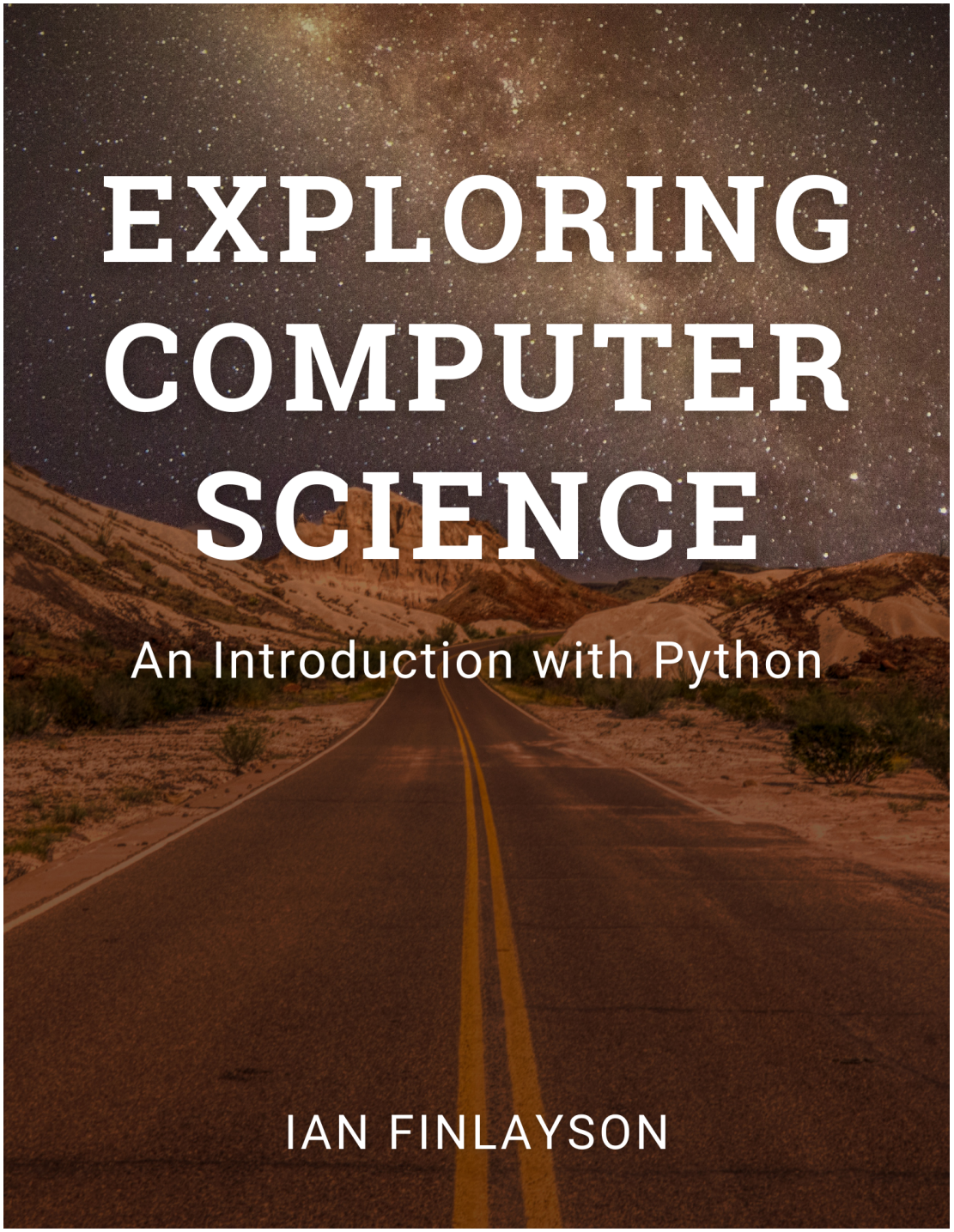

Figure 1: Exploring Computer Science

Preface

This book is designed to introduce the reader to the field of computer Science. It is intended to be used in a one-semester college level course. It could also be used for self-study.

This book uses the Python programming language for coding. However the goal is not really to cover Python itself. Instead Python is just used as a means to learn about developing algorithms and solving problems. In particular, we will not spend too much time talking about Python-specific features. We will also not always do things in the most Python-specific way. For example, we will look at how to solve some problems "by hand" that could be done just by calling Python methods. The goal of this book is to teach you the core ideas of computer science which are independent of any particular language.

Chapter 1: The Road Ahead

Learning Objectives

- Understand the concept of an algorithm.
- Understand how computers work and execute programs.
- Become familiar with what you can do with computer science.

