we can run the algorithm by using the function's name in a Python program. We refer to this process as *calling* the function.

Section 4.1 of this chapter introduces functions as a method of abstraction, and their relationship to algorithms. Section 4.2 discusses the Python syntax for calling functions. Finally, section 4.3 introduces the Python language syntax for defining functions so that we can implement our own algorithms as functions.

4.1 Functions and Abstraction

Functions are an example of abstraction. They allow us to refer to a block of Python code by name, and ask for that code to be executed. The code within a function can be executed without knowing what it is, or how it works. In fact, we've secretly been doing exactly that already, except we've been using the word 'command' instead of 'function'. All of the Processing code for drawing shapes from Chapter 3 are examples of using functions.

Recall from Chapter 2 that algorithms can have input. Thus, it may not surprise you that functions can have input, and indeed most functions do. Inputs to functions in Python are called *arguments*. When we used the `line` function before, we provided four numbers that told the Processing environment where to draw the line on the screen. Those numbers were arguments to the `line` function.

We also said in Chapter 2 that algorithms can have output. Python functions can have output too, but we're not going to talk about that too much until a little later. Of course, you might rightly be asking yourself: "isn't making something appear on the computer's monitor a form of output?". In a way, that's true. However, with regard to functions, output has a very precise and specific meaning and making something appear on the screen doesn't qualify. We'll return to this point in a later chapter.

The key thing to note about functions is that, at the time that you write the code that calls them, you only need to know **what** the function does, not **how** it does it. You might think that drawing a line on a screen sounds pretty simple, but it's actually still too abstract for a computer to be able to do it without further explanation. The authors of Processing wrote the details of the `line` function for us, and we don't need to know how they did it in order to use it. All we need to know is what it does, and what the meaning of its arguments are.

4.2 Calling Functions

In Python we invoke the algorithm inside of a function by *calling* the function. To call a function, we write the name of the function followed immediately by a pair of parentheses. The *arguments* to the function (inputs!) are given as a comma-separated list within the parentheses. We illustrate this with the `line` function we saw in Chapter 3:

```
line(0, 0, 10, 10)
```

We have coloured the different parts of the function call:

Red: The function name.
Green: Parentheses enclosing the list of arguments.
Blue: The arguments (inputs) to the function.
Brown: The commas separating the arguments in the argument list.

All function calls have the same general format and look like this:

```
functionName(argument1, argument2, argument3, ...  )
```

Functions can, of course, have different numbers of arguments. We've already seen an example of this as well; the `point` function only had two arguments, and looked like this:

```
point(0, 0)
```

In fact, a function might require no arguments at all. The `clear` function in Processing, which simply sets the entire drawing canvas to black, is an example of this:

```
clear()
```

Notice that we still need to include the parantheses, even though there is nothing in them! That's just proper Python syntax for making a function call. From now on, when we refer to a function like `line()` in this text, we'll include a pair of matching round brackets just to make it clear that we're talking about a function. This is just a short-hand for clarity; note that we didn't include any numbers in between the parantheses, even though to actually call the `line()` function, you would need to provide four numbers as arguments. We're just making it clear that `line()` is, in fact, a function.

4.3 Creating Functions

The ability to create your own functions and create abstractions of your own algorithms is a tremendously powerful feature in any programming language. The main reasons for writing your own functions are abstraction and decomposition of large programs into manageable pieces. We can give names to our algorithms and abstract away their details by writing them as functions. The purpose of this section is to learn how to do this in Python.

4.3.1 Defining Functions: The `def` Statement

In Python, functions are defined by writing the keyword `def`. A *keyword* is a name we give to words in a programming language that have special meaning. Up until now, you might have thought that words like `print` and `line` were Python keywords, but they weren't: they were just function names. Keywords cannot be used as function names, or as anything else for that matter, except for the very specific purpose defined by the programming language.

4.3.2 Functions that Perform Simple Subtasks

The simplest form of function is one that takes no arguments as input. Suppose that you wanted to put your name and contact information in the upper left corner of the canvas for every program that you write in Processing. We can create an abstraction of this algorithm by writing a function called `signature()` that performs the algorithm when we call it. Here's what that definition would look like:

```
def signature():
    text("Cookie Monster", 5, 10)
    text("123 Sesame Street", 5, 20)
```

Let's break down what's happening here. The first line uses the keyword `def` to define a function called `signature()`. The name of a function in a function definition must be followed by a pair of parentheses, then a colon. This entire line is called the *function header*. Notice how the rest of the lines after the function header are indented. In Chapter 1, we called this a *block*; when it's part of a function, we also refer to such a block as the *function body*. Just like in our pseudocode, we group statements together in blocks by indenting them. All of the Python code that is part of a function body has to be in the same block, so it has to be indented. What's more, it must be indented by **exactly** the same amount or Python will complain thinking that some lines are not part of your function even when you want them to be[1].

[1] If we're getting really picky, we can note that you can use either tabs OR spaces for indentation, but whichever you pick, you **must** be consistent. If you mix spaces and tabs in together, Python will complain!

```
# Wrong indentation; Python will think that the second text line
# is not part of the function, and will just execute it.
def signature():
     text("Cookie Monster", 5, 10)
text("123 Sesame Street", 5, 20)


# Wrong indentation; Python will issue an error here because the
# indentation is inconsistent, and it can't figure out
# which lines are part of the function and which aren't
def signature():
     text("Cookie Monster", 5, 10)
    text("123 Sesame Street", 5, 20)
```

Now, the correctly indented function definition doesn't actually do anything other than define the function. A function definition is just a set of instructions that has been given a name. Those instructions only get executed by the computer when the function is *called* by using its name somewhere else in the program:

```
# defines the function only:
def signature():
     text("Cookie Monster", 5, 10)
     text("123 Sesame Street", 5, 20)

# this function call tells Python to execute
# the instructions defined above
signature()
```

Finally, it's worth noting that once you've defined a function (by using the `def` keyword), you can call it elsewhere in your program not just once, but as many times as you like. In fact, that's the main point of having function definitions at all! Each time the function is called, the computer will execute the instructions that are specified in the function definition.

Defining before Calling

In Python functions must be defined before they are called. Thus a function definition must appear in a file prior to any calls to that function.

```
signature()
# this fails -- function called before definition

def signature():
     text("Cookie Monster", 5, 10)
     text("123 Sesame Street", 5, 20)

# this is fine, by this point the function has been defined
signature()
```

4.3.3 Comments in Python

By this point, you might be wondering why we started including lines that begin with the # symbol in our code examples. In Python, the # denotes a *comment*. All of the text on any line that begins with the # symbol will be completely ignored by the computer when it tries to execute your code. Comments are purely to help humans understand the computer code when they are reading it. Because computer programs can get complicated very quickly, good commenting is essential to being a good programmer. Of course, since comments are a non-technical part of programming, rigorously defining a 'good' comment isn't very easy. One way we can put it is that good comments should be at a higher level of abstraction than the code itself. Comments should try to explain the purpose or behaviour of whole blocks of code — you certainly don't need a 1-to-1 ratio between lines of code and comments.

The following is an example of bad commenting:

```python
# draws a line from (10, 10) to (10, 20)
line(10, 10, 10, 20)
# draws a line from (10, 10) to (20, 10)
line(10, 10, 20, 10)
# draws a line from (20, 10) to (20, 20)
line(20, 10, 20, 20)
# draws a line from (10, 20) to (20, 20)
line(10, 20, 20, 20)
```

The comments here are completely redundant to anyone who understands Python syntax and knows what the `line()` function does. Good comments should instead give an idea of *why* you are calling the line function, and how (if at all) the four function calls are related. The following is an example of much better commenting:

```python
# draws the outline of a square using lines
line(10, 10, 10, 20)
line(10, 10, 20, 10)
line(20, 10, 20, 20)
line(10, 20, 20, 20)
```

4.3.4 Documenting Function Headers

The purpose of a function is to encapsulate a portion of your program so that it can be called without the need to know exactly how the function does its job. As a result, function headers should always be immediately followed with documentation that indicates what the function does and how to use it. Because this function documentation is so important, we use a separate syntax for it: the function documentation is enclosed in matching sets of triple quotes (the " character repeated 3 times). It should look like this:

```python
def signature():
    """ prints my name and address
    to the upper left corner of the canvas """
    text("Cookie Monster", 5, 10)
    text("123 Sesame Street", 5, 20)
```

The text in between the triple quotes is a special kind of comment called a *docstring*. You can only use docstrings when you are defining a function, starting on the line immediately after the function header. The docstring can span multiple lines if need be, as shown in the example above.

4.3.5 Functions That Accept Arguments

As we mentioned in Chapter 2, most useful algorithms have inputs. The same is true of functions. In the function header, we can specify the inputs that the function needs in order to do its job. When defining a function header, we call these inputs *parameters*.

Let’s try an example of this. Suppose you were writing a program in Processing and you knew you were going to draw a lot of circles on the screen. We already know we can use the `ellipse()` function to do this, but `ellipse()` takes four arguments, and that’s a lot of typing. We’ll write a function that only draws circles to make things more convenient for us.

```
def circle(x, y):
    """ this function draws a 15x15 circle
    x: x-coordinate of the circle’s center
    y: y-coordinate of the circle’s center
    """
    ellipse(x, y, 15, 15)
```

In addition to the `def` keyword and the name of the function, you can see that we now have something inside the parentheses of the function header, namely, `x` and `y` separated by a comma. This shows that whenever you call the function, you’ll need to give it two arguments, and that whatever the value of those arguments, the function is going to refer to the first argument as `x` and refer to the second argument as `y`. In other words, the function is going to give names to any inputs that you give it. It has to do this, after all, because the function doesn’t know in advance the values of the arguments it will be given. Ideally, these names should be chosen so as to give some idea of what the parameters will be used for.

The function body here consists of just one line of code, where the `x` and `y` parameters are passed along as inputs to the `ellipse()` function that we saw in Chapter 3.

Once we’ve defined our `circle()` function, we can call it multiple times to draw circles on the screen. The following code will draw three circles in a single column by using function calls to `circle()`.

```
def circle(x, y):
    """ this function draws a 15x15 circle

    x: x-coordinate of the circle’s center
    y: y-coordinate of the circle’s center
    """
    ellipse(x, y, 15, 15)

circle(10, 10)
circle(10, 30)
circle(10, 50)
```

Note that the function definition must occur **before** any attempt to call the function. The following code, for instance, will result in an error:

```
circle(10, 10)

def circle(x, y):
    """ this function draws a 15x15 circle

    x: x-coordinate of the circle's center
    y: y-coordinate of the circle's center
    """
    ellipse(x, y, 15, 15)
```

Arguments Versus Parameters

While talking about functions, we've used both the terms *arguments* and *parameters* in a way that might make them seem like interchangeable words. They're not, but it can certainly be a little tricky to keep them straight. *Arguments* are the values that we pass to a function when we **call** it. *Parameters* are the names listed in between the parantheses in the **definition** of the function header. So with regard to our `circle()` example from before, it would be correct to say "`circle()` is a function that has two parameters called `x` and `y`". It would also be correct to say "we later called the `circle()` function with arguments of 10 and 30." To put it another way, parameters tell you what the function needs; arguments are the specific values that you give the function in any particular instance.

4.3.6 Designing Programs with Functions

One of the main purposes of functions is to make programs easier to design and read. Take a look at the following program in Processing. Can you figure out what it does just by looking at it?

```
size(200, 200)
ellipse(100, 35, 25, 25)
line(100, 50, 100, 80)
line(100, 52, 70, 45)
line(100, 52, 130, 45)
line(100, 80, 85, 125)
line(100, 80, 115, 125)
```

Of course, you could copy this program into Processing and run it to see what you get, but even then, you would only know what the program does as a whole. If there was some small thing about the program's behaviour that you wanted to change, you wouldn't know where to start.

Here's the same program, but organized into functions. Now can you guess what it might do?

```
def drawHead():
    ellipse(100, 35, 25, 25)

def drawBody():
    line(100, 50, 100, 80)

def drawArms():
    line(100, 52, 70, 45)
    line(100, 52, 130, 45)

def drawLegs():
    line(100, 80, 85, 125)
    line(100, 80, 115, 125)

size(200, 200)
drawHead()
drawBody()
drawArms()
drawLegs()
```

Sure, the overall program is longer now since we had to type all of those function headers, but the organization is much better. And if you wanted to tweak this program — say you decided that the legs were too long — you know exactly where to start looking without messing up anything else. Of course, even this is just a very simple example. Functions are even more useful when they have parameters that let us alter their behaviour in different circumstances, like with the `line()` function that Processing provides for us.

5 — Colour in Processing

Learning Objectives

After studying this chapter, a student should be able to:

- describe the grayscale colour model
- describe the RGB colour model
- author Processing code that uses colour

Proper use of colour can make visual output more appealing and useful. In this chapter, we'll learn how to use colour in our Processing programs.

5.1 Grayscale Colour

The simplest form of digital colour is the *grayscale* colour model. In grayscale, vibrant colours that you know and love, like blue, red, yellow and so on, don't exist. There's only black and white, and in between them, various shades of gray.

In Processing (and on computers more generally), colour is represented by numbers. A grayscale colour is represented by a single whole number in the range of 0 to 255 (inclusive). 0 means black and 255 means white. All the values in between represent increasingly light shades of gray.

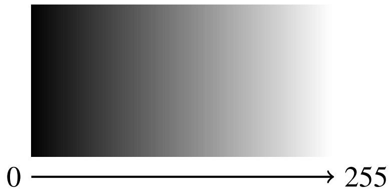


Figure 5.1: Grayscale values from 0 (black) to 255 (white)

5.2 Colour Functions in Processing

Just as Processing provides us with several ready-made functions for drawing simple shapes, it also provides functions for using colour. We'll look at a few such functions here.

5.2.1 Background Colour

The `background()` function changes the background colour of Processing's canvas. You've probably noticed by now that the default background for the canvas is a light-ish shade of gray, but we can set it to anything we want. The following function call would set the background to white:

```
background(255)
```

5.2.2 Stroke Colour

The `stroke()` function sets the colour that Processing will use to draw lines and the borders around all its shapes. It's worth noting that calling this function by itself won't actually appear to do anything on the canvas. We'll also need some function calls that draw some lines or shapes to see the effect of calling the `stroke()` function. The following program will draw a pair of parallel white lines on a black background.

```
background(0)
stroke(255)
line(10, 10, 10, 20)
line(20, 10, 20, 20)
```

Notice that in the program above, we only called the `stroke()` function once, but both of the lines that get drawn will be white. That's because calling the `stroke()` function changes the default colour for all lines and shapes until it gets called again. If we call the `stroke()` function a second time, then any shapes drawn **after** that point will use the new colour, but it won't change the colour of any shapes already drawn. Think of `stroke()` in the sense of painting on a real-life canvas with a paintbrush; you dab the paintbrush into the paint colour you want to paint with so that every new stroke you draw on the canvas will take on that colour until you dab the paintbrush into another paint colour, from whence all new strokes after that will take on the new paint colour. For example, the following program draws two white lines followed by two grey lines (all on a black background).

```
background(0)
stroke(255)
line(10, 10, 10, 20)
line(20, 10, 20, 20)
stroke(125)
line(10, 40, 10, 50)
line(20, 40, 20, 50)
```

It might occur to you that there are other ways Processing could have handled the issue of setting colours for shapes. For instance, instead of having the notion of a default stroke colour that Processing remembers and uses for all forms of drawing, the authors could have designed the `line()` function (and all other functions that draw shapes) to have an extra function parameter that specifies the colour of that particular line or shape. Doing it the way they did was a conscious design decision that the

Processing authors made. Whether or not this decision was the right one depends on what the people who use Processing — people like you — find most useful.

5.2.3 Fill Color

The `fill()` function sets the default colour that Processing will use to fill the middle of shapes that it draws. Conceptually, it's the same as the `stroke()` function in that when you call it once, all subsequent shapes will use the new colour until the `fill()` function gets called again to set a different colour. The following program uses the `fill()` function to draw a white box and a black box (on the default gray background).

```
fill(255)
rect(10, 10, 20, 20)
fill(0)
rect (10, 50, 20, 20)
```

The only extra thing we'll note here is that the outline for the white box is still clearly black (and so is the outline of the black box), even though the middle of the box is white. If we wanted the box to be completely white, we'd have to change both the stroke colour and fill colour like so:

```
fill(255)
stroke(255)
rect(10, 10, 20, 20)
```

5.3 The RGB Colour Model

Processing also implements the *RGB colour model*, and this is what we use if we want fuller access to the colour spectrum. Under the RGB model, colours are represented by three separate numbers, instead of just one. As before, the numbers range from 0 to 255. The three numbers represent the intensities of **Red**, **Green** and **Blue** (RGB) light, respectively. For example, the RGB value $(255,0,0)$ represents pure red, because the red component (the first one) is at its maximum value and the other two components are zero. Similarly, $(0,255,0)$ is pure green. $(255,255,0)$ is bright yellow, since that's the colour we get from mixing red and green light. $(0,0,0)$ gives us black (the absence of any light), and $(255,255,255)$ gives us white.