1.1 What is Computer Science?

This book is an introduction to the field of computer science, so we will begin by talking about what computer science is. Unlike other areas, like biology or history, this may not be that obvious. People often are confused about what computer science actually is all about. In particular, it is **not** any of the following things:

- The study of how to build computer hardware (this is "computer engineering").
- The practice of setting up computer systems and installing things on them (this is "information technology").
- The use of computer applications such as email clients, word processors and spreadsheets (this is "computer literacy").

Instead, computer science is all about algorithms. An **algorithm** is a detailed, step-by-step procedure for solving a particular problem. Algorithms are essentially instructions that tell us how to solve a problem from beginning to end. They are similar to recipes. When you follow a recipe, you also perform the instructions given one by one. The difference is that the result of your work when following a recipe is a food of some kind (hopefully cake). The result of your work following an algorithm is a solution to a problem.

The study of algorithms actually predates the existence of computers by thousands of years[1]. One of the first known algorithms was described by the Greek mathematician Euclid in his book *Elements* around 300 BCE. The algorithm was to find the greatest common divisor between two numbers. For example, the greatest common divisor of 12 and 30 is 6 because 6 is the biggest number that goes into both 12 and 30 evenly. We'll take a look at Euclid's algorithm in Chapter 6.

Other mathematicians devised algorithms for solving other problems, such as adding, multiplying, factorizing, finding roots of equations, etc. The word "algorithm" itself comes from the name of Muḥammad ibn Mūsā al-Khwārizmī, a Persian mathematician who wrote on algorithms for solving algebra and arithmetic problems. The word "algebra" also comes from the title of one of al-Khwārizmī's books.

You actually have learned several of these sorts of mathematical algorithms yourself while in school. For example, if I asked you to add 137 to 226, you could do so (even though you probably have never added these specific numbers before). That's because

[1]Before the advent of computers, the study of algorithms was not called "computer science". It was just part of mathematics. After computers were invented, it split off into its own field.

Figure 2: Euclid

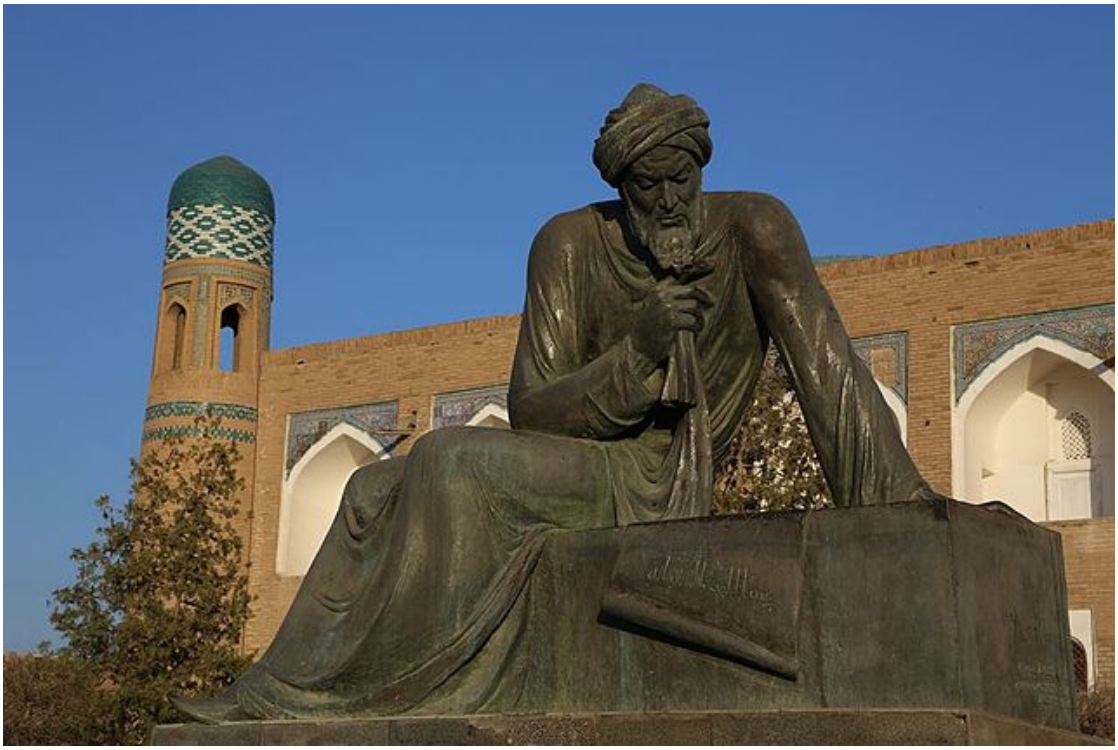

Figure 3: Al-Khwarizmi

you learned an algorithm as a child for adding numbers like this. That algorithm will allow you to add any two numbers by following its step-by-step process.