Thinking about colour as a mixture of different colours of light, as done in Processing and almost all digital displays, is known as the *additive* colour model. This is in contrast to the *subtractive* colour model, which is the model we use if we're making colours by mixing paints or dyes. That's why ink for colour printers typically comes in cyan, magenta, and yellow, and not red, blue and green. But the only important thing for us to know right now is that Processing uses the additive, not the subtractive, colour model.

5.3.1 Using RGB colours in Processing

It turns out that all of Processing's colour-related functions we saw when using grayscale have an RGB colour version as well. For instance, if we want to set the canvas background to some RGB colour, we can do so with the `background()` function:

```
background(0, 0, 255)
```

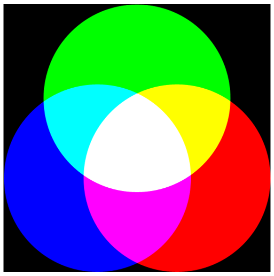


Figure 5.2: The RGB colour model with examples of additive colour mixing

Notice that the function name `background()` is exactly the same as when we called it with a grayscale value, but this time we provide three arguments instead of just one. You might think this is at odds with our earlier statement that “nothing in a computer program can be ambiguous”, but it actually isn’t. When we make this function call, Processing looks through all of the functions it knows about and asks itself “can I find a function called `background()` that has exactly three parameters”? It is the function name in **combination** with the number of parameters that allows Processing to uniquely identify the correct function definition to run.

The `stroke()` and `fill()` functions, and indeed all Processing functions that involve colour, work the same way and are capable of using either grayscale or RGB colour based on the number of arguments provided when calling the function.

6 — Interaction and Events

Learning Objectives

After studying this chapter, a student should be able to:

- define the elements of the interaction cycle
- distinguish between *interactive* and *run-to-completion* systems
- describe the behaviour of the Processing `setup()` function
- describe the behaviour of the Processing `draw()` function
- author basic interactive Processing programs

One of the big strengths of using the Processing environment is that it very quickly lets us create interactive systems. In this chapter, we'll define the essential elements of such systems, and present the Processing functions that facilitate their use.

6.1 Interactive Systems

An *interactive system* is a computer program that is designed to continually interact with a human user. The idea is that the human performs some action that the computer can detect, the computer processes that action and gives immediate feedback to the human. The human then decides what to do next based on the feedback, and the process continues as long as the human desires. Below we'll describe each of the elements of interactive systems in more detail.

6.1.1 The User

In computer science, the humans who are ultimately going to use a computer program on a regular basis are typically called *users*. The user is — or at least, should be — always the one in control of using an interactive system. Thinking, planning and decision-making are all jobs for the user, and in fact, users can sometimes get quite resentful if the system tries to take away one of these jobs!

Sometimes, the intended user for an interactive system is extremely general. The Google search engine is an interactive system and it is used by virtually everyone who uses the internet. Other times, a system's user-base is much more specific. For example, software designed to control a medical imaging machine is only likely to be used by doctors, nurses, and medical technicians. Knowing the capabilities of your users is often a key aspect of designing good interactive systems.

6.1.2 The System

The *system* is the computer program that interacts with the human user. The system needs to be able to interpret the actions of the human user, perform calculations, modify any data it might be working with, and modify its behaviour according to what the user needs and wants. It's also worth noting that the system is very rarely a stand-alone entity. It's usually sitting on top of a whole host of digital infrastructure that creates several layers of abstraction in between the human user and the system itself. For example, if you click the mouse on your computer, it's actually the operating system (for instance, Windows or maybe Mac) that first "realizes" that the mouse has been clicked, and it's the operating system that passes along this information to other programs that are running on the computer.

6.1.3 User Actions

User actions are physical events initiated by the user that the interactive system is able to detect. Moving the mouse or pressing the spacebar key are examples of user actions. Scratching the back of your head is not a user action, because the computer has no idea that you did it[1]. The purpose of a user action is to communicate something to the computer or to tell it to perform some task.

6.1.4 System Feedback

System feedback is the mechanism by which the system gives information back to the human user. This too must be done using a medium that the human user can access. The vast majority of system feedback is visual, and is achieved by displaying information on the computer's monitor. Sometimes audible feedback is used as well, such as the computer making a "ding" sound when the user clicks a button (or perhaps fails to click a button). Once the user has this feedback, they can use it however they please in order to inform their planning and decision-making as they continue to interact with the system.

6.1.5 Interactive versus run-to-completion

It might seem at this point that nearly any computer program could be called an interactive system, but that isn't the case. Traditionally, computer programs were often viewed as *run-to-completion* systems. The virus-scanning programs found on many modern computers are examples of run-to-completion systems. They pop up a little box that says "I'm about to scan your computer for viruses," and then they go ahead and do it (often whether you really like it or not!). There's no notion of continuous interaction with the human user. In fact, all of the sample Python programs we've shown in these readings so far have been "run-to-completion" systems. Such systems are sometimes still useful for teaching fundamental concepts in computer science, but wherever possible, we're going to try to use examples of interactive systems in this course from this point onwards.

[1] Not in general, anyway. But interfaces based on picking up user gestures via a computer's camera do exist, and you might even get to build one if you are pursuing a computer science degree.

6.2 Interaction in Processing

The Processing environment provides some extra functionality that makes it more convenient for us to build interactive systems. Previously, we've seen how Processing makes it easy for us to draw lines and shapes simply by using function calls to some ready-made functions. For interaction, the extra functionality is a little different. Instead of providing complete functions, Processing specifies a set of pre-determined **function names** that are related to making interactive programs. We still have to write the function definitions for these functions ourselves, using the `def` keyword that we learned in Chapter 4. But as long as we name these functions using a set of special names, Processing will automatically **use** these functions for us in interesting ways. Let's take a look at how this works.

6.2.1 The setup() function

The `setup()` function is a function that Processing will automatically call for you as the **very first thing** that happens whenever you run your Processing program. You don't need to put a function call anywhere in your code to make this happen, you just need to define the `setup()` function itself. As you might guess from the name, the `setup()` function is an ideal place to put instructions about things that only need to happen once and aren't going to change once your interactive system starts running. The following program is an example of defining the `setup()` function so as to give us a 300x300 canvas and set the initial background colour to white:

```
def setup():
    size(300, 300)
    background(255)
```

Including a definition of the `setup()` function in your program actually does one more thing which is a little bit subtle. If you define a `setup()` function in your program, then you **cannot** use any statements that draw on or modify the Processing canvas **outside** of any function definition. This is because by including a definition of `setup()`, you're telling Processing that you're making an interactive program, and so it uses slightly different rules. The following program, for example, will result in an error:

```
def setup():
    size(300, 300)
    background(255)

rect(20, 20, 15, 30)
```

The `rect()` funtion call isn't indented in the code above, so it's not part of the `setup()` function. Processing will complain about this. You **can** put other kinds of statements outside of the `setup()` function so long as they have nothing to do with the canvas. The following program will work just fine:

```
def setup():
    size(300, 300)
    background(255)

print("Set up and ready to go.")
```

Recall that the print() function displays text on the console, not on the canvas, so this is okay.

Finally, you can put statements that draw shapes **inside** the setup() function if you want. These shapes will be drawn on the canvas as soon as your program starts running, since that is when setup() automatically gets called.

6.2.2 The draw() function

The draw() function is a function that Processing will automatically and **continually** call for you for as long as your program is running. By default, Processing will call this function 60 times per second, which is once every 16.67 milliseconds. This might sound like a lot of work for the poor computer, but remember, repetitive work like this is exactly what computers are for! Regardless, it's certainly true that any statements you put inside the definition of the draw() function will get executed an awful lot. The following program increases the size of the canvas using setup(), and then repeatedly draws a circle in the center of the canvas using draw():

```
def setup():
    size(200, 200)

def draw():
    ellipse(100, 100, 10, 10)
```

Now at this point, you might well be wondering what the difference is between "repeatedly drawing a circle" and just "drawing a circle". In this particular case, to a human being watching this program when it runs, there is no difference. To a human, the following program will appear to behave exactly the same as the program above.

```
def setup():
    size(200, 200)
    ellipse(100, 100, 10, 10)
```

To the computer, however, these two programs are quite different. In the first case using draw(), every 16.67 seconds, the computer is dutifully drawing a circle in the middle of the canvas. The fact that there was already a circle displayed in exactly that spot is irrelevant to the computer's logic. In the second case, the computer just draws the circle once and then doesn't do anything else; it has no more instructions that it needs to follow. So why use the draw() function at all then? In this particular case, we wouldn't, as repeatedly re-drawing a circle in exactly the same place serves no constructive purpose. But as soon as we add some interaction to the program, everything will suddenly fall into place.

Interactive Drawing

In order to make our programs interactive, we need to take user actions into account. One of the most basic things a user can do is move the mouse cursor around on the screen. Fortunately, Processing gives us a very easy way to track mouse movement. The special Processing words mouseX and mouseY will report for us the x and y coordinates of the mouse cursor (so long as the cursor is anywhere on the Processing canvas). We can then use these values any way we like. In particular, we can use them as arguments to Processing's drawing functions. The following program will draw circles wherever the user moves their mouse on the Processing canvas:

```
def setup():
    size(200, 200)

def draw():
    ellipse(mouseX, mouseY, 10, 10)
```

We can see above that instead of passing in **fixed** arguments to the `ellipse()` function, like we did before, we're now passing in the *x* and *y* coordinates of the mouse. Recall that the `draw()` function is called automatically by Processing 60 times per second. Whenever that happens, Processing checks where the mouse is at the time that the function call happens and lets us access that information using `mouseX` and `mouseY`. So each time the `draw()` function is called, the mouse cursor may be at a different location and we get a new circle drawn at that location.

If we want to play around with this a little, we can adjust how frequently Processing calls the `draw()` function by using the `frameRate()` function. `frameRate()` requires a single number as an argument and that is the number of times per second that Processing should call the `draw()` function. If we call the `frameRate()` function with an argument of 1, then Processing will only call `draw()` — and consequently, only update its canvas — once per second. This is really quite slow, as you will see if you try running the following program:

```
def setup():
    size(200, 200)
    frameRate(1)

def draw():
    ellipse(mouseX, mouseY, 10, 10)
```

Due to the very slow frame rate, with this program you can move the mouse quite a long way before Processing gets around to drawing a new circle.

6.2.3 Event handling functions

Event handlers are special functions in an interactive system that are designed to process user actions. Unlike the `draw()` function, which gets called automatically and continually no matter what the user does, event handlers only get called when specific events occur. Just like with `draw()` and `setup()`, Processing has set aside a list of special function names to be used as event handlers. We have to define these functions in our programs, but as long as we use the appropriate function name, Processing will call these functions for us whenever the relevant events occur.

The mouseClicked() function

The `mouseClicked()` function is automatically called once each time the mouse is clicked (e.g. pressed down and then released). Suppose we wanted to write a Processing program in which the user could move the mouse around and draw a circle anywhere they clicked. The following program would do exactly that.

```
def setup():
    size(200, 200)

def mouseClicked():
    ellipse(mouseX, mouseY, 10, 10)

def draw():
    return
```

The only thing thing funny you might note is the word `return` in the body of the `draw()` function. Processing requires the `draw()` function to be there in order to update its canvas, but in this case, we don't really want the function to do anything. So here, the `return` is just a Python keyword that says the function body is over and nothing more needs to be done.

The keyPressed() function

The `keyPressed()` function is automatically called whenever a key — any key — on the computer's keyboard is pressed. Let's say we wanted to write a Processing program where the user could click to draw circles on the screen, but also give them the option to erase everything and start over by hitting the spacebar. The following program would do the trick.

```
def setup():
    size(200, 200)
    background(210)

def mouseClicked():
    ellipse(mouseX, mouseY, 10, 10)

def keyPressed():
    background(210)

def draw():
    return
```

This program is nearly identical to our previous one, except that now we've added the `keyPressed()` function. In that function definition, we just call the `background()` function to erase the screen and replace it with a fresh, empty background. This will happen any time the user hits the spacebar, or indeed, any key on their keyboard. There are ways to determine exactly which key was pressed, but we'll leave that for later.

7 — Data and Data Types

Learning Objectives

After studying this chapter, a student should be able to:

- distinguish between atomic data and compound data
- describe what a data type is
- describe what a literal value is in Python
- give examples of literal values corresponding to integer, floating-point, and string data
- use the `print` syntax to display literal values on the console

7.1 Data

Data is information. Computer programs need data to do anything useful. Data can take many forms (numbers, text, pictures, etc.), but ultimately, at a low enough level of abstraction, all data is numbers because that is what computers know how to store. It is abstraction that makes it appear that we can store things more interesting than numbers, such as images, video, text, web pages, etc. These things are all just large collections of numbers interpreted in different ways — the different interpretations are abstractions! At an even lower level of abstraction, all data is just sequences of 0's and 1's, because computer hardware stores data as binary numbers using different electric voltages to represent the binary digits 0 and 1. Fortunately, computer programmers don't have to work at such a low level of abstraction. In the rest of this section we'll look at the kinds of data we, as programmers, can use.

7.1.1 Atomic Data

Atomic Data is the smallest unit of data that a computer program can define. The word "atomic" derives from the word "atom". At one point in the history of chemistry, atoms were believed to be the smallest indivisible pieces of matter in the universe. In computer science, the word "atomic" is often used to refer to something that is indivisible or cannot be made smaller. By this, we don't mean "smaller" in the mathematical sense; just because you can make the number 10 "smaller" by dividing it by 2 doesn't mean that the value 10 isn't an example of atomic data. Rather, we mean "smaller" in the sense of simplicity. There's no simpler way to express the number 10 than simply writing 10, so therefore it's atomic.

7.1.2 Compound Data

Compound data is data that can be subdivided into smaller pieces of data which are organized in a particular way. An example of compound data is a list. A list consists of several pieces of data which have a specific ordering. The data items that comprise a piece of compound data may themselves be either compound or atomic. For example, we could imagine a list of numbers. The list itself is compound data, while each piece of data in the list is atomic data. We could also imagine a list of lists of numbers. In such a case, the list of lists is compound data, and each piece of data in the list is itself an example of compound data whose individual pieces are, in turn, atomic data.

Don't worry too much if the notion of "list of lists" makes your head hurt! We'll revisit the idea of compound data in more detail later. For now, the only type of compound data we will be using are *strings* (which we will define momentarily in 7.1.3).

7.1.3 Data Types

In a computer program, every piece of data, compound or atomic, has a *data type*. For one last moment, let us recall that data in a computer is, at a very low level of abstraction that we don't usually worry about, made up of binary 0's and 1's (we call such numbers *bits*). The *data type* of a piece of data tells the computer how to interpret those bits. For example, the exact same sequence of 0's and 1's could be interpreted as a character (or letter) from your keyboard, as a whole number, or as a fractional number. The computer needs to know which interpretation to choose, so that's why Python (and most other programming languages) have a notion of data type.

Atomic Data Types

In this section we describe the most commonly used atomic data types in Python.

Integer Integer data are whole numbers, that is, numbers without a fractional component. Integers include both positive and negative numbers, as well as the number zero. In Python, there is no limit to the size of an integer number.

Floating-point Floating-point data are real numbers, that is, numbers that are not necessarily whole numbers, such as the number 42.5. Floating-point data in Python (and any other language) have a limited precision, and limited range. This means that numbers with infinite representations, such as 1/3=0.33333..., $\sqrt{2}$, or π, cannot be represented exactly, and we can't represent numbers that are too big either. In Python, floating point numbers can range between 10^{-308} to 10^{308} (positive or negative) with at most 16 to 17 digits of precision. For brevity, we will sometimes refer to floating point numbers simply as *floats*.

Boolean Boolean data can only be one of two values: `True` or `False`. Note again that capitalization matters – `true` and `false` are not valid boolean values.

Compound Data Types

In this section we briefly describe some of the standard compound data types that are built into Python. However, we will save the the details of most of these until later chapters.

String Strings are sequences of characters (e.g. letters of the alphabet, digits, and punctuation) and are usually used to store text. We'll say more about strings and characters later in this chapter.

List Lists are a sequence of data items. Each item in a list can be of any data type. We'll discuss lists in more detail in Chapter 14.

Dictionary A dictionary consists of key-value pairs in no particular order. You can look up values by their key. We'll discuss dictionaries in more detail in Chapter 16.

7.2 Literals

A *literal* is a number or string written right into the program by the programmer, that is, **literally** typed right into the program's code such as `42`. In fact, we've already been using literals in most of the sample programs we've seen so far. When we use a function call like `background(200)`, the `200` is a literal value that we're using as an argument to the `background()` function.

When we use a literal in our programs, Python needs to know the literal's data type. We communicate the correct data type to Python by the way we write the literal, as follows:

Integer literals: Any number written without a decimal point is an *integer literal*. Thus, `42`, `-17`, and `65535` are integer literals.

Floating-point literals: Any number written **with** a decimal point is a *floating-point literal*. Examples are: `42.0`, `-9.8`, and `3.14159`. Note with care that even the literal `42.` (decimal point included) is a floating-point literal because it contains a decimal point. An empty sequence of digits after the decimal point is different from no decimal point at all! We can try this out in Processing using the `print()` function:

```
print(42.)
```

If you run the program above, you'll get `42.0` printed to the console.
Floating-point literals can also be written in scientific notation. For example, the speed of light is 3×10^8 m/s, a quantity which can be written as the literal `3e8`. Again, we can try this with the `print()` function:

```
print(3e8)
```

This program will print the value `300000000.0` to the console. Note the decimal point preceding the rightmost zero in the sequence. Literals written in scientific notation are **always** floating point, never integers.

String literals: Recall that in the previous section we said that strings are sequences of characters. A string literal is specified by enclosing a sequence of characters with a pair of single or double quotes. `"Hello world."` and `'The night is dark and full of terrors.'` are both examples of string literals. Strings can contain spaces because spaces are characters too. Any symbol that appears on the keyboard is a character.[1] Note that there is a difference between the literals `'7'` and `7`; the former is a string literal and does not actually have the numeric value of 7, while the latter is an integer literal, which does. Similarly the literals `"3.14159"` and `3.14159` are different. The former is a string literal, which does not actually have the numeric

[1] Other, stranger things can be characters too, but we'll avoid that discussion for now.

value 3.14159, and the latter is a floating-point literal, which does. However, ’Bazinga!’ and "Bazinga!" are exactly the same.

So why have two ways of writing string literals? It is so that we can conveniently include single or double quotes as part of a string (after all, the " symbol is on your keyboard, so it’s also a character!). The string ’The card says "Moops".’ is enclosed in single quotes and contains two double quotes as part of the string. The single quotes are not part of the string, they’re special syntax that tells Python that everything between them should be interpreted as a single piece of string data.

```
print(’The card says "Moops".’)
```

If we run the program above, the phrase The card says ‘‘Moops’’. will be printed to the console. Notice that the print() function didn’t print the enclosing single quotes, because they weren’t part of the data.

By contrast, let’s look what happens with the following program:

```
print("The card says "Moops".")
```

If you try to run this program in Processing, you’ll get an error. The reason for this is because Python interprets the characters between the first two double quotes (the first one and the one right before the word Moops) as a string literal. Then the word Moops makes no sense to Python because Python thinks the string literal is finished. So Python tries to interpret Moops as part of the Python language, which it isn’t, so Python doesn’t know what to do and gives up.

The bottom line is that you can write single quotes inside string literals enclosed in double quotes, and double quotes inside string literals enclosed in single quotes.

So should you use single or double quotes for strings? Well, there’s no right or wrong answer to this question. Unless you need single or double quotes within a string literal, it doesn’t matter. Normally, one chooses to use either single or double quotes as one’s “default” style, and only uses the other when necessary.

8 — Variables and Expressions

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a variable is
- explain the naming rules for variables
- describe what an expression is in Python
- list the basic arithmetic operators in Python
- compose valid arithmetic expressions in Python using operators and literals
- compose valid expressions in Python using variables
- use the `print` syntax to display literal values and the values of variables on the console
- use Processing event handling to save user actions to a variable

8.1 Variables

Variables are a way of giving names to data. Giving a name to data allows a program to operate on different data values each time a problem is run. We can then ask Python to do something to the data with a certain name. If we only had literals, we could only ask Python to do something with a specific literal data value.

We've already seen a hint of just how powerful variables are when we used `mouseX` and `mouseY` in Chapter 6 to draw shapes in different places depending on the mouse position. It turns out `mouseX` and `mouseY` were just pre-made variables that Processing sets up for us. Now we're going to learn how to use variables in general.

8.1.1 Variable Names

The essence of variables is assigning symbolic names to data. The names can be almost anything, but there are a few naming rules we have to follow. Variable names (also called *identifiers*) in Python have to follow these constraints:

- may contain letters or digits, but cannot start with a digit;
- may contain underscore (`_`) characters and may start with an underscore; and
- may not contain spaces or other special characters.

Thus, `KyloRen`, `IG88`, and `luke_Skywalker` are valid variable names, but these are not: `Luke+Leia_4_Evar` (contains special character `+`), `2ManyStormTroopers` (starts with a digit), and `Han Shot First` (contains spaces).

In addition, Python keywords (like the `def` keyword for defining functions) cannot be used as variable names. Function names, on the other hand, **can** be used as variable names. So even though Processing has a function called `line()`, we could also make a variable named `line` as well, and the two would have nothing to do with each other.

As long as we follow the rules above, the computer doesn't care how we name our variables. But humans who read your computer code certainly care! In general, you should strive to choose variable names that give an indication of how the variable will be used. For example, the Processing authors could have chosen the variable name `cookie_crumb` instead of `mouseX`, but that would have been a poor choice. After all, the variable `mouseX` has absolutely nothing to do with cookies and everything to do with the x-coordinate of the mouse! This might seem obvious, but it's not at all uncommon for novice programmers to figure out that they're going to need five variables in their program, so they name those variables `a`, `b`, `c`, `d`, and `e`. Don't do this! If you can't come up with a better naming scheme for your variables than an alphabetical listing, it means you haven't sufficiently understood the algorithm you are trying to code. Spend the time to understand the algorithm first, **then** worry about trying to code it in Python.

8.1.2 Variable Assignment

Variable assignment is the process of giving a valid name to a piece of data, and it is one of the most central tasks in a programming language. Python uses the equal sign (=) to *assign* a variable name to a value. Here are some examples of variable assignment:

```
x = 5      # assign the name x to the integer 5.
y = 42.0   # assign the name y to the floating-point number 42.0

# assign the name errorMessage to the string: "That didn't work!"
errorMessage = "That didn't work!"
```

One of the trickiest things for novices to understand about variable assignment is that it is **not** a symmetric operation. In mathematics, the statement $1+2=3$ is a statement of fact. In that context, the $=$ sign is saying "the two things on either side of me are the same". In Python (and most other programming languages), the $=$ sign is not a statement of fact, but a **command** to create an association. Translated into English, the Python statement `x = 5` means "assign the variable name `x` to the value 5". Thus, from this point on, when you refer to the variable `x` you are in turn referring to the value 5 (but not the other way around!). The thing on the left of the $=$ sign **must** be a variable name and the thing on the right of the $=$ sign **must** be a value. As a result, not only are `x = 5` and `5 = x` **not** the same statement, but `5 = x` is not even a valid Python statement at all, because `5` is not a valid variable name.

The data or value to which a variable refers to always has a type, but you cannot tell from the variable name what type of data it refers to. You can even change the type that a variable refers to:

```
x = 10;      # x refers to the integer 10
x = 10.0;    # now x refers to the floating-point value 10.0
```

There are ways to determine the type of data that a variable refers to, but we'll leave that for a later discussion. For now, just be aware that there is no way to guarantee that a variable always refers to data of a specific type. You can write a program so that a variable is always **supposed** to be of a certain type, but the type might change as a result of an error in your code, and there is no way to force Python to notify you of this.

8.2 Expressions

Expressions in a programming language are combinations of data and *operators*. *Operators* are special symbols which have specific meaning in the programming language. These operators perform computations on one or more pieces of data to produce a new piece of data. When all of the computations associated with operators in an expression have been carried out, the result is a new piece of data whose value is the result of the expression.

This might sound pretty abstract, but you'll find that you're already familiar with a lot of the operators that we're going to discuss. For example, `1 + 2` is a simple and valid Python expression. The + sign is the operator in this expression, and it works as you might expect; it combines the two data values, `1` and `2`, using the rules of addition. The *value* of this expression is the integer number `3`.

8.2.1 Literals as Expressions

Literals, just by themselves, are one of the simplest forms of expressions. The value of an expression containing a single literal is the value of the literal itself. In other words, it would be correct to say that when Python evaluates the expression `42`, it produces the value of `42`. This framing might seem redundant or even a little silly to you at this point, but this sort of thinking is very common in computer science. We start from something that seems trivially true and build up complexity from there.

8.2.2 Variables as Expressions

Just like a single literal, a single variable is an expression all by itself. The value of such an expression is the data value that the variable refers to. So if we tell Processing to print a variable to the console, it will display the value of that variable.

```
x = 10
print(x)
```

The code above will print `10` to the console.

Since variables can be expressions (they evaluate to a value), they can actually go on the **right** side of an assignment command, like so:

```
x = 10
y = x
```

After executing the code above in Python, both `x` and `y` will refer to the value 10. It's important to note that the statement `y = x` doesn't establish any kind of permanent equivalence between `x` and y. It just associates y with whatever value is associated with x at the time that the statement is excecuted. For example, consider the following code:

```
x = 10
y = x
x = 2
print(x)
print(y)
```

This program will print first a 2 then a 10 to the console. The value of y didn't change just because x later got associated with a new value.[1]

8.2.3 Operators

Operators can be used to write expressions that compute new values from existing pieces of data. We say that the operator *operates* on these pieces of data. For example, we mentioned earlier that one common operator is the + sign, and so the expression 2 + 3 has the value 5. The data items that an operator operates on are call *operands*. Operands can be any expression. Most of the operators we will see are *binary operators* because they require two operands.[2] Operands need not be literals, they can be variables too:

```
x = 2
y = 3
z =  x + y
print(z)
```

Since x refers to the integer 2, and y refers to the integer 3, the value of the expression x + y is 5, which is the value that gets associated with the variable z and then printed to the screen.

This is also a good time to note that a variable name cannot be used in an expression if it has not been assigned to a value. For example:

```
x = 2
a = x + z
```

In the above example, when we try to add together the value referred to by x and the value referred to by z (which refers to no value because none was assigned), Python cannot perform the addition operation, and issues a NameError which is its way of saying that the identifier z was never assigned to a value.

Arithmetic Operators

The basic arithmetic operators in Python are summarized in the following table:

[1] This is definitely true with atomic data. The situation is a bit more complicated when we start dealing with lists, but that won't be until a little later.

[2] Here the word "binary" only conveys that the operator requires two operands, as opposed to *unary* operators which only require one operand. Do not confuse binary operands with binary numbers — the latter are entirely different.

Usage	Description	Example Expression	Value
`x ** y`	Exponentiation; `x` to the power of `y`	`2 ** 5`	32
`-x`	Negation	`-42`	-42
`x * y`	Multiplication; `x` times `y`	`6 * 7`	42
`x / y`	Division; `x` divided by `y`	`8 / 4`	2
`x % y`	Modulo; remainder after integer division of `x` by `y`	`6 % 4`	2
`x + y`	Addition; `x` plus `y`	`3 + 6`	9
`x - y`	Subtraction; `x` minus `y`	`2 - 7`	-5

When using arithmetic operators in Python, the usual order of operations that you're used to from math applies and is reflected in the table above. The operators higher in the table are evaluated before operations lower in the table. Multiplication, division, and modulo have the same precedence and if more than one of these appears in the same expression, they are evaluated from left to right. Addition and subtraction have the same precedence (but lower than the others) and again, are evaluated from left to right.

As you might expect, you can override the normal order of operations by enclosing things in parentheses. The parentheses have higher precedence than any of the operators. As an example, the value of the expression `2 * 4 + 10 * 3` is 38. Without parantheses, multiplication happens first, so the previous expression becomes `8 + 30`. By contrast, the value of the expression `2 * (4 + 10) * 3` is 84. Because of the parantheses, the addition `(4 + 10)` happens first, so we now have `2 * 14 * 3`, which is 84.

Mixing Number Types

Recall that in Python, integers and floating point numbers are completely different data types, and we have special notation (using a decimal point or not) to tell Python what kind of number we want to use. So what happens when we try to combine an integer and a floating point number in an expression? Does Python even allow this? The answer is yes, but it can result in behaviour that is a bit tricky.

In general, the following rules apply when mixing integers and floats.

- If you combine two integers, the result is an integer
- If you combine two floats, the result is a float
- If you combine an integer and a float, the result is a float

So the value of the expression `2 + 1.5` is 3.5, which is probably what you'd expect. The tricky part is that the value of the expression `3/2` is **not** 1.5! It can't be, since both 3 and 2 are integers and 1.5 is a float. By the rules above, when we combine two integers, the result must also be an integer. Therefore, in Python, `3/2` is just 1. This is an example of *integer division*, which is common in many computer languages. You can think of integer division as being like normal division, except that we **drop** (not round!) all decimal places from the answer. So in integer division, `5/2` is 2, and `9/10` is 0.[3]

[3] This behaviour changes slightly in different versions of Python, which isn't relevant to this course, but is relevant if you go on to take CMPT 141, or start using Python on your own.

Modulo

On the subject of integer division, you might have noticed the modulo operator (represented by %) in the table above. Modulo is the complement of integer division; it gives us the *remainder* of performing division with whole numbers. For example, take the expression 5%2. When we limit ourselves to just integers, 2 goes into 5 just two whole times, after which there's a remainder of 1 that hasn't been accounted for. The modulo operator gives us that remainder, so 5%2 is 1.

Operators on Strings

Some arithmetic operators can be applied to string operands, but their meanings are different. The "addition" of two strings results in their concatenation, which means just joining them together one after the other. The "multiplication" of a string and a number *n* concatenates the string with itself *n* times. For example, the expression `'Winter' + 'is' + 'coming!'` produces the string value `'Winteriscoming!'`. The expression `'Na' * 8 + ' BATMAN!'` produces the string value `'NaNaNaNaNaNaNaNa BATMAN!'`.

8.3 Using Variables in Functions

Recall that functions give us a mechanism by which we can encapsulate algorithms. Since variables are very useful for writing almost any kind of algorithm, we can certainly use variables in functions:

```
def setup():
    size(200, 200)

def draw():
    size = 10
    ellipse(100, 100, size, size)
```

In the preceding program's `draw()` function, we associate variable `size` with a value of `10`. We then make a function call to `ellipse()` using the value of the `size` variable as both the 3rd and 4th arguments in the function call. This is the same as just calling `ellipse(100, 100, 10, 10)`, since we gave `size` a value of 10.

Since the purpose of functions is encapsulation, any variables that we use as part of a function are **only visible to that function**. For example, the following code will result in an error:

```
def setup():
    size = 10
    size(200, 200)

def draw():
    ellipse(100, 100, size, size)
```

Since the `size` variable is assigned a value in the `setup()` function, when we try to use it in the `draw()` function, Python will complain that `draw()` doesn't know anything about a variable called `size`.

Finally, it's worth noting that function parameters, which we already learned about, are also really just variables with one special property: when the function is called, the parameter variables are automatically initialized with the argument values that were used in the function call. After that, they're just normal variables in every way. The following program re-uses a function parameter for drawing a donut (or a circle within a circle):

```
def donut(size):
    ''' draws a white donut with a black donut hole
    The parameter size gives the radius of the donut '''
    fill(255)
    ellipse(50, 50, size, size)
    fill(0)
    size = size / 2
    ellipse(50, 50, size, size)

donut(30)
```

Notice how in the program above we were able to change the value associated with the `size` variable by setting it to one-half of its original value and then use it again in the second function call to `ellipse()`.

8.3.1 Handling User Input using Variables

We already know that in Processing, we can detect user actions such as mouse clicks and keys pressed on the keyboard via event handler functions. The thing about computers, though, is that they don't actually remember anything unless we explicitly tell them to do so. So how do we get our program to remember that a key on the keyboard was pressed, or better yet, which key was pressed? The answer is that we'll store that information in a variable.

Let's say, for example, that we wanted to write an interactive program that allowed the user to type in their name. We can do that by waiting for the user to type on the keyboard and concatenating each character typed onto a string variable. The Processing code would look like this:

```
def keyPressed():
    name = name + key
```

Here, we're making use of another pre-defined variable that Processing provides for us called `key`. `key` is just a variable that holds the value of the most recently pressed key on the keyboard. So every time the user presses a key, Processing automatically calls the `keyPressed()` function, during which we add the key that was pressed to the end of a string variable called `name`.

Now let's say that as the user types in their name, we want to be able to show them what they've typed. The obvious way to do that is to define the `draw()` function and display the name using a function call to the `text()` function. But there's a problem here: as we just learned in the previous section, variables are only visible to the function that uses them. So `draw()` won't know anything about the variable `name` that we use in the `keyPressed()` function.

The way to get around this for dealing with user actions is to use the `global` Python keyword. This allows us to initialize a variable *outside* of any function and then make that variable visible to any function that needs to see it. Thus, we could write our program that prints out the user's name as they type it as follows:

```
name = "" # initialize with the empty string

def keyPressed():
    global name
    name = name + key

def draw():
    global name
    text(name, 20, 20)
```

By including the statement `global name` in both of the function definitions, we are telling these functions that a variable called `name` exists at a global level, and that they can see and modify this variable.

In general, the `global` keyword should be used sparingly, since it wreaks havoc with the concept of proper encapsulation for functions. In our Processing programs, we should mostly only use global variables when dealing with event handlers like `keyPressed()` or `mouseClicked()`. The global variables allows us to remember what happened as a result of user inputs, and we can then use this information in other functions like `draw()`.

9 — Functions with Outputs

Learning Objectives

After studying this chapter, a student should be able to:

- describe the role of a function's return value
- compose functions in Python that perform a subtask and return the result
- explain the role and behaviour of the return statement
- save the result of a function call to a variable
- use functions with return values in expressions
- trace program behaviour in and out of nested function calls

9.1 Functions That Compute Values

In Chapter 4, we introduced functions as a mechanism for supporting encapsulation. Functions allow us to write an algorithm that solves a specific sub-problem, give that algorithm a name, and henceforth keep the algorithm logically separate from the rest of a program. We provide inputs to functions by passing arguments when we make function calls.