One really important aspect of algorithms is that you can use them to solve problems *whether or not you understand how the algorithm is working*. This was important in the past because it meant you only had to be really clever once, to come up with the algorithm in the first place. After that you, and anyone else, could just follow the algorithm's instructions to solve new problems with it. As a child, you didn't have to understand all of the logic behind the adding algorithm to use it. Nowadays this is even more important because most algorithms are not followed by people, but by computers. Computers don't understand anything, but they can be made to follow the steps of an algorithm automatically.

1.2 Algorithm Design

One major part of computer science is *designing* algorithms. As we will see throughout this book, there are many problems that can be solved with a good algorithm. Designing algorithms can be a fun and challenging activity. It involves both sides of your brain in that it takes both logic and creativity to do.

Let's look at an example of an algorithm to solve the "guess the number" game. In the simplest version of this game, one player thinks of a number between 1 and some upper limit. Then the other player guesses which number they picked until they get it.

Let's say that Mark and Sofia are playing this game with an upper limit of 10. The game might go like this:

```
Mark: I'm thinking of a number between 1 and 10.
Sofia: Is it 7?
Mark: No.
Sofia: Is it 3?
Mark: No.
Sofia: Is it 2?
Mark: No.
Sofia: Is it 8?
Mark: Yes.
```

In this case, Sofia solved the problem by eventually guessing Mark's number. Now we'll consider writing an algorithm which will always solve the game, no matter what number Mark picks.

Sofia tried numbers in a sort of random order until she hit on the right one. Most people would do this, but it isn't really necessary. Instead we could just start at 1, and then guess 2, and then 3 and so on until we get the right number. So this particular algorithm won't solve the problem the same way a person might, but that's OK. An algorithm for doing this could be written as:

Algorithm 1

```
1. Set G to 1.
2. Ask if their number is G.
3. If it was, then we are done!
4. If it was not, then add 1 to G.
5. Go back to step 2.
```

This algorithm[2] works by keeping track of which number we are going to be guessing next, which we call "G". This is a **variable**, which is a very important concept in programming. A variable is a name that we give to a value which may change as the algorithm is run. They are like variables in math, but are used somewhat differently. In math, you often have to solve for a variable which has one value that you just need to find. In computer science, variables change values as the algorithm progresses. Here, G starts off at 1, but it will change.

In step 2, we ask the other player if their number is G or not. Of course we don't ask them if they literally picked "G". Only numbers are allowed in this game, not letters! Instead the algorithm means that we should instead ask if their number is the *current value* of G, whatever that happens to be. The first time it will be 1, but as we have said, G will change.

In step 3 and 4, we are going to do different things based on whether or not the guess was correct. If it was, then the algorithm is done. If not, we add 1 to the variable G. When the algorithm first reaches this step, it will change G from 1 to 2.

Step 5 is crucial here. It tells us to go back to step 2. This creates a **loop**, which is when an algorithm does the same step or steps multiple times. Even though only step 2 makes a guess, the algorithm can keep on guessing numbers because of the loop.

We'll now trace through the algorithm to see how it works. Let's suppose that we pick 3 as our number and see if the algorithm can guess it. The algorithm will go step by step as follows:

1. It will start with step 1 and set G to 1.
2. Next it will go onto step 2 and ask us if the number is G (currently 1).
3. We will respond that no, it is not.
4. The algorithm will then skip over step 3 (we aren't done yet) and go to step 4.
5. In step 4, the algorithm will add 1 to G. Now G is going to be 2.
6. Step 5 will then move the algorithm back to step 2.
7. Back at step 2, the algorithm will ask us if the number is G (which is now equal to 2).
8. We will tell it no.

[2]This algorithm is written in "pseudocode" which is not real computer code, but is similar to computer code. For most of this book we will use real code, but for these first examples, we just want to talk about the concepts in general.

9. The algorithm will again skip over step 3 and go to step 4.
10. In step 4, the algorithm will again add 1 to G. This changes it from 2 to 3.
11. Step 5 sends the algorithm back to step 2 again.
12. The algorithm will again ask us if the number is G (which is now 3).
13. This time, we tell it yes, since our number was 3.
14. The algorithm then sees this in step 3 and stops.

One important part of working with algorithms is *testing* them. This involves stepping through the algorithm line by line like this to see how it's working. Hopefully you're convinced that the algorithm will try every number until it gets the right one.