In this chapter, we're going to learn how to write functions that have output as well. In a computer program, function output always takes the form of data. A function with output typically calculates some kind of result or answer, which it gives back whenever the function is called. An example of such a function is Python's `max()` function. `max()` takes numbers as inputs and gives us back the single largest number from among its inputs, like so:

```
biggest_number = max(1, 4, 6)
print(biggest_number)
```

The program above prints the value `6` to the console, since 6 is the largest of the three values that were given as inputs to `max()`.

We'll come back to explaining how to use functions with outputs shortly. First, we're going to see how to return output when we write our own function definitions.

9.1.1 The Return Keyword

The `return` keyword is used to specify the output of a function. We write the keyword `return` followed by an expression (remember, an expression *can*, but doesn't have to, be a lone variable or literal value); the value of the expression is the end result that your function is "sending back" to wherever it was called from in your program.

For example, here is a function that calculates the area of a square and returns the answer.

```
def square_area(width):
    """ Returns the area of a square.

    width: the length of the square's side """
    return width * width
```

As you can see, the function body in the code above is one line (not including the docstring) that returns the result of the expression `width * width`. Calculating the area of a square is pretty straightforward, so there was nothing else we needed to do as part of the function's body. But we could have had any number of statements in between the function header and the return statement.

We will see later that functions may have more than one return statement[1]. As soon as a return statement is executed, regardless of where it appears in the function, execution of the function immediately ceases (even if there are lines of code after it!), the value of the accompanying expression is returned, and execution of the program continues from the line immediately after the call that invoked the function. For this reason, the return statement will usually be the last line in any function that you write.

Returning Nothing

If a function does not need to return a value, then simply do not include a return statement. We've been doing this already for all the functions that we've used prior to this chapter. If we're being really precise, a function without a return statement is formally called a *procedure*, but we're not going to worry about the formality regarding that definition in this class.

We can also return nothing by just using the word `return` all by itself on one line. We did this when we needed an effectively empty function body for `draw()`. When Python executes such a statement, it will immediately "jump" out of the function and back to wherever the function was called from, but no value will be returned.

9.1.2 Return Statements Versus System Feedback

We mentioned back in Chapter 4 that drawing a line or printing text to the computer's monitor doesn't count as "function output". That's because function output is done via the `return` keyword, and we can only return data that way. "Make a red circle appear on the monitor" isn't data[2], it's a command. As such, there's an important distinction between function output, which is just data that the computer passes around to itself, and system feedback, which is the manipulation of the computer's peripheral devices in order to give information to a human user.

[1] Some programmers are of the opinion that writing functions with more than one return statement is bad style!

[2] Well, it *could* be string data, but then it's just a sequence of characters, not an instruction to the computer to actually do something.

9.2 Functions as Expressions: Obtaining/Using a Function’s Return Value

Function calls can be used as expressions when the function being called returns data (i.e. yields data in the form of function output). Like all other expressions, they have a value. The value of a function call is the return value of the function, so therefore we can use these function calls anywhere that we use values.

One of the most important things we can do with the value returned by a function call is to store it in a variable. After all, computers don’t remember anything unless we tell them to, and what would be the point of calculating some value only to immediately forget about it? We’ve already seen an example of this, but here it is again:

```
biggest_number = max(1, 4, 6)
print(biggest_number)
```

The function `max()` returns the largest number from among its arguments. Because the result of the function call is a value, we can put the function call itself on the right hand side of an assignment statement. The resulting value will become associated with the variable `biggest_number`, which we can then use however we like.

We can also use function calls as operands of an operator, like so:

```
sum_of_biggest = max(1, 5) + max(8, 10)
print(sum_of_biggest)
```

The code above will print the value `15` to the console. This is because assignment is the last operation to occur in a Python statement; the expression to the right of the = sign gets completely evaluated first. The first function call to `max()` produces the value `5`, and the second call produces the value `10`. `5 + 10` is `15`, so this is the value that gets associated with the variable `sum_of_biggest`.

We can also use function calls as arguments to another function call. We’ll do that here with the `min()` function, which is exactly the same as `max()` except that it returns the smallest of its arguments rather than the biggest:

```
def draw():
    background(215)
    line(0, 0, min(mouseX, 50), min(mouseY, 50))
```

In the code above, we’re using function calls to `min()` as the 3rd and 4th arguments to the `line()` function. This will cause Processing to a draw a line from coodinate (0, 0) to wherever the mouse is. However, if one of the mouse coordinates is higher than 50, the value 50 will be used instead of the mouse’s actual location due to the `min()` function calls. This results in a program where the user utilizes the mouse to draw a line starting from the top left corner of the canvas, but that line is restricted to a 50x50 area.

9.2.1 More Built-In Python Functions

We conclude this section with a few examples of commonly used built-in Python functions.

Name	Number of Arguments	Description and Example
`max`	≥ 2	Returns the maximum value of all arguments. Accepts any number of arguments. `biggest = max(10, 15, 42,  19)`
`min`	≥ 2	Works like `max` but returns the minimum value of all arguments. Accepts any number of arguments.

`len`	1	Returns the number of characters in a string.
		`print(len(‘cookie’))`
		`6`

`type`	1	Returns the data type of its argument.
		`print( type(5) )`
		`<class ’int’>`
		`print( type(2.3) )`
		`<class ’float’>`
		`print( type(’foo’) )`
		`<class ’str’ >`

`int`	1	Converts the argument to the integer data type (if possible) and returns the result. Strings can be converted if they contain only digits 0–9.
		`x = int(42.0)`
		`x = int(’42’)`

`float`	1	Similar to `int`; converts the argument to the float-point data type (if possible) and returns the result.
`str`	1	Similar to `int`; converts the argument to the string data type and returns the result.

9.3 Nested Function Calls

There is no limit to the number of function definitions you can have in one program. And so long as functions are only called after they are defined, there’s no limit to where or how many times a function can be called. The consequence of this is that we can define some function `A()`, and then define some function `B()` which, as part of its function body, includes a function call to `A()`. We could even then define a function `C()` which makes function calls to both `A()` and `B()` - even though `B()` itself includes a function call to `A()`!! All of this is perfectly fine from the computer’s perspective. The only thing that might make it a bit tricky for humans to follow is that we’re used to reading text top-down and left-to-right[3], when in fact that won’t be the order in which the statements in our computer programs are executed once we start using multiple function calls.

[3] English-speakers are used to reading this way, in any case.

Let’s illustrate this process through an example. Suppose we have the following program:

```
def A():
    print(’A’)

def B():
    print(’B’)
    A()

def C():
    print(’C’)
    A()
    B()

C()
```

The only statements in this program that actually **do** anything visible are the function calls to `print()`, which just print the given text to the console one line at a time. Let’s unroll this program and take a look at the order in which the print statements are actually going to happen when the program is run.

The first thing that happens is a function call to `C()`, so we take a look at `C()`’s function body and start executing the statements there one at a time. The first line there is `print(‘C’)`, so a `C` will appear on the console. The next line is a function call to `A()`, so the computer will look at `A()`’s function body. There’s only one statement there, which is `print(‘A’)`, so that’s the next thing that will happen. After that, `A()`’s function body is finished, so we return to the middle of `C()`’s function body, since that’s where we came from when `A()` was called. Next is a funciton call to `B()`. The first statement in `B()` is `print(‘B’)`, so that happens next, followed by a function call to `A()`. As before, `A()` has only one statement, so `print(‘A’)` will be executed at this point. Then `A()` is finished, so we return to where `A()` was called. `B()` doesn’t do anything else after calling `A()`, so we return to where `B()` was first called in `C()`’s function body. `C()` doesn’t do anything either after calling `B()`, so we return to where `C()` was called. There are no more lines of code after the function call to `C()`, so the program is finished.

The following is a list of the order of `print()` statements from the example we just walked through:

```
print(’C’)
print(’A’)
print(’B’)
print(’A’)
```

What would the order of execution of the print statements be from the previous program if we swapped the lines `B()` and `print(‘C’)` within `C()`’s definition? We’ll leave that walkthrough to you as an exercise (see the footnote here[4] to check your answer).

[4] `print(‘B’)`, `print(‘A’)`, `print(‘A’)`, `print(‘C’)` in that order

10 — Libraries

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a library is and why one would want to use one
- author Processing code that imports Processing libraries
- identify and author Processing programs that use library functions

10.1 Libraries

In a programming language, *libraries* are files that contain function definitions that add additional, and much more powerful, features to the language. By default, most basic languages provide only very fundamental building blocks for programs. Libraries are a way for people to share functions that they have written so that other people can use them in their own programs. Viewed another way, libraries contain *abstractions* of algorithms that we can use in our own programs without having to understand how they work.

10.2 How to use libraries in Processing

Libraries are stored in separate files from our own programs. Thus, in order to use the functions defined by a library, we have to tell our own program to look in those files and read those definitions. Sometimes, we even have to first download the library files themselves from wherever they're stored on the internet. Fortunately, Processing provides us with an easy interface to do this.

On your Processing menu bar, under the 'Sketch' menu, you'll see an 'Import Library' option. That's what we want. Hovering your mouse over it will bring up a sub-menu that shows a list of items. The very top item in this list is the command "Add Library"; everything else in the list is the name of a library that's ready to be added to your program. If the library we want isn't on that list, we click on the "Add Library" option, which will then display a pop-up box that shows many, many different libraries. These are libraries that Processing knows how to find on the internet and can download and install for you if you like. We're going to use the "sound" library, which will let us use sound files and other forms of audio feedback in our Processing programs. Once we've successfully downloaded the "sound" library, it should now be included in the list of libraries under "Import Library". If we click on it, it should add a line that looks like this to the very top of our program.

```
add_library('sound')
```

The presence of that one line will allow us to use anything that's in the "sound" library in our own program. For example, let's say we have a sound file (we'll say it's called "sample.mp3") that we would like to play every time our program starts. The very first thing we have to do is put the .mp3 file itself in the right place, so that our Processing program will be able to find it. We can do this in Processing by once again accessing the "Sketch" menu and selecting "Show Sketch Folder". In the resulting folder, we need to create a new subdirectory called "data". We can put our sample.mp3 file in that "data" directory (indeed, any sort of sound or image data that our Processing programs might need can go there). Once we've done that, we only need the following few lines of code to use the sound file.

```
add_library('sound')

def setup():
    start_sound = SoundFile(this, "sample.mp3")
    start_sound.play()

def draw():
    return
```

In the code above, we used two functions that came from the "sound" library. First, we called a function called `SoundFile()`, which returned for us a piece of data that represents our sound file, which we associated with the name `start_sound`. We then used the function `play()` to actually play the sound.

Two things probably look odd to you about the code above. The first is that `SoundFile()` had two arguments. The second argument, "sample.mp3", is just the name of the sound file we want to access, which you probably already guessed. But the first argument was simply the word `this`. Explaining exactly why we need this word is well beyond the scope of CMPT140, but for now, it suffices to say that you'll often see it if we use library functions to deal with "complicated" things like sound and video.

The second odd thing is that when we called `play()`, we put it right after the variable name `start_sound`, joined by a period. We haven't seen this way of calling functions before, so we're going to explore it in a bit more detail in the next section.

10.3 Objects and Methods

We learned earlier about atomic data, such as integers, floats and Booleans, which are the simplest possible data types that exist in a programing language. We can have more complicated data types as well, which we refer to collectively as *objects*. Objects are pieces of data that often (though not always) represent a high-level concept of some kind. In our example from the previous section, we used the function `SoundFile()` to return a "sound object" that represents all the audio data associated with our .mp3 sound file.

The reason we use objects is that they have a number of functions associated with them that let us do some useful things. These are similar to the functions we already know about, but the syntax for calling them is slightly different. Because of this different syntax, we use a different name for them: *methods*. Methods are similar to functions in that they still accept arguments, and they still sometimes, but not always, have a return value. The difference is that methods are always called with respect to a particular object. You can think of a method call as "asking a particular object to do something to itself." Syntactically, we call a method by first using the **name** of a particular object, followed by a period, followed by the method name and arguments (including parentheses). The general form looks like this:

```
# general form for calling object methods
# assume my_object refers to an object of some kind
my_object.function_name(arguments...)
```

So basically, objects are pieces of data that know how to do things to themselves. Atomic data types, like integers and floats, aren't objects, so they don't know how to "do" anything (i.e. they don't have methods).

10.4 Finding Library Documentation

At this point, you may perhaps be wondering how one finds out about what libraries are out there, or what methods are available to a particular object. The short answer is: internet search. For example a search for "processing libraries" returns us a link https://processing.org/reference/libraries/, where you can see a list of all the Processing libraries that the Processing authors have documented.

While in general it can sometimes be hard to find documentation for libraries, and sometimes the documentation that does exist isn't as clear as you might like, rest assured that, for this course, we'll always tell you how to use a library function or object that we expect you to use, or at least tell you exactly where the documentation is.

11 — Conditional Branching

Learning Objectives

After studying this chapter, a student should be able to:

- identify and define the behaviour of relational operators, logical operators, and Boolean expressions in Python
- identify and author correct Python language syntax for branching statements: if, if-else, if-elif-else, and chained statements
- design and author Python programs that use if, if-else, nested if, and chained-if statements

Way back in Chapter 1, we introduced the idea of *control statements*, which are statements in an algorithm that tell us under what conditions certain other actions should be carried out. For example, the following plain English algorithm uses a control statement:

```
try turning doorknob
if the door is locked:
    insert key into lock
    turn key
    remove key from lock
turn knob and open door
```

The algorithm accommodates for different situations by allowing whoever is following the algorithm to act differently based on their circumstances. We, the algorithm writers, do not know in advance whether our audience for this algorithm will be dealing with locked or unlocked doors, so we write the algorithm such that it can handle either case. This is called *conditional branching*, because depending on certain conditions, different branches of the algorithm will be followed when it is executed.

In this chapter, we will learn how to use conditional branching in Python.

11.1 Conditions

A *condition* is an expression that evaluates to one of two values: `True` or `False`. You may recall from Chapter 7 that we mentioned the *Boolean* data type; that's the type of data needed to evaluate a conditional statement.

The absolute simplest form of condition is just a literal Boolean value all by itself. But if that's all we had, we couldn't write very interesting or useful conditions. So we're going to introduce some operators that we can use to build expressions that produce Boolean values and are therefore suitable for use as conditions.

11.1.1 Relational Operators

A *relational operator* checks whether its operands satisfy a specified relationship and produces a Boolean value based on its assessment. Put another way, relational operators are a method of asking simple "true or false" questions about data. For example, the value of the expression `2 < 4` is `True`. This is because the < operator is the "less than" operator. The expression *x* < *y* has the value `True` if the value of *x* is smaller than the value of *y* and `False` otherwise. The following table lists several commonly used Boolean operators in Python.

Operator	Meaning	**Example**	**Result**
`==`	are the operands equal?	`42 == 42`	`True`
`!=`	are the operands unequal?	`42 != 42`	`False`
`<`	is the first operand smaller than the second operand?	`10 < 42`	`True`
`>`	is the first operand larger than the second operand?	`'Bill' > 'Lenny'`	`False`
`<=`	is the first operand less than or equal to the second?	`42 <= 42`	`True`
`>=`	is the first operand greater than or equal to the second?	`'R' >= 'Z'`	`False`

Notice how the operators work with non-numeric data as well, like strings and characters. In such cases the comparison is made lexicographically (dictionary ordering). `'Bill'` is not greater than `'Lenny'` because `'Bill'` comes before `'Lenny'` in dictionary ordering. For the same reason, the expression `'Bill' < 'Lenny'` has the value `True`.

Relational operators all have the same precedence and are evaluated from left-to-right. But all relational operators have a lower precedence than all arithmetic operators, which means arithmetic operators get evaluated first. Thus, the expression `5 + 5 < 10` is `False` because the addition happens first, resulting in the value 10. Since 10 is not less than 10, the < operator evaluates to `False`. Of course, we can always use parentheses to either force a particular ordering, or just to make the default ordering very clear. So `(5 + 5) < 10` is a perfectly valid expression, and still produces the value `False`.

11.1.2 Logical Operators

Logical operators (also called *Boolean operators*) are operators that act only on Boolean values. You can think of them as the Boolean equivalent to arithmetic operators, like addition or multiplication. The three logical operators are `and`, `or`, and `not`. Notice that these operators are just English words and not special symbols of any kind. This isn't true in all programming languages, but it is true in

Python. We can use logical operators to combine Boolean values to get new Boolean values, just like arithmetic operators let us combine numbers to get new numbers.

All logical operators have a **lower** precedence than relational operators. So that means that logical operators always get evaluated **after** relational operators.

The `and` Operator

The expression *x* `and` *y* has a value of `True` only if both *x* and *y* are `True`. In all other cases, such an expression has a value of `False`. Remember that *x* and *y* could be Boolean literals, Boolean values, Boolean expressions, or even a function call that returns a Boolean value. Here are some examples:

Expression	Value
`1 - 1 > 0 and -2 > 0`	`False`
`False and 'x' < 'y'`	`False`
`5 < 10 and 20 != 42`	`True`
`len('Skywalker') > 0 and len('Skywalker') < 10 and 'Ren' < 'Rey'`	`True`

Note the order of operations in the first example. The subtraction happens first, because it has higher precedence than all relational and logical operators. Then the two greater-than operators are evaluated because relational operators have higher precedence than logical operators. The last thing that happens is the `and` operator. Since both > operators result in `False`, the entire expression is `False`.

In the last example, the two `and` operators are evaluated left-to-right (recall that `len()` returns the length of a string). The result of the first `and` is `True`, which becomes the first operand to the second `and`, then `True and 'Ren' < 'Rey'` evaluates to `True`, so the whole expression evaluates to `True`.

The `or` Operator

The expression *x* `or` *y* has a value of `False` only if both *x* and *y* are `False`. In all other cases, such an expression has a value of `True`. Here are some examples of expressions using `or`:

Expression	Value
`5 < 7 or 0 == 0`	`True`
`7 < 5 or 0 == 0`	`True`
`2**5 < 16 or max(7, 42) == 7`	`False`

The `not` Operator

The `not` operator is a unary operator. It only takes one operand. The expression `not` *x* has a value of `True` only if *x* is `False`; it has a value of `False` if *x* is `True`. So `not` changes the Boolean value of its operand to the other Boolean value. Here are some examples:

Expression	Value
`not 42 < 0`	`True`
`not 6 == 6`	`False`
`not max(17, 50) > 80`	`True`

In the last example, the function call max has the highest precedence; it returns 50. The next highest precedence is the > operator (relational operators have higher precedence than logical operators), which results in False since 50 is not greater than 80, then not False results in True.

Mixing Logical Operators

We don’t want you to get the idea that you can only use one kind of logical operator per expression. You can mix them up as much as you like, but take care — **the logical operators do not have the same precedence!** The operator not has higher precedence than and which, in turn, has higher precedence than or. Take a look at these expressions:

Expression	**Value**
not 5 < 7 or 0 == 0	True
not (5 < 7 or 0 == 0)	False
len(’Vader’) < 7 or len(’Maul’) < 3 and ’Vader’ < ’Maul’	True
(len(’Vader’) < 7 or len(’Maul’) < 3) and ’Vader’ < ’Maul’	False

You might expect the first expression to have a value of False, because 5 < 7 or 0 == 0 is clearly True, and the not would change that to False. But the not operator has higher precedence than or. In this expression, the relational operators evaluate first, giving us not True or True. Now the not is applied to the first True, giving us False or True, which ends up as True. If we really want to apply not to the result of the or, we have to add parentheses, like in the second example. The relational operators still evaluate first, again giving us not (True or True). But now, because of the parentheses, the or evaluates next, which gives us not True, and ultimately False.

Note how in the third and fourth examples, if we want the or to evaluate before the and we have to use parentheses around the or expression. You can see that it matters because we get different answers depending on which of or or and evaluates first.

Variables in Relational and Logical Expressions

We also don’t want you to get the idea that you can’t use variables with these operators. In fact, in practice, most useful expressions will involve variables. There’s very little point in writing a program that contains an expression for which you, the programmer, already know the value ahead of time, because you could just substitute that value for the whole expression yourself! In any of the examples above where a literal appears in an expression, we could replace the literal with a variable. For example, a < b and c < d is valid. We just can’t evaluate this expression without knowing the values of the variables. Here’s a complete example where we associate the variable names with values and use them in a Boolean expression:

```
a = 1
b = 5
c = 2
d = 4
print( a < b and c < d )
```

The program above will print the value True to the console.

11.2 Branching and Conditional Statements

Now that we know how to ask questions about data using Boolean expressions, we can use the values of Boolean expressions to get our programs to perform different actions under different circumstances. This is called *branching* and it allows us to perform one block of code if a Boolean expression is `True`, and a different one if it is `False`.

In Python, we perform branching using a *conditional statement* or *if-statement*. The syntax is the keyword *if*, followed by a Boolean expression (the condition), followed by a colon, like this:

`if` *condition*`:`

The if-statement is then followed by a block (a series of indented lines of code — just like a function body). The block of code following the if-statement is only executed if the condition evaluates to `True`. Let's look at an example:

```
radius = 10

def setup():
    size(300, 300)

def keyPressed():
    global radius
    if key == "+":
        radius = radius + 5

def draw():
    global radius
    ellipse(150, 150, radius, radius)
```

Listing 11.1: A program that uses a conditional statement.

This program draws a circle in the middle of the canvas. By default, this circle is quite small; we use global variable `radius` to represent the circle's radius, and initially this is set to 10. The conditional statement is in the `keyPressed()` function. Recall that this is an event handler that is automatically called whenever *any* key on the keyboard is pressed.

You may also recall the special variable `key`: Processing set this variable up for us to hold the value of the most recently pressed key. In this case, we use an if-statement to check if the + key was pressed. If it was, we increase the value of `radius` by 5. If any other key was pressed, then the value of the expression `key == "+"` is `False`, and so we will skip changing the value of `radius` and do nothing. The end result of this is that the circle on the screen gets bigger every time the + key is pressed.

So what if we wanted to make our program a little more user-friendly by adding a message that tells the user what to do if they're hitting the wrong key? It might be natural to try this:

```
radius = 10

def setup():
    size(300, 300)

def keyPressed():
    global radius
    if key == "+":
        radius = radius + 5

    print("Hit + to increase size")

def draw():
    global radius
    ellipse(150, 150, radius, radius)
```

Listing 11.2: A program that uses a simple conditional statement.

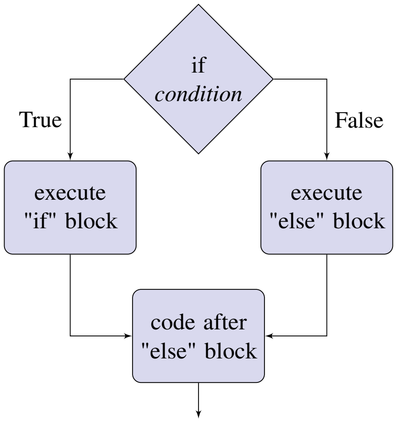
However, this doesn't quite do what we want. The call to `print()` executes regardless of the value of the Boolean expression in the if-statement since the call is outside of the if-statement's code block. In other words, even if the user is doing the right thing, our program will still present them with a message telling them what to do!

What we need is a way of specifying a second block that gets executed only if the Boolean expression in the if-statement evaluates to `False`. We can do this using an *else-statement*. An else-statement is the word `else` followed by a colon. It is written after the if-statement and its block of code:

```
radius = 10

def setup():
    size(300, 300)

def keyPressed():
    global radius
    if key == "+":
        radius = radius + 5
    else:
        print("Hit + to increase size")

def draw():
    global radius
    ellipse(150, 150, radius, radius)
```

Listing 11.3: A program that uses an if-else statement.

Now, if the user hits the + key, the program will increase the circle's radius. Otherwise, it will execute the `else` statement's block of code, which prints instructions to the console. In general, the flow of execution for conditional statements looks like this:



```
if condition:
     # if block (indented)
else:
     # else block (indented)

# code after else block
```

Suppose we want to add functionality to the code which decreases the size of the circle whenever the user presses the - key. Here's one way we could do that:

```
radius = 10

def setup():
     size(300, 300)

def keyPressed():
     global radius
     if key == "+":
          radius = radius + 5

     if key == "-":
          radius = radius - 5


def draw():
     global radius
     ellipse(150, 150, radius, radius)
```

Listing 11.4: A program that uses an if-else statement.

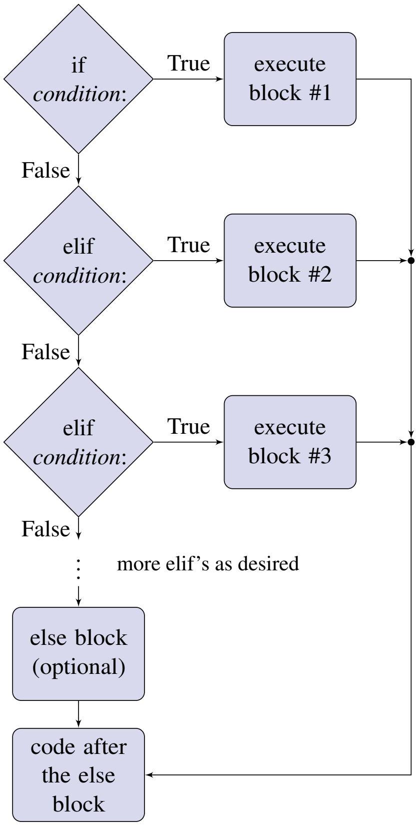
Python, like most other programming languages, gives us a cleaner way to handle multiple conditions intended to be mutually exclusive (i.e. when we want exactly one branch of many to be executed). In Python, there is an *elif-statement* ("elif" is short for "else if"). An elif-statement consists of the word "elif", followed by a Boolean expression, followed by a colon, followed by a block of statements to execute if the Boolean expression is `True`. An elif-statement can appear after the block associated with an if-statement or another elif-statement, but is only executed if the preceding if-statement or elif-statement was found to be `False`.

Here's a different way we can write our program, where we combine shrinking the circle with the - key and printing out instructions if any key other than + or - is pressed:

```
radius = 10

def setup():
     size(300, 300)

def keyPressed():
     global radius
     if key == "+":
          radius = radius + 5
     elif key == "-":
          radius = radius - 5
     else:
          print("Hit + to increase size")
          print("Hit - to decrease size")

def draw():
     global radius
     ellipse(150, 150, radius, radius)
```

Note that only one of the three blocks is executed. As soon as an if-statement or elif- statement is found to be True, its block is executed and no more conditional statements are tested. Thus, no more of the blocks can execute — even if more than one of the conditions are actually true! So when you're using elif, be aware that the order of the conditions can potentially impact your code in a different way than you envisioned. The final else block only executes if **none** of the preceding conditions are True.

Multiple elif-statements and accompanying blocks are allowed as long as the first conditional statement is an if-statement. In all cases, the else statement is optional. Once **any** one of the conditional statements' blocks executes, the program continues at the first line of code following the else block. The flow of execution in an if-elif-else chain is described by the following flowchart and code template:



```
if  condition:
     # block 1 (indented)
elif condition:
     # block 2 (indented)
elif condition:
     # block 3 (indented)

# ... more elif's as desired

else:   # (optional)
     # else block

# code after the else block
```

Once again, notice that only one of the blocks in the if-elif-elif-...-else chain can execute no matter how many elif-statements there are.

12 — Repetition

Learning Objectives

After studying this chapter, a student should be able to:

- identify and correctly author Python language syntax for repetition: while loops and for loops
- trace by hand the flow of program execution for programs that use while-loops and for-loops
- design and author Python programs that use one or more loops
- describe what is an infinite loop

Very frequently in computer programming we would like to repeat certain actions. Sometimes we want to repeat these actions a specific number of times: for example, drawing 1000 circles on the screen. Other times, we want to repeat some actions as long as some specified condition (i.e. Boolean expression) is `True`: for example, keep asking for a password until it is entered correctly. Sometimes we'd like to repeat some action(s) for every element of data in some collection of data elements: for example, for every student's grade in CMPT 140, give a 5% bonus[1]. In Python, we can do all of these things using *loops*.

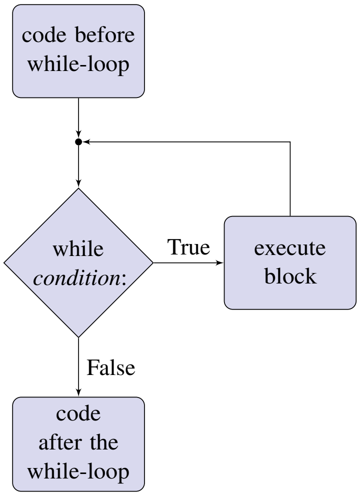
12.1 Repetition in Processing

With Processing, we've actually been using repetition in many of our programs. The `draw()` function itself is an example of repetition: it gets called again and again so long as our interactive program is running. But we, as programmers, don't really have a lot of control over how or when that repetition happens (although we did learn how to change the frame-rate), because Processing does it automatically for us. In this chapter, we'll learn how to use more general purpose repetition in our programs.

[1] You wish!

12.2 While-Loops

A *while-loop* is a type of loop whereby a block of code executes repeatedly so long as a given condition is `True`. While-loops work a lot like an if-statement in that they have very similar syntax — a condition followed by a block—but the block can be executed multiple times instead of just once. While-loops consist of the keyword `while`, followed by a Boolean expression (the *loop condition*), followed by a colon, followed by a block of code. Below you can see the general form of a while-loop, and the corresponding flow of execution presented as a flowchart.



```
# code before while-loop

while condition:
    # block (indented)

# code after the while-loop
```

When execution of code reaches a while-loop, the loop's condition is evaluated. The condition must be a Boolean expression yielding a result of `True` or `False`. If the condition is `True`, the block of code following the while-loop's condition is repeated until the condition becomes `False`. Then the (unindented) code after the while-loop executes. Note that it is possible that the loop condition is `False` the first time it is encountered. If this is the case, then the block is never executed, and execution proceeds to the code after the while-loop.

The following example uses a while-loop to draw a trail of small circles up to the edge of the canvas.

```
# the variable width gives us the current width of the canvas
x_position = width/2

while x_position < width:
    ellipse(x_position, 50, 10, 10)
    x_position = x_position + 10
```

In the code above, the variable `x_position` is used to keep track of the x-coordinate of the circle we are currently drawing. Inside the while-loop, we draw a circle whose centre is at the current `x_position`, and then increase `x_position` by 10, so that the next circle we draw will be slightly further to the right. We keep doing this so long as the `x_position` is still on the screen.

12.2.1 While Loops for Counting

While loops can also be used to execute a block of code a pre-determined number of times. These are called *counting loops* because an integer variable is used to count the number of times the block has executed, and the loop condition is such that the condition is `True` as long as the loop has executed fewer than the required number of times. The following example uses a while-loop to draw 10 progressively smaller circles inside one another:

```
def setup():
    size(200, 200)

def draw():
    size = 100
    count = 0
    while (count < 10):
        ellipse(100, 100, size, size)
        size = size - 10
        count = count + 1
```

In this example, we use two variables that are repeatedly updated in the body of the loop. The `size` variable keeps track of the size of the circle diameter we are drawing. The first time the loop is executed we draw a circle of diameter 100, but the diameter of each circle will get smaller and smaller with each iteration. The `count` variable is simply used to keep track of how many circles have been drawn; so long as `count` is less than 10, we will keep drawing more circles. Every time we draw a circle, we increase `count` by one, thus the `count = count + 1` statement on the last line of the loop's body.

Don't forget to update counter variables within loops; it is a common error that often leads to infinite loops (we will come back to those shortly).

12.3 For-Loops

A *for-loop* is another kind of loop in Python that conveniently allows us to do something to each item in a sequence. You might recall from Chapter 7 that strings are a compound data type comprised of a sequence of characters. A for-loop makes it very easy for us to access those characters, one at a time, and do something with each one before moving on to the next character.

For example, say we wanted to display all of the characters in a string in a diagonal line on the canvas. The following program would do just that:

```
word = "COOKIE"
x = 10
y = 10
for letter in word:
    text(letter, x, y)
    x = x + 10
    y = y + 10
```

The block following the for-loop is executed once for each character in the string `word`; each time the block is repeated, the variable `letter` refers to the next character in the string.

In general, the syntax of a for-loop consists of the word `for`, followed by a variable name, followed by the keyword `in`, followed by a sequence, followed by a colon, followed by a block:

```
for variable in sequence:
    # Block of code -- each time this block is repeated,
    # variable refers to the next item in the sequence.
    # Repetition stops after each item in the sequence has
    # been processed.
```

When we do something for each element of a sequence, we say that we are *iterating* over the sequence.

12.3.1 Sequences

You may have noticed that in our discussion on for-loops, the word *sequence* has shown up quite a bit. In Python, a *sequence* is a compound data-type that consists of several pieces of data in a particular order. In a way, you can think of a sequence as a sort of “super data-type” that describes several other compound data-types. Strings are an example of a sequence, because in a string, the order of characters is important. “cat” and “tac” are not the same string, even though they contain the same letters. So all strings are sequences. But not all sequences are strings, as we shall presently.

12.3.2 Ranges and Counting For-Loops

We can use for-loops to create counting loops just like we did with while-loops. To do so, we first need to learn about a new kind of sequence called a *range*.

A *range* is a sequence of integers that begins at an integer a (the *start*), ends **before** an integer b (the *stop*), and in which the difference between each element in the sequence, called the *step size*, is equal. For example, the range starting at 1 that stops at 5 and has a step size of 1 is comprised of the sequence of integers 1, 2, 3, 4. Ranges are created with Python’s built-in `range()` function. The `range` function requires two arguments, *start* and *stop*, and can optionally accept a third argument for the step size which, if not given, defaults to 1. You may also provide just a single argument to range; `range(x)` is equivalent to `range(0, x, 1)`, and is the sequence $0, 1, 2, \ldots, x-1$. Here are some example ranges:

```
range(0,5,1)      # the sequence 0, 1, 2, 3, 4
range(5)          # the sequence 0, 1, 2, 3, 4
range(-4, 4)      # the sequence -4, -3, -2, -1, 0, 1, 2, 3
range(0, 11, 2)   # the sequence 0, 2, 4, 6, 8, 10
range(2, -3, -1)  # the sequence 2, 1, 0, -1, -2
range(0, 5, 10)   # the sequence 0

# General form:
range(start, stop, step_size)
```

Remember: the value *stop* is **not** part of the sequence.