1.3 Another Version of the Game

Next we will look at a more interesting variation of this game. In this variation, instead of just answering "no" for an incorrect guess, the player will either say that the guess was too low, or too high. Again, consider an example with two players. This time, the limit will be 100:

```
Mark: I'm thinking of a number between 1 and 100.
Sofia: Is it 50?
Mark: Too high.
Sofia: Is it 25?
Mark: Too low.
Sofia: Is it 37?
Mark: Too low.
Sofia: Is it 44?
Mark: Too high.
Sofia: Is it 40?
Mark: Yes.
```

This time, Sofia did not just guess numbers randomly until she hit on the right one. By having the extra information, she's able to guess more intelligently. Her first guess of 50 was the best she could have done to start with. The reason is because that way, no matter what Mark answered, she had half the potential numbers eliminated. When Mark said 50 was too high, Sofia knew the number must have been between 1 and 49. Had Mark instead answered that 50 was too low, then Sofia would know the number had to be between 51 and 100. Either way, half the possibilities were cut out.

Sofia used this trick again with her second guess. When she knew the number was between 1 and 49, she guessed right in the middle, which gave her 25. To get this "middle value", we can just add the numbers and divide by 2. She kept on doing this until she hit on the right number. This algorithm can be given like this:

Algorithm 2

```
1. Set min to 1.
2. Set max to 100.
3. Set G to (max + min) ÷ 2 (rounding down if needed).
4. Ask if their number is G.
5. If it is, then we are done!
6. If the guess was too high, set max to (G - 1).
7. If the guess was too low, set min to (G + 1).
8. Go back to step 3.
```

This algorithm is just bit more complicated than the last one. Now we have three variables involved. `min` is used to keep track of the smallest number the other player could be thinking of. Likewise `max` keeps track of the biggest number it could be. For example, if we have narrowed it down so we know the number is between 20 and 40, then min would be 20 and max would be 40. These variables will change as we narrow down the possibilities. `G` is once again used for the number we are going to guess.

The algorithm starts by setting min and max to reflect the fact that the number could be anywhere between 1 and 100 to start with. Then step 3 figures out which number to guess. The first time we do this, we will get 100 plus 1, divided by 2, which gives 50.5. The note about rounding down if needed is to make sure we always guess a whole number, in this case 50.

After each guess, there are 3 possibilities. If we got the guess right, then we are done, just like before. If we guessed too high, then that means that we need to change our max variable. We set it to whatever our last guess was, minus 1. To see why, consider the starting case where min is 1, max is 100, and G is 50. If the guess of 50 was too high, then the number must be between 1 and 49. So we 49 as the new max value. Similar logic holds for when the guess was too low.

Finally we go back to step 3 to pick a new guess again. This algorithm will eventually guess the correct number for playing this version of the game. I would encourage you to try it yourself once or twice to convince yourself that it works.

The things these two algorithms are doing, such as changing variables, checking different conditions, and going back to previous steps, are the same things pretty much all the algorithms we will look at in this book will do. You'll see that we can use these building blocks to solve all kinds of interesting problems.

1.4 Algorithm Analysis

Another major part of computer science is *analyzing* algorithms. Oftentimes a problem will have more than one algorithm to solve it. We then will want to compare the algorithms and see which takes less steps. We now have two algorithms we can analyze. Notice that we could use both algorithms for the second variant of the game (where the player answers too low or too high instead of just no). The first algorithm would just

guess starting at 1 all the way up until it got the number. But which algorithm is better? And by how much? These are the questions of algorithm analysis.

When analyzing algorithms, we want to know how many steps they take in different cases. We often consider the average case or the worst case. The average case is helpful because it gives you an idea of how long the algorithm will usually take. The worst case is helpful because it gives you a guarantee on how long it will take. If the algorithm performs well enough in the worst case, you know it will work well for you. The best case is usually not very interesting. Here we will focus on the worst case.

Let's consider algorithm 1 first (the one that just starts at 1, then 2, then 3, etc.). If we are playing the game between 1 and 100, what's the most number of guesses that the algorithm could take? Clearly the worst case is that the other player chose 100 as his or her number, because that would be the algorithm's last guess. In this case, we would have to make 100 guesses before we get it right. If we were playing between 1 and 1,000 then we would need 1,000 guesses in the worst case.