Ranges can be used to write counting for-loops. Here is a for-loop that repeats its block exactly `N` times:

```
for i in range(N):
    # do something
```

In this loop, i refers to the value 0 on the first repetition, 1 on the second repetition, and so on, up to N-1 on the last repetition. It is equivalent to the following while-loop:

```
i = 0
while i < N;
    # do something
    i = i + 1
```

12.4 Choosing the Right Kind of Loop

Generally, for-loops are what you want to use to iterate over a sequence. Both for-loops and while-loops are appropriate for simple counting loops. You may prefer using for-loops with ranges for counting purposes because it requires less typing than the equivalent while-loop. For most other non-counting loops that have complicated loop conditions and/or don't involve iterating over sequences, while-loops are likely the best choice.

12.5 Infinite Loops

Infinite loops are loops that repeat forever, and they can sometimes be the worst enemy of the novice programmer. That's because infinite loops can make it **look** like your program is doing nothing, when in fact the real problem is that your program is effectively running uselessly in circles! A while-loop whose loop condition can never become False is an infinite loop. Here is an example:

```
x = 10
while (x < 100):
    ellipse(x, x, 10, 10)
```

The intent for this program is to draw a diagonal line of circles across the canvas. But in the body of the loop, we forgot to include any kind of statement that changes x. Since x is initialized with a value of 10 that doesn't change, the loop condition x < 100 will always be True. If you try running this program, it won't actually draw anything on the screen because the program gets stuck in an infinite loop. The only thing we can do is click on Processing's "stop" button, which will stop the program from running. If you ever find yourself mystified by the behaviour of a program that you write, where the program seems to simply refuse to do what you tell it to, be weary that an infinite loop may be the culprit.

13 — Nesting Programming Constructs

Learning Objectives

After studying this chapter, a student should be able to:

- trace behaviour for programs using nested if-statements
- trace behaviour for programs using nested loops
- trace behaviour for programs using nested loops and if-statements combined
- author Python code using nested if-statements, loops, and combinations thereof

So far, we've learned about two types of control statements: conditionals (involving the keywords `if`, `elif` and `else`) and loops (involving the keywords `while` and `for`). Both of these types of control statements have the same basic structure: the control statement appears first, followed by an indented block of code. The block of code below the control statement consists of one or more valid Python statements. There are no restrictions on the sorts of statements that can go in the block, so naturally, the block can include additional control statements. We can have if-statements inside loops and loops inside if-statements in any combination you can imagine. This concept of including control statements within control statements is known as *nesting*.

In this chapter, we're not going to present any new information. We'll just look at several examples of programs that combine conditionals and loops in different ways.

13.1 Nesting If-Statements

Here's a simple example of using an if-statement inside another if-statement:

```
# assume first_name and last_name are variables
# that have already been given values

if last_name == "Monster":
    print("Welcome home, Mr. Monster.")
    if first_name == "Cookie":
        print("Would you like a cookie?")
```

Notice that the second if-statement is part of the block associated with the first if-statement (because it is indented). The line `print("Would you like a cookie?")` is indented **again**, because it is associated with the second if-statement. As a result, if `last_name` has the value `"Monster"` and `first_name` has the value `"Cookie"`, then both `print()` statements will be executed when the program is run. If `last_name` has the value `"Monster"` and `first_name` has some value **other** than `"Cookie"`, then the first `print()` statement (`"Welcome home, Mr. Monster."`) will be executed, but the second will not.

So what happens if `last_name` isn't set to `"Monster"` but `first_name` **is** set to `"Cookie"`? In such a case, neither `print()` statement will be executed. Recall the behaviour of an if-statement: if the if-statement's condition is `False`, then the **entire block** associated with the if-statement is skipped over. Since the second if-statement is part of the block that is being skipped, the fact that its associated condition happens to be `True` is irrelevant; the computer won't even check.

When we combine nested if-statements with `else` (or `elif`) statements, paying attention to indentation becomes very important. Consider the following two programs:

```
if last_name == "Monster":
    print("Welcome home, Mr. Monster.")
    if first_name == "Cookie":
        print("Would you like a cookie?")
else:
    print("There's a burglar!  Protect the cookies!")
```

Listing 13.1: The else-statement is associated with the first if-statement

```
if last_name == "Monster":
    print("Welcome home, Mr. Monster.")
    if first_name == "Cookie":
        print("Would you like a cookie?")
    else:
        print("There's a burglar!  Protect the cookies!")
```

Listing 13.2: The else-statement is associated with the second if-statement

The programs are identical except for the indentation of the `else` statement and its associated block. For listing 13.1, the program will declare there is a burglar whenever `last_name` is not equal to `"Monster"`, regardless of the value of `first_name`. That's because the `else` is at the same level of indentation as the **first** if-statement, indicating that the `else` is associated with the first if-statement. In listing 13.2, the `else` is associated (via indentation) with the **second** if-statement. Thus, the program outlined in 13.2 will only report a burglar when `last_name` is equal to `"Monster"`, but first_name is **not** equal to "Cookie". If last_name is not equal to "Monster", then the program won't print anything to the screen.

13.2 Nesting Loops

Loops that contain other loops are a common and useful tool in computer science. The key thing to keep in mind here is that any loop(s) on the inside of another loop will execute completely for every single iteration of the 'outer' loop!

Let's consider an example. Suppose we want to print out all pairs of 26 letters in the English alphabet, i.e. (aa, ab, ac...az, ba, bb, bc...zy, zz). If we manually write out every pairing's print() statement, we would need $26^2 = 676$ print() statements. We don't want to do that! Using nested loops, we can get the job done in just a few lines of code:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
for letter1 in alphabet:
    for letter2 in alphabet:
        pair = letter1 + letter2
        print(pair)
```

We know from the length of the alphabet string that the first (outer) for-loop will execute exactly 26 times. During the first iteration of the loop, letter1 will be set to a. Then we enter the second (inner) for-loop and iterate through it. The variable letter2 will also cycle through all 26 letters of alphabet. For each letter2, we concatenate it together with letter1 and print the resulting string to the console. This all happens during the **first** iteration of the outer loop, so letter1 doesn't change! Once the inner loop finishes its 26 iterations, we hit the end of the outer loop for the first time, at which point letter1 becomes b for the next iteration. The entire inner loop will then repeat again, with the only difference being that letter1 is now b instead of a. The inner loop will continue to repeat its iterations until the outer loop is done cycling through its iteration of alphabet's characters.

13.3 Nesting If-Statements and Loops

We can combine any number of if-statements and loops in any order that we like. Consider the following program, which prints out appropriately timed messages for a rocket lift-off:

```
for countdown in range(10, 0, -1):
    print(countdown)
    if countdown == 8:
        print("Heating up!")
    elif countdown == 4:
        print("Ignition!")
    elif countdown == 1:
        print("BLAST OFF!")
```

For each iteration of the for-loop, the countdown variable refers to a different value (starts at 10 and counts down to 1). No matter what its value, countdown is printed to the console with each iteration. Then, depending on the value that countdown has for a particular iteration, an additional message may be printed as well.

In this next example, we have two different loops under two different branches of a conditional statement:

```
# Assume count_up has already been given a Boolean value
if count_up:
    print("Counting up!")
    for i in range(1, 11):
        print(i)
else:
    print("Counting down!")
    for i in range (10, 0, -1):
        print(i)
```

Here, because we used an if/else structure, only one of the two loops will ever be executed. If the `count_up` variable is `True`, the program will print `Counting up!` and display the numbers from 1 to 10 in ascending order. If instead `count_up` is `False`, the upward-counting block won't even be examined and instead the program will print `Counting down!` and display the numbers from 10 to 1 in descending order.

13.4 Multiple Layers of Nesting

There's no limit as to how deeply we can nest our programming constructs. For example, we can extend our alphabet-printing program from earlier to print as many letter combinations as we like just by adding extra loops. This version prints out all possible quadruplets of letters from the alphabet:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
for l1 in alphabet:
    for l2 in alphabet:
        for l3 in alphabet:
            for l4 in alphabet:
                quadruplet = l1 + l2 + l3 + l4
                print(quadruplet)
```

Sometimes, multiple layers of nesting can be replaced by using a single Boolean expression. The following two programs are an example of this. They have the exact same behaviour, but program 13.3 uses multiple nested if-statements, while program 13.4 uses a single if-statement with the conditions from 13.3's nested-if statements expressed as a single Boolean expression.

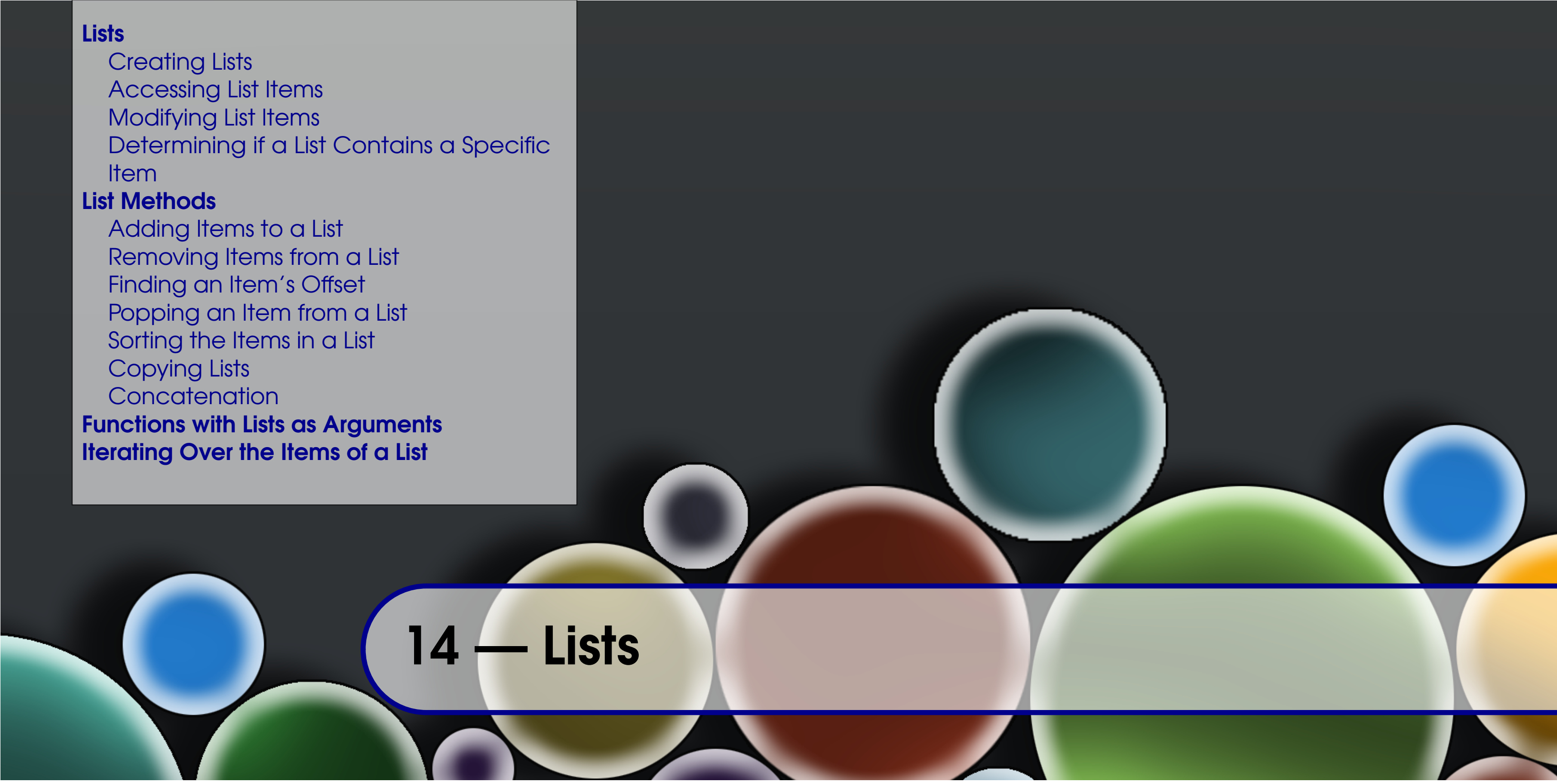
```
if first == "Alistair":
    if middle == "Cookie":
        if last == "Monster":
            print("Welcome, Master Cookie!")
```

Listing 13.3: Program using multiple nested if-statements

```
if first=="Alistair"and middle=="Cookie" and last=="Monster":
    print("Welcome, Master Cookie!")
```

Listing 13.4: Program using single if-statement with longer Boolean expression

In this case, neither of these programs are necessarily more “correct” than the other; it’s a matter of which one is easier for a human reader to understand. The point, for now, is simply to show that in a programming language, just as in a natural language, there are often many ways to express the same idea.



14 — Lists

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a list is
- create lists in Python
- access and manipulate the items in a list in Python
- employ simple slicing on lists
- write Python programs that use lists to store data

14.1 Lists

A *list* is a compound data type consisting of a set of data items arranged in a specific linear ordering. We first mentioned lists, along with the notion of a compound data type, back in Chapter 7. Conceptually, lists in Python share many properties with the informal, plain language notion of a list that you're familiar with from everyday life. For example, suppose we had a grocery list, and on that list were the following items:

```
flour
butter
brown sugar
white sugar
eggs
vanilla
baking soda
chocolate chips
```

There are a number of things we could say about such a list. For example, we could say that “there are eight items on the list.” We could also refer to the items on the list with regard to their ordering. For example, we could say “the first item on the list is flour,” and “the third item on the list is brown sugar.”

Lists in Python are formalized, but all of the same concepts apply. In this chapter, we’ll learn how to use lists in our programs.

14.1.1 Creating Lists

In order to use lists, we first have to learn how to construct one so that we have a list to interact with. One way of creating a list is by writing a list literal. To do that, we’re going to use a new type of bracket that we haven’t used before: square brackets. A pair of matching square brackets tells Python that you are creating a list. The contents of the list go in-between the brackets, separated by commas. Of course, lists are themselves data, so typically when you create a list literal, you’ll associate it with a variable name so that you can refer to the list by that name. Here are several examples:

```
# a list of some prime numbers
x = [2, 3, 5, 7, 11]

# a list of video game titles
y = [’Super Mario Bros’, ’Civilization’, ’Bionic Commando’]

# a list containing different types of data
z = [’Ultimate Answer’, 42.0, 6*9]
```

The multiplication of a list and an integer works the same way as multiplication of integers and strings, so we can create a list of *n* copies of a value by first creating a list that contains only that one value, and then multiplying it by *n*:

```
n = 10;
# create a list of n zeros:
zeros = [0] * n

# create a list of n empty strings:
empty_strings = [’’] * n

print(zeros)           # this prints [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(empty_strings)   # guess what this will print, then try it.
```

Creating a list of *n* copies of a value is useful when you want to create a list of initial values and then modify those initial values later in the course of a task.

Finally, another common thing you might want to do is create a completely empty list. This is similar to grabbing a blank sheet of paper on which you plan to write a grocery list. The syntax for writing an empty list is just two matching square brackets without any items in-between, like this:

```
# create an empty list
grocery_list = [ ]
```

14.1.2 Accessing List Items

The whole purpose of a list is to store multiple data items. But in order to examine or modify those items, we still need to be able to access them individually. One way to do this is by using the item's *offset*. An item's offset is simply its position in the list. The first item in a list has an offset of 0 — not 1, but 0. Computer scientists start their counting from 0. It can take a little bit of getting used to. In any case, that means that the second item in the list has an offset of 1, the third item has an offset of 2, and so on. If there are n items in the list, then the final element has offset $n-1$ (because we started counting at 0!). To access a particular item in a list, we use the name of the list followed by a pair of square brackets. Inside those brackets, we put the offset of the item we want. Here's an example:

```python
alphabet = ["A", "B", "C", "D", "E"]
print(alphabet[2])
```

The code above would print out the element of `alphabet` at offset 2, which is the value `C`.

If you attempt to access an item at an offset that does not exist, Python will issue an error[1]. Also, **offsets must be integers**. You cannot use a float as a list offset, even if the float happens to be a whole number. Here are some examples of erroneous code:

```python
alphabet = ["A", "B", "C", "D", "E"]

print(alphabet[5])
# can’t do this!  The last valid offset is 4

print(alphabet[1.0])
# can’t do this!  The offset must be an integer

print(alphabet[int(1.0)])
# this is ok.  The int() function will convert 1.0 to 1
```

Slicing: Accessing Parts of a List

What if you have a list and want to make a *new* list that contains only some of the data from the original list? Python gives us a very easy way to do that. Once again, we'll use square brackets, but instead of having a single offset inside the brackets, we'll have *two* offsets (separated by a colon). This tells Python "make me a new list with all the data from in-between those two offsets." This process is called *slicing*. It looks like this:

```python
alphabet = ["A", "B", "C", "D", "E"]
abc = alphabet[0:3]
```

After the two statements above, the list `abc` will contain data values `["A", "B", "C"]`. The only tricky thing to note here is that although we specified the offset range `[0:3]`, the data item at offset 3 from the original list (`"D"`) didn't get included in the new list. In other words, the range of offsets that you specify for slicing is **inclusive** on the lower end and **exclusive** on the upper end. This manner of describing a range of values (including the start but excluding the end) is standard throughout the

[1] This is very handy! There are languages that will NOT issue an error when you do this, which can wreck your program in unpredictable ways

Python language, as we already saw with the range() function that we used for for-loops in chapter 12.