Analyzing algorithm 2 is more complicated. The worst case with this algorithm is that we never "get lucky" by guessing the number correctly until we are 100% sure of what it is. We are only 100% sure when we have narrowed down the range to 1 number, and min and max are the same value. Below is an example of when this could happen:

```
> min = 1, max = 100.  Is your number 50?
Too high.
> min = 1, max = 49.  Is your number 25?
Too low.
> min = 26, max = 49.  Is your number 37?
Too low.
> min = 38, max = 49.  Is your number 43?
Too high.
> min = 38, max = 42.  Is your number 40?
Too high.
> min = 38, max = 39.  Is your number 38?
Too low.
> min = 39, max = 39.  Is your number 39?
Yes.
```

As you can see, the algorithm was not lucky enough to get the number right until the very end when it had eliminated all the other possibilities. In this worst case it took 7 guesses to get the number. There are other worst case numbers, but they all take 7 guesses to reach.

The number 7 here comes from the number of times we can cut the possibilities in half before we run out. There were 100 possibilities to begin with, and with each guess we eliminate half of them. If we repeatedly divide 100 by 2 (rounding down because we eliminate the guessed number itself as well), then we get this sequence of numbers:

```
100
50
25
12
6
3
1
```

Because we can cut 100 in half 7 times before we hit 1, our algorithm takes 7 guesses in the worst case[3]. Below is a table showing the number of guesses needed in the worst case for both algorithms, based on the highest number the other player could pick:

Highest Number	Algorithm 1 Guesses	Algorithm 2 Guesses
10	10	4
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

It is sort of amazing that we can ask algorithm 2 to guess a number between 1 and 1 billion, and it will need only 30 guesses at most to get it! Algorithm 2 is not just a bit better for this problem, it's way better. If you were to follow Algorithm 2 with a billion as the highest number, you'd be done in a few minutes. If you used Algorithm 1, and you guess one number per second (with no breaks for eating and sleeping), it would take you more than 31 *years*.

It's actually pretty common in computer science for there to be huge differences in speed between different approaches like this. Oftentimes the obvious solution is not very efficient, but a more clever one is. This book will spend more time on algorithm design than analysis, but it is an important part of computer science and we will touch on it from time to time.

1.5 Another Algorithm Example

Let's look at another example of an algorithm. Imagine you were asked to add up all of the numbers from 1 to 100. When coming up with an algorithm for a problem I find it's best to begin thinking about how you would solve the problem manually. In this case, we might follow steps like the following:

1. Start by writing down our first number, 1.

[3] For the mathematically inclined, the worst case number of guesses has the following relationship with the number of possibilities we start with: guesses $= \log_2(\text{possibilities})$

2. Take the second number and add it to the first giving us 3. Here we'll scratch out the 1 and replace it with the 3.
3. Move to the next number, which is 3 and add it in, giving us 6. Again we replace the 3 with the 6.
4. Move to the 4, and again add it in...

Thinking this through, there's basically two things we are keeping track of: which number we are currently on, and what our total sum is so far. In algorithms, the way we keep track of things is with variables. So let's make a variable called *current* for keeping track of what number we're on, and one called *sum* for keeping track of the running total.

With these variables, the algorithm might look like this:

Algorithm 3

```
1. Set current to 1.
2. Set sum to 0.
3. Set the sum equal to the current sum + current.
4. If current is equal to 100, stop we are done.
5. Add one to current, moving to the next number.
6. Go back to step 3.
```

This algorithm will run through all the numbers 1 to 100 and compute the total. If we were to give it to a careful and patient person, they would find that the sum is equal to 5,050. We could also write this algorithm in a language that a computer can understand (a topic we'll spend most of the rest of this book covering) and the computer would be able to give us that answer much more quickly and with no chance of a mistake.

However, this problem also happens to have a more efficient algorithm! The more efficient algorithm was supposedly discovered by the mathematician Carl Friedrich Gauss when he was in elementary school. The story (which may or may not be apocryphal) goes that Gauss's teacher gave him exactly this problem to keep him occupied and was surprised when Gauss came back after only a few minutes. The way Gauss solved the problem was by realizing you could rewrite the numbers like this:

```
  1 +   2 +   3 +   4 + ... +  98 +  99 + 100
100 +  99 +  98 +  97 + ... +   3 +   2 +   1
-----------------------------------------------
101 + 101 + 101 + 101 + ... + 101 + 101 + 101
```

The first row is the numbers 1 through 100 added together. The second is the numbers 100 through 1 added together (which results in the same sum of course). When we add the two sets of numbers together like this, every column is equal to 101. To add the one hundred 101's together we don't actually need to add. We can just multiply 101 times 100, giving us 10,100. Then we need to divide by 2, because we added two sets of numbers 1 to 100 instead of one. That gives us the answer 5,050 just as the other algorithm did.

Using this technique, we can write a faster algorithm. For clarity, we'll make a variable for the number we are adding up to, which could be 100 or any other number we wish. We'll call this variable "N".

Algorithm 4

```
1. Set the ending number, N, to 100.
2. Set sum = ((N + 1) * N) / 2
```

Despite being perhaps harder to understand, algorithm 4 is more efficient than algorithm 3 for solving this problem. To add the numbers 1 through 100, we would do 99 additions with algorithm 3. With algorithm 4, we would do one addition, one multiplication, and one division. Like the guess the number example, the differences between these algorithms also increases as we look at bigger numbers:

Highest Number	Algorithm 3 Operations	Algorithm 4 Operations
10	9	3
100	99	3
1,000	999	3
1,000,000	999,999	3
1,000,000,000	999,999,999	3

Again the choice of algorithm can be very important in terms of how long it takes to solve a problem!

1.6 What is a Computer?

Computer science is primarily the study of algorithms, but computers do play a role as well. In this section we will talk about what computers are, what programs are, and how computers can run programs.

It might seem a bit silly to define what a computer is, since you likely use a computer every day and clearly know what one is. However, we will define a **computer** as any device that is capable of running algorithms automatically. The earliest computers were created to run a handful of particular algorithms, and couldn't do anything else.

One of the earliest such devices was Pascal's calculator, also called a Pascaline. This device, created by the French mathematician Blaise Pascal in the mid-1600s, was capable of adding and subtracting numbers. It could also do multiplication and division by means of repeated additions or subtractions. A number of other mathematicians and inventors made similar mechanical devices, including Gottfried Wilhelm Leibniz.

Another pioneer in early computers was the British mathematician and engineer Charles Babbage. He designed the difference engine, which could compute polynomial values.

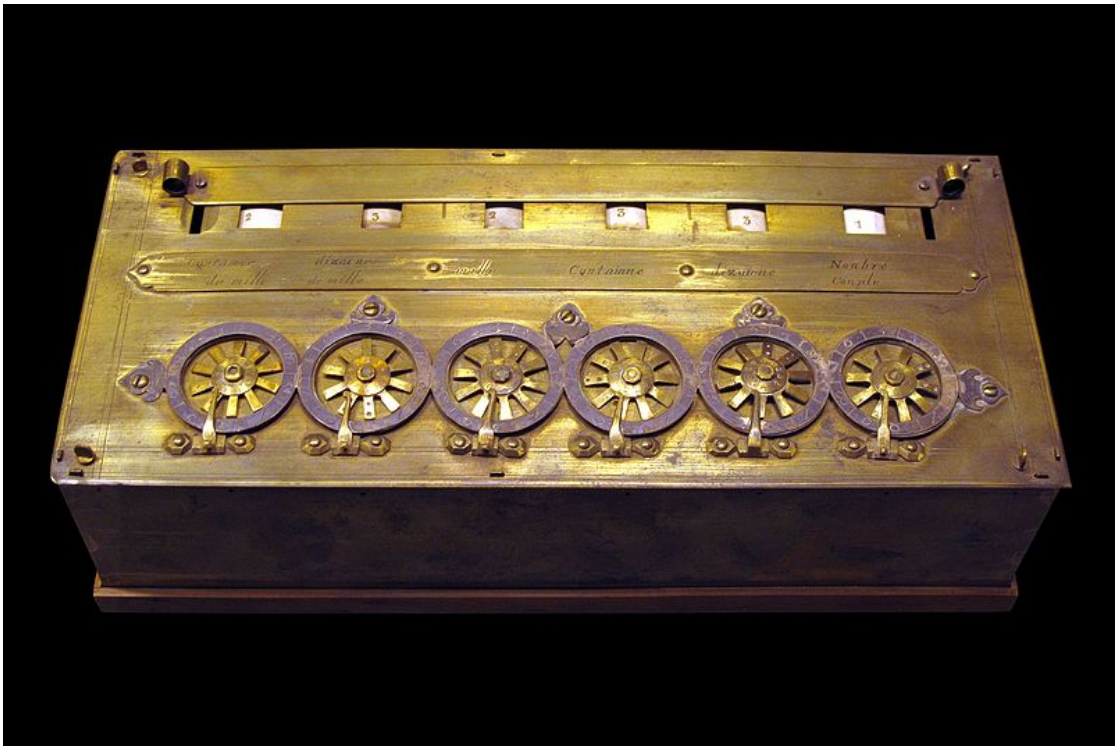

Figure 4: Pascal's calculator (1649) (© Rama, Wikimedia Commons CC BY-SA 3.0 FR)

He completed a prototype of the difference engine in 1822. The difference engine was mechanical and was operated with a hand crank. He began working on a larger version which could operate on bigger numbers, with more precision.