14.1.3 Modifying List Items

We can also modify the items in a list individually. This means that we can change the value at a particular list offset without affecting any of the other items in the list or creating a brand new list. Doing so uses the same basic syntax as variable assignment:

```
alphabet = ["A", "B", "C", "D", "E"]
alphabet[2] = "X"
```

After executing the code above, the content of alphabet contains ["A", "B", "X", "D", "E"]. Only the third element was changed. What happened to the "C" that used to be in third position of the list? It's just gone. Coming back to our analogy of a physical grocery list, it's like we crossed out the third item on the list and wrote something else in its place.

14.1.4 Determining if a List Contains a Specific Item

We can check whether a specific data value exists anywhere in a list using the Python keyword in. We've already seen this word before in Chapter 12 when talking about for-loops. Here, we will use the keyword in a slightly different way.

In the context of lists, we use in as an operator that produces a Boolean value. The in operator requires that its left operand is an expression, and its right operand is a list. It evaluates to True only if the value of the left operand is an item in the given list:

```
classlist = ["Bert", "Ernie", "Cookie", "Big Bird"]

if "Ernie" in classlist:
    print("Ernie is here!")
```

So long as the string value "Ernie" is anywhere in the list classlist, then the expression "Ernie" in classlist evaluates to True.

We can also use not in to check whether an item is NOT in a list. It uses the exact same syntax as with in (only with a preceding not this time) and does pretty much what you might expect it to:

```
classlist = ["Bert", "Ernie", "Cookie", "Big Bird"]

if "Cookie" not in classlist:
    print("Cookie's skipping class!  Call his parents!!!")
```

In the example above, the print() statement will not in fact be executed, since in this case "Cookie" not in classlist evaluates to False (as "Cookie" is, indeed, in the list).

14.2 List Methods

In Chapter 10, we introduced the notion of objects, which are pieces of data have special functions called methods. As a reminder, here is the basic syntax for a method call.

```
# general form for calling object methods
# assume my_object refers to an object of some kind
my_object.function_name(arguments...)
```

Lists have methods as well, and the syntax for calling them is the same as in the example above. Methods called like this will often modify the associated list in some way, as we'll see with the methods we look at next.

14.2.1 Adding Items to a List

We already know how to create lists that contain data items and how to modify those existing items, but what if we want to add something new to a list **without** replacing anything that's already there? The list method append() does just that. It takes a single argument, which is the data item we want to add to the list, and adds it **to the back** of the list. Here's an example:

```
lucky_numbers = [1, 3, 33]
lucky_numbers.append(100)
```

After executing the statements above, the list lucky_numbers will consist of [1, 3, 33, 100].

With append(), we can add any kind of data to a list. It doesn't have to match the type of the data that's already in the list. The following is perfectly valid:

```
count = [3, 2, 1]
count.append("BLAST OFF!")
```

14.2.2 Removing Items from a List

The remove() list method deletes a specific item by value from the list, no matter what index it occupies:

```
rebels = ["Han", "Chewie", "Luke", "Leia", "C3PO"]
rebels.remove("Luke")  # delete "Luke" from the list
```

After running the code above, the rebels list will consist of the items ["Han", "Chewie", "Leia", "C3PO"]. The item "Luke" is no longer in the list and the entire list is shorter by one data entry.

Be careful: If there are multiple occurrences of the specified item in the list, only the first one (i.e. the one at the smallest offset location) will be removed. All the other occurrences remain! Also, if you try to remove an item that doesn't exist, Python will report an error.

14.2.3 Finding an Item's Offset

You can retrieve the offset of an item in a list using the list's index() method. This function **returns** the offset of a given item as an integer, and like all functions that return values, we'll probably want to assign the returned value to a variable so we can do something useful with it later. Here's an example where we find the offset of an item and then change the item at that index:

```
names = ["Bert", "Ernie", "Kookie", "Big Bird"]

i = names.index("Kookie")
names[i] = "Cookie"
```

Just like with remove(), if the item exists more than once, index() will only report the offset of the first occurrence of the item. And if the item does not exist, Python will report an error.

14.2.4 Popping an Item from a List

The pop method **returns** the item at a specified list **index** and then **deletes** that item, all in one go! Here's an example:

```
names = ["Bert", "Ernie", "Cookie", "Big Bird"]

student = names.pop(0)
message = student + " dropped the class!"
print(message)
print(names)
```

The program above will print out `Bert dropped the class`, followed by the list, which will then consist of just `[‘‘Ernie’’, ‘‘Cookie’’, ‘‘Big Bird’’]`.

14.2.5 Sorting the Items in a List

If we want to re-order the existing items in a list, we can use the `sort()` method. This function will re-arrange the list items so that they are in increasing (i.e. smallest to largest) order. This only works if the items in a list are all comparable with one another.

Numbers are sorted in numeric order. Note that we can mix integers and floats; although they are not precisely the same data type, they can still be compared using > and <, so `sort()` will work with them.

```
numbers =  [42.0, 7,  2.6, -17, -42]
numbers.sort()
print(numbers)
```

The program above will print out the list `[-42, -17, 2.6, 7, 42.0]`.

Strings are sorted in lexicographic order (dictionary order):

```
words = ["what", "is", "dead", "may", "never", "die"]
words.sort()
print(words)
```

This will print out the list `["dead", "die", "is", "may", "never", "what"]`.

14.2.6 Copying Lists

Recall that the assignment operator, =, associates a variable name (also called an *identifier*) with a piece of data. Suppose we did this:

```
x = 42
y = x
```

After executing the statements above, x and y are both associated with the value 42. But if we later associate x with a different value, it doesn't change the fact that y is still associated with the value 42.

But things are a bit different with lists. Firstly, if we do this:

```
x = [2, 4, 6, 8, 10]
y = x
```

This is no different from the previous example – we have simply assigned two variable names to refer to the same list. No problems so far. How about this:

```
x = [2, 4, 6, 8, 10]
y = x
x = [’A’, ’B’, ’C’]
print (y)
```

Which of the two lists will be printed by the code above? The answer is the first one: `[2, 4, 6, 8, 10]`. Initially, we made `x` refer to that first list, and then we made `y` refer to the same list. Then we created a completely new list, which we associated with `x`. This didn’t change the fact that `y` was still associated with the original list. So far, this is all the same behaviour as when we were just using atomic data types.

Now how about this:

```
x = [2, 4, 6, 8, 10]
y = x
x[2] = 500
print(y)
```

What will be printed now? The answer is `[2, 4, 500, 8, 10]`. Yes, that’s right. Even though we didn’t appear to change `y`, the list that gets printed has a 500 in the middle. That’s because `x` and `y` are both referring to the **same list**. The statement `y = x` doesn’t create a copy of the list associated with `x` before associating that list with `y`. It just associates `y` with the already-existing list that is associated with `x`.

You might think that Python does things this way just to be mean to poor, novice students. But actually, most of the time, it’s quite sensible behaviour. Recall that there is no limit to the size of a list. A given list might have millions or even billions of data items. Making a copy of so much data is a lot of work for the computer, work that we don’t really want the computer to have to do if all we want is to be able to refer to a list by two different names.

The important thing to remember is that the assignment operator = does not make a copy of data. It only associates a new name with that data. If you really want to make a copy of a list, you can do it manually using a loop, like so:[2]

```
x = [2, 4, 6, 8, 10]
y = [ ]
for i in x:
    y.append(i)
```

14.2.7 Concatenation

We saw, back in Section 8.2.3, that the + operator concatenates two strings. The + operator can actually be used as a concatenation operator with any type of sequence, and it turns out lists are also a sequence:

[2] Even this works only for lists that contain just atomic data! But properly copying more complicated lists is beyond the scope of this class.

```
a = [1, 3, 5, 7, 9]
b = [2, 4, 6, 8, 10]
c = a + b
print(c)
```

The code above will print the list `[1, 3, 5, 7, 9, 2, 4, 6, 8, 10]`. Notice that the items are not sorted or re-arranged in any way. All of the items in the list `b` were just tacked on to the end of the items in `a`. The result is a completely new list consisting of both sets of items, which we then associate with the name `c`.

14.3 Functions with Lists as Arguments

In the previous section, we looked at some list methods: special functions that we call by stating the name of the list, followed by a period, followed by the function name. Python also has a number of built-in functions that accept lists as arguments. These functions typically give back (via a *return* value) some kind of information about a list. Here are some of those functions:

- `max(L)` returns the largest item in list `L`
- `min(L)` returns the smallest item in list `L`
- `sum(L)` returns the sum of the items in list `L`
- `len(L)` returns the number of items in list `L`

In fact, most of these built-in functions work on strings as well (recall that we said strings really are a compound data type, even though we started using them right away with other atomic data types).

14.4 Iterating Over the Items of a List

We often want to perform some kind of computation for every element in a list. This is called *iterating* over the list. Fortunately, again because lists are a sequence, we already know how to do this, with for-loops!

```
classgrades = [50, 90, 30, 70]

# print out a comment based on each grade
for grade in classgrades:
    if grade >= 80:
        print("Excellent work!")
    elif grade >= 50:
        print("You pass!")
    else:
        print("You FAIL!")
```

This form of loop is excellent if we want to use each data item in the list in some kind of action or computation.

We can also iterate over a list by iterating over its indices. This enables us to modify each element of the list, since we can only modify a list item if we know its offset:

```
classgrades = [50, 90, 30, 70]

# give everyone a five point grade bonus
for i in range(len(classgrades)):
    classgrades[i] = classgrades[i] + 5
```

After executing the above code, the list `classgrades` will be changed to `[55, 95, 35, 75]`.

15 — File I/O

Learning Objectives

After studying this chapter, a student should be able to:

- distinguish text files from binary files
- describe some common ways in which data may be organized in a text file
- author Python code to open and close files
- author Python code to read a text file one line at a time
- apply basic string processing to read numeric data from a text file containing numbers
- author Python code to write data to a text file

Up to this point, the only mechanisms we have used for data **input** into our programs are to either code the data right into our program as literal data (this is sometimes called *hard-coding* the data) or acquire it from the user through user actions (keyboard or mouse). In this chapter, we look at how to obtain input data stored in files. Similarly, the only way we have seen our programs give system feedback is by printing to the console or drawing on the canvas. In this chapter, we will also look at how to write data to a file. We refer to these capabilities as *file I/O* — the *I/O* is short for **Input/Output**, indicating that we're using files to get data in and out of programs.

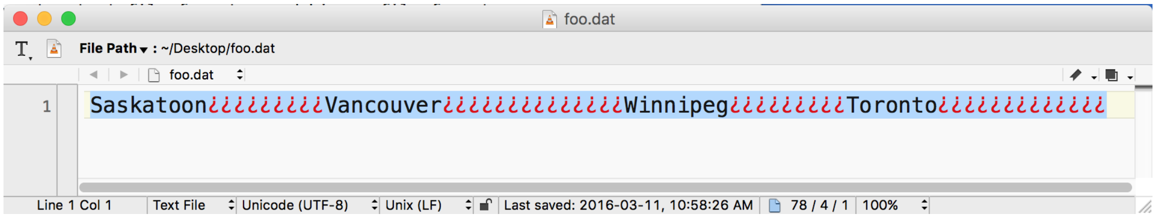
15.1 Data File Formats

The term *file format* refers to the way in which data is organized in a file. There are two main types of file formats: text file formats and binary file formats.

Text file formats are readable by humans. You can open them in any text editor and see the data inside and how it is organized. In text files, numbers are stored as strings of digits. A text file containing data about cities and their average annual temperatures might look like this:

```
Saskatoon   9
Vancouver   14
Winnipeg    9
Toronto     13
```

Binary file formats are generally not readable by humans because the data is binary-encoded. Such files generally do not contain any meaningful whitespace such as spaces or newlines and appear as gibberish when viewed in a text editor. In a binary file format, numbers are stored in binary (base 2) format, in groups of 8-bits (a byte). A number might be comprised of the bits in one, two, or four consecutive bytes. If we stored the temperature data, above, in a binary file, it might look something like this when we load it into a text editor:



```
Saskatoon¿¿¿¿¿¿¿¿¿Vancouver¿¿¿¿¿¿¿¿¿¿¿¿¿¿Winnipeg¿¿¿¿¿¿¿¿¿Toronto¿¿¿¿¿¿¿¿¿¿¿¿¿
```

Binary files are typically more compact and use less disk space and are used frequently in commercial applications and games. Since CMPT 140 is an introductory course, we will not be using any binary file formats, only text file formats.

15.1.1 Common Text File Formats

In this section, we review two typical ways in which we might organize data in a text file.

List Files: One Data item Per Line

A *list file* has one data item per line, and usually each line contains the same type of data. List files are very simple to read into a program since each line of the file contains one data item, and most programming languages have built-in functions for reading one line from a file. An example of a list file might be observations of temperature recorded over a single day:

```
-2.7
-1.8
0.3
2.4
3.5
5.9
```

Tabular Files: One Group of Related Data Items Per Line

A *tabular file* format has a fixed number of data items per line. We can think of such a file as a table, because it will have a certain number of rows (lines) and a certain number of columns (data items per line). Almost all spreadsheet programs can save spreadsheet data in this sort of format (usually the file type is a .csv file).

The data items on a line may be different types, but often, each column of data is all of the same type, that is, the n-th piece of data on each line is of the same type. Data items on a line might be separated by spaces, or another character, such as a comma. Whatever character is used to indicate separation of data items on a line of the file is called the file's *delimiter*. It delimits, or separates one data item from the next. Here is an example of a tabular text file, delimited by commas, where each line holds data about a student:

```
Big Bird,Arts,1112222,76
Cookie Monster,Arts,3334444,91
Snuffleuppagus,Engineering,5556666,49
```

Each line of this file holds the data for one student from a grading spreadsheet, and contains exactly 4 data items, separated by commas.

If our data items themselves do not contain spaces, we can use whitespace as a delimiter, which makes the text file look more like a table. Here is an example of a tabular datafile that stores weather observations taken every four hours for different weather stations on one specific day of the year where each weather station is identified by a four-digit ID number:

```
1783 22   25   27   28   21   19
2214 -4   2    6    7    6    0
9934 -40  -32  -26  -21  -24  -32
5538 15   17   21   22   23   19
```

The first column contains the weather station ID number, and the remaining columns store temperature observations. Since observations are every four hours, there are six such columns.

Other Formats for Text Files

Any format you can think of is theoretically possible, but you might have to write custom code that can process unconventional formats.

15.2 Files in Python

In Python, we interact with files on the disk via an abstraction. We can ask Python to return a particular type of data called a `file object` that allows us to interact with a data file on disk. This is called *opening* a file. We can open a file and obtain an object for that file using Python's built-in `open()` function. The `open()` function returns a file object that we can associate with a variable. File objects, like the lists we saw in chapter 14, have methods, which we can then call to get access to the data stored in the file.

Suppose the table of temperature data from the previous section is stored in a file called `temperatures.txt`. Furthermore, assume that you've put this file in your Processing sketch folder. You can access this folder by clicking on **Sketch** in the Processing menu bar and selecting **Show Sketch Folder** . Under these conditions, we can open the file in Python like this:

```
f = open('temperatures.txt', 'r')
```

The first argument to `open()` is a string containing the name of the file to be opened — this can be any valid pathname. The second argument string is the *mode*. Here, we are using the file mode `'r'`, to indicate that we want to **read** from the file. Later, we'll see how to write to files using the `'w'` mode.

Now that `f` is associated with your file, we can do some nifty things with it to access the data in that file. Before we move on, we must note that once a file is opened, it must be *closed* again when you are done with it. We do that like this:

```
f.close()
```

Once you call `f.close()`, `f` can no longer be used to manipulate the file; trying to do so will result in an error message. Note that `close()` is a method of the file object, and thus we call it using the period-notation for calling methods that we first saw in chapter 14. The `open()` funciton, on the other hand, is not a file method, because at the time we need to call `open()`, the file object doesn't exist yet. We can't tell a file that doesn't yet exist to open itself; that would be silly!

If you forget to close a file that was opened in **read** mode, usually nothing bad will happen, although you really should always do it. If you forget to close a file that was opened in **write** mode, it is possible that data that you wrote to the file will not actually be written, and that is very bad!

15.3 Reading Text Files

In this section, we're going to learn how to read data from the simplest form of text files that we described above: list files. This will actually be pretty easy: it turns out that we can iterate over the lines of a file object just like we can iterate over the elements of a list. When we do this, each line in the list file is treated as a single string.

15.3.1 Reading List Files

List files are pretty easy to deal with since each line of a file contains a single data item and, as we have already mentioned, we can access each line of a file as a string easily.