Before the full difference engine was completed, however, Babbage began working on a more ambitious project, the analytical engine. Babbage's design of the analytical engine was a huge breakthrough in computer science, because it was the first design for a *programmable* computer. Pascal's calculator, the difference engine, and every other computing device up until that point was designed to do one fixed task. If you wanted it to solve a different problem, you had to build an entirely new machine.

The analytical engine, on the other hand, was designed so that it could execute any algorithm at all. It did this by taking the instructions that it should execute as input, along with data values to be used in the computation. These sets of instructions are the world's first computer programs. The analytical engine's programs themselves were created by punching holes in cards. The machine would then read the cards in, and the patterns of holes would affect its behavior.

Babbage worked with Ada Lovelace, who was incidentally the daughter of the poet Lord Byron. She worked on translating algorithms so that they could be executed by the analytical engine. The first of these was a program to compute the Bernoulli numbers.

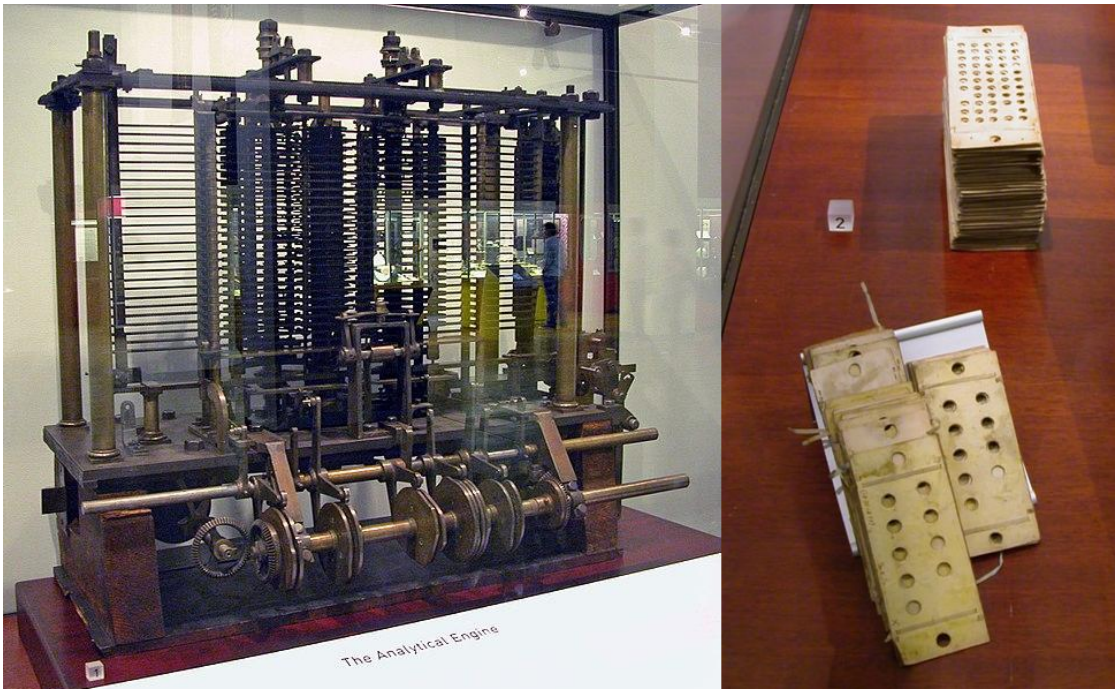

Figure 5: Left: a model of part of the analytical engine (© Bruno Barral, Wikimedia Commons, CC BY-SA 2.5). Right: punched cards for use with the machine (© Karoly Lorentey, Wikimedia Commons, CC BY 2.0)

This was the first published computer program. A **program** is an algorithm or collection of algorithms that is written specifically for a computer to execute. Unlike Babbage and others at the time, Lovelace believed the analytical engine to be capable of going beyond number crunching, including speculating that the engine could be used to create music.

Unfortunately, the analytical engine was never completed, primarily due to a lack of funding. A general-purpose computer was not actually completed until more than 100 years after the design of the analytical engine. In the 1940s, there were several working computers developed. These include the Z4 by the German Konrad Zuse, the Colossus developed in Great Britain to break coded messages, and the ENIAC developed at the University of Pennsylvania to calculate ballistics trajectories.

Unlike the Analytical Engine, which was completely mechanical, these computers used electronic circuitry. After these machines were successfully built and used, there was no turning back, and computers have been constantly built and improved upon until the present day.

Despite being as large as rooms, these older computer were laughably limited compared to even the most basic of today's computers. The Z4 ran at 40 Hz and had 512 bytes of memory. The first iPhone, released in 2007, could run at 620 MHz (15 million times faster), and came with at least 4 GB of memory (7 million times more).