Suppose we have a file called `movietitles.txt` which contains one movie title per line. We can read the movie titles in the file and store them in a Python list like this:

```
# Open the file for reading
f = open(’movietitles.txt’, ’r’)

# create an empty list
titles = []

# iterate over each line of the file
for line in f:
     # append the next line (movie title) to the list
     titles.append(line)

# close the file
f.close()
```

If `movietitles.txt` contains the following data:

```
The Fellowship of the Ring
The Two Towers
The Return of the King
```

Then the above code will result in `titles` referring to the list:

```
[’The Fellowship of the Ring\n’, ’The Two Towers\n’, ’The Return of the King\n’]
```

Hey, wait, that’s weird. What are those `\n`’s at the end of each string in the list? Those are *newline* characters; they are invisible characters that mark the end of each line of a text file, and therfore are included in the string that comprises a line of the file. In Python, `\n` represents the newline character. Even though it is represented by two characters, `\` and `n`, it is actually a single character. It is represented this way so that we can see it because normally it is invisible since it is not associated with any symbol on the keyboard.

Usually we don’t want newline characters at the end of our strings. We can remove them by using a string method called `rstrip()`. If `s` refers to a string, then `s.rstrip()` returns a **copy** of `s` that has all of the whitespace at the end of the string, including spaces and newlines, removed. Revising our loop in the previous code to this:

```
f = open(’movietitles.txt’, ’r’)
titles = []
# iterate over each line of the file
for line in f:
    # append the next line (movie title) to the list
    titles.append(line.rstrip())
f.close()
```

results in `titles` referring to the list:

```
[’The Fellowship of the Ring’, ’The Two Towers’, ’The Return of the King’]
```

What if we have a list file of numbers? Remember that if we just iterate over the lines in a file like we did above, every line is returned as *string* data. That’s probably not what we want for numeric data. The solution is to use the built-in functions `int()` or `float()` to convert strings to numbers. For example `int(’42’)` returns the integer 42, and `float(’64.9’)` returns the floating point value 62.9. If you use `int()` or `float()` on a string that doesn’t represent a number of the appropriate type, Python will respond with a `ValueError`. We could read the list file containing temperature data at the beginning of Section 15.1.1 and store the data as a list of floats like this:

```
f = open(’temperatures.txt’, ’r’)
temps = []
for line in f:
    temps.append(float(line))
f.close()
```

The program above would cause `temps` to refer to the list:

```
[-2.7, -1.8, 0.3, 2.4, 3.5, 5.9]
```

15.3.2 Reading Tabular Files

Reading tabular files is almost the same as reading list files. The main difference is that since we initially access each line of the data file as a single string, we need an extra step to separate the data items on each line. Remember that in a tabular file, the data items on each line are separated by a delimiter. Strings have a `split()` method which returns a list of individual strings that occur between a specific delimiter character. For example, the string `’The king in the north.’` can be separated into individual words like this:

```
my_string = ’The king in the north.’
words = my_string.split()
```

This results in `words` referring to the list:

```
[’The’, ’king’, ’in’, ’the’, ’north.’]
```

If we want to split a string based on a delimiter other than whitespace, we just pass the desired delimiter to `split` as an argument. Here’s how we can obtain a list of strings from a string delimited by commas:

```
my_string = ’42,38,27,99,55’
numbers = my_string.split(’,’)
```

This results in `numbers` referring to the list

```
[’42’, ’38’, ’27’, ’99’, ’55’]
```

They’re still strings, but we could later use a for-loop to convert the contents of the list to integers using the `int()` function.

We can obtain the lines of a tabular data file in the same way that we obtained lines for list files, but then we have to use `split` to divide up each line into its individual data items. We might then need to convert these individual data items into integers or floats. Let’s walk through an example of how to do this.

Suppose we have a data file of student grades called **students.txt** that looks like this:

```
Ernie     40   55   48   60   72
Bert      99   99   99   99   99
Elmo      65   72   0    75   80
```

The first column of the file is the student’s name, the next five columns are their grades (out of 100) for five different assignments. Suppose we wanted to read in this file, calculate an average grade for each student and print out that grade. Here’s what the program to do that looks like:

```
data = open(’students.txt’, ’r’)
for line in data:
     grades = line.split()
     name = grades.pop(0)
     avg = 0.0
     for g in grades:
          avg = avg + float(g)
     avg = avg / len(grades)
     print(name)
     print(‘: ‘)
     print(avg)
data.close()
```

The program starts by opening the file and uses a for-loop to examine each line in the file, which is so far just like we did with list files. Then, we use the `split()` function to put all the items from the line into a list. We know that the very first item on each line is the student's name, so we `pop()` that off of the list (recall the `pop()` function from Chapter 14) and associate it with the variable `name`. Then, we use a for-loop to access each of the remaining items in the list, using the `float()` function to convert them to floating point numbers as we add them to a variable called `avg`. We need to do this, since `split()` just breaks up a string into a list of strings. Finally, once we're done, we divide `avg` by the number of items in the list of grades, and then print the `name` and the `avg` to the console.

15.4 Writing Text Files

So far, whenever we've wanted to display textual output to a human user, we've displayed it via the console (using `print()`) or drawn it onto the Processing canvas (using `text()`). It turns out we can also have the program send text output to a file. When we do this, the user doesn't see the text right away; they'll have to go and find the resulting file on their computer and open it using a text editor in order to see the results.

To write to a file, you have to open it in **write** mode:

```
f = open('file_to_write.txt', 'w')
```

Be careful! If a file is opened in write mode, and a file of the same name already exists, then the existing file is destroyed; a new file of the same name replaces it. If the file opened for writing does not exist yet, it is created.

It is possible to write data at the end of an existing file without destroying it. To do so, open the file in **append** mode:

```
f = open('file_to_write.txt', 'a')
```

15.4.1 The `write()` method

Writing data to text files is very similar to printing to the console. First you have to open a file in **write** or **append** mode. Then, instead of using the `print` function, you use the `write()` method, which is a method of file objects. If the variable `f` refers to a file object, and the file was opened in **write** mode, then the code

```
f.write(string)
```

writes the the string *`string`* to the file. The `write` method does not write a newline character to the file unless the string given as an argument includes one. Note that this behaviour is different from the `print` function which, by default, always outputs a newline after printing its argument.

15.4.2 Writing List Files

List files can be written by writing each data item followed by a new line. If we have a list of strings, we can write those strings, one per line, to a file called `shoppinglist.txt` like this:

```
ingredients = [’eggs’, ’milk’, ’flour’, ’yeast’]
f = open(’shoppinglist.txt’, ’w’)
for i in ingredients:
     f.write(i + ’\n’)
f.close()
```

This code iterates over each item in the list `ingredients`, and writes it to the file. Note how we concatenate each item in the list with a newline before writing it so that each string appears on its own line. The resulting file looks like this:

```
eggs
milk
flour
yeast
```

If the items we are writing are not strings, we have to convert them to strings because the `write` method can only write strings to files. We can do this using the built-in `str()` function which converts its argument to a string, if possible. Here’s how we would write a list of integers to a file, one per line:

```
ingredients = [99, 88, 77, 66, 55]
f = open(’numbers.txt’, ’w’)
for i in ingredients:
     f.write(str(i) + ’\n’)
f.close()
```

Note how the integer `i` is converted to a string prior to concatenating it with a newline.

15.4.3 Writing Tabular Files

To write a tabular file, a typical strategy is to construct a string consisting of one line of the tabular file to be written, and then write it. This is done by combining the data items to appear on that line into a single string, separated by the appropriate delimiter. Just as strings have a `split()` method for breaking them up, there’s also a `join()` method for putting them together. It works in a bit of a funny way though, at least at first glance. The string on which we call the `join()`, again using the period-notation we use for method calls, is **the separator** that we want to use, and items that we want to join together are provided as a list **argument** to the method call.

Suppose we have a list of numbers called `grades` which should all appear on one line of a tabular file, separated by commas. We can construct the appropriate string to write to the file like this:

```
grades = [42, 24, 87, 21, 76]
# first we have to convert numbers to strings
grades_string = []
for g in grades:
     grades_string.append(str(g))
# now join grades together using commas
line = ’,’.join(grades_string)
print(line)
```

This produces the output:

```
42,24,87,21,76
```

16 — Dictionaries

Learning Objectives

After studying this chapter, a student should be able to:

- describe what a dictionary is
- distinguish dictionaries from lists
- access and manipulate data stored in dictionaries
- write Python programs that use dictionaries

16.1 Dictionaries

A *dictionary* associates pairs of data items with one another. The first item in such a pair is called a *key* and the second item is called the *value*. A dictionary stores a collection of these key-value pairs. Dictionaries allow you to look values up by their key.

Dictionaries are named as such because in some ways they are similar to the physical dictionaries we use to look up the meaning of words. In a physical dictionary, we look up definitions based on the word we are interested in. It doesn't really matter to us where that word is in the dictionary so long as we can find it when we want it. With physical dictionaries, it is almost never useful to ask questions like "what's the definition of the 356th word in the dictionary?" We usually know the word we want to look up, and we just need to get the value (definition) associated with that word.

Of course, in Python, dictionaries can be used for any kind of data look-up, not just looking up word definitions. Suppose we had a dictionary called *friends* containing key-value pairs where the keys are people's names, and the value associated with each key is that person's email address. We could then find out someone's email address by querying the dictionary for the value associated with a person's name. If there is a key-value pair in the dictionary `friends` whose key is `'John Smith'`, the value of `friends['John Smith']` would be the email address of John Smith. The keys of a dictionary must be unique — the same key cannot be associated with more than one value.

However, the values need not be unique — different keys can be associated with the same value. Thus, there can only be one `’John Smith’` key in `friends`, but another friend with the key`’Jane Smith’` may have the same e-mail address as John Smith.

Over the next few sections, we’ll see how to create a dictionary and look up the items it contains.

16.1.1 Creating a Dictionary

Dictionaries can be created in a few different ways. They can be literally written out like a list, except dictionaries are enclosed in curly braces rather than square brackets. We can construct an empty dictionary using an empty pair of curly braces:

```
# associate the variable name ’friends’ with an empty dictionary
friends = {}
```

We can create a non-empty dictionary by writing a comma-separated listing of key-value pairs within a pair of curly braces. Each key-value must consist of the key, followed by a colon, followed by the value. The following defines a dictionary with four key-item pairs; each pair is a name and an email address:

```
# associate ’friends’ with some known key-value pairs
friends = { ’Bilbo Baggins’     : ’burglar1@theshire.net’,
            ’Sauron the Great’ : ’greateye@mordor.gov’,
            ’Gandalf the White’: ’whitewizard@valinor.org’,
            ’Saruman’           : ’entkiller@isengard.gov’ }
```

We can have line breaks and line indentations between key-value pairings within the curly braces because Python ignores whitespace within curly braces (the same applies to square brackets enclosing lists as well!).

Dictionary **keys** can be of any of the basic data types we’ve seen so far: integers, floats, and strings. However, you are not allowed to use lists or other dictionaries as keys.

Dictionary **values**, on the other hand, may be of **any** type, including lists, or even another dictionary!

16.1.2 Looking Up Values by Key

Looking up values by key in a dictionary works very much like indexing a list. You write the variable name that refers to the dictionary, then a pair of square brackets enclosing the key whose value you want to look up. You read that correctly, **square brackets**. So even though we use curly brackets to create or initialize a dictionary, we still use square brackets to access individual items, just like with a list.

```
print(friends[’Bilbo Baggins’])
# this will print burglar1@theshire.net
print(friends[’Sauron the Great’])
# this will print greateye@mordor.gov
```

If you try to look up a key that is not in the dictionary, you get a `KeyError`. This is similar to when you try to index a list at a position that doesn’t exist. Again, the fact that Python gives you an error here is very useful! Not all programming languages are so helpful.

16.1.3 Adding and Modifying Key-Value Pairs

You can add a key-value pair, or modify the value associated with a key using the same syntax as a lookup in conjunction with the assignment operator.

```
# add Haldir's email address to the dictionary
friends['Haldir'] = 'smug_elf_531@lothlorien.net'

# update Saruman's email address
# (This is an update since key 'Saruman' is already in
# the dictionary)
friends['Saruman'] = 'bag_end_squatter@theshire.net'
```

Adding and modifying keys look very much the same. If the key already exists in the dictionary, the existing key becomes associated with the new value on the right of the assignment operator. This was the case for ‘Saruman’ in the example above, which was already a key in our dictionary and had the previous value of ‘entkiller@isengard.gov’’. If the key does NOT exist in the dictionary, it is added and becomes associated with the value to the right of the assignment operator, like in the case of the key ‘Haldir’ in the example above.

16.1.4 Removing Key-Value Pairs

Dictionaries have a pop() method that works more or less the same as th pop() method that we saw for lists.

```
# remove the pair with key 'Saruman' from the dictionary,
# after associating its value with the name value
value = friends.pop('Saruman')
```

Dictionaries also have a method called clear() that will remove **everything** from the dictionary.

```
# remove everyone's email address from the dictionary
friends.clear()
```

16.1.5 Checking if a Dictionary has a Key

The in operator, which we have seen many times before, can be used to determine if a dictionary has a particular key.

```
if 'Sauron the Great' in friends:
    print('Yeah, I am friends with Sauron')
else:
    print('Sauron is not my friend.  I hope his tower collapses.')
```

16.1.6 Iterating over a Dictionary's Keys

You can iterate over all keys in a dictionary, and do something with each key's corresponding value using a for-loop.

```
spam_addresses = []
for k in friends:
     # add all my friends’ email addresses to list of spam recipients
     spam_addresses.append(friends[k])
```

Note that this is a little different than using a for-loop to iterate over a list. With a list, the for-loop gives us the values that are stored in the list, one at a time. With a dictionary, the for-loop gives us the **keys** for each key-value pair in the dictionary. We still have to use each key to get the value associated with a key.

It is important to note that there is no guarantee on the order in which each key of `friends` is processed in such a loop. Real physical dictionaries are organized alphabetically, but that’s only so that humans can quickly and easily find what they’re looking for. In Python, the computer takes care of doing the look-up (using techniques that are indeed fascinating, but beyond the scope of this course), so the organization is not necessarily alphabetical at all.

16.2 Dictionaries vs. Lists

Dictionaries are similar to lists in the following ways:

- both are *containers* that hold a collection of data items;
- both allow storage of data items of different types; and
- both allow you to look up individual data items.

Dictionaries are different from lists in the following ways:

- there is no ordering of the key-value pairs stored in a dictionary, whereas items in a list are in a specific order; and
- values in a dictionary are looked up by their key, whereas items in a list are looked up by their integer index (position in the ordering).

16.3 Common Uses of Dictionaries

In this section, we will discuss some common data storage patterns that can be realized with dictionaries.

16.3.1 Dictionaries as Mappings

Dictionaries, by definition, associate keys with values. Such an association can be viewed as a mapping that translates one type of data into another. For example, we could use a dictionary to map animal species names to their taxonomical class:

```
species_to_class_mapping = {
     ’red squirrel’: ’Mammal’,
     ’komodo dragon’: ’Reptile’,
     ’chimpanzee’: ’Mammal’,
     ’snowy owl’: ’Bird’,
     ’green cheeked conure’ : ’Bird’,
     ’rainbow trout’ : ’Fish’
}
```

Now we can use this mapping to look up what basic type of animal a certain species is.

When a dictionary is used to store a mapping, we can think of the dictionary as a collection of many individual data items, much like a list. But we can also think of dictionaries in other ways as well, as we will see next.

16.3.2 Dictionaries as Records

Another common use of a Python dictionary is to represent a *record*. A *record* is a group of related named data elements, for example, the spaces that get filled out in a form, such as name, address, phone number, etc. Note that the term *record* is not specific to a particular programming language but rather is a name for a data organization paradigm. The main purpose of a record is to store, as a group, several pieces of data that can be accessed via named fields, rather than through a numeric index.

Records are defined by the names of the data items, and the type of the data items. If we were studying the history of pirates, we might want to define a record that has five data items: given name, family name, pirate name, birth year, and death year. Such a record might be used to store and group together all of the data we want to collect about one pirate. An example of such a record might be:

given_name	Edward
family_name	Teach
pirate_name	Blackbeard
birth_year	1680
death_year	1718

Most programming languages support some way of defining and handling records. In Python, records are stored as dictionaries. The names of the data items in a record are a dictionary's keys, and a dictionary's values are the values associated with each data item. The record shown above would be stored in Python as the following dictionary:

```
pirate1 = { 'given_name': 'Edward',
            'family_name': 'Teach',
            'pirate_name': 'Blackbeard',
            'birth_year': 1680,
            'death_year': 1718 }
```

Now we can look up data about a particular pirate based on the name(s) of the field(s) that we're interested in. For example, given the above dictionary, we could compute Blackbeard's age when he died:

```
pirate_age = pirate1['death_year'] - pirate1['birth_year']
```

When a dictionary is used as a record, we can think of the entire dictionary as a **single** data item with several properties. In the example above, we used one dictionary to represent all the relevant information for one pirate. If we wanted to store data for multiple pirates, we would need multiple dictionaries - one per pirate.