Figure 6: Left: Charles Babbage. Right: Ada Lovelace.

While these older computers were woefully slow and had tiny memories, they functioned in essentially the same way as every computer designed since. They all consist of a processor, memory and some input/output devices. Below is a simplified diagram of a computer system.

The **CPU** (which stands for central processing unit), is responsible for carrying out the instructions of a program. Each instruction is very simple and precise. For example, the following are examples of typical CPU instructions:

- Take the numbers in two memory cells add them together. Put the answer in another memory cell.
- Check if one memory cell is equal to another. If so, jump to a different part of the program.
- Check if a certain key on the keyboard is pressed. If so, set a memory cell to 1. Otherwise, set it to 0.

Although the individual instructions a CPU carries out are very basic, enough of them put together are able to accomplish great things.

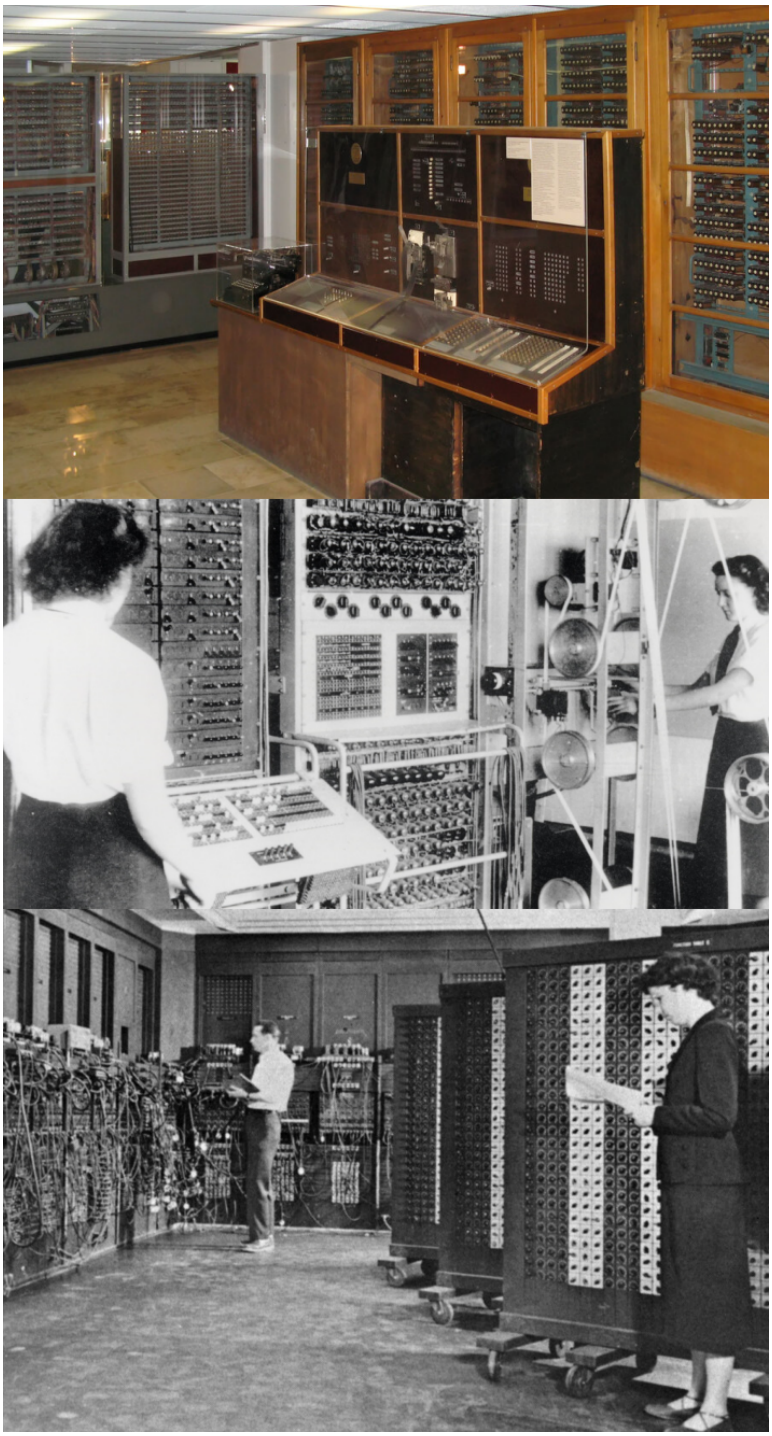

Figure 7: Early Computers Top: The Z4 (1944) (© Clemens Pfeiffer, CC BY 2.5) Middle: The Colossus (1944) Bottom: The ENIAC (1946)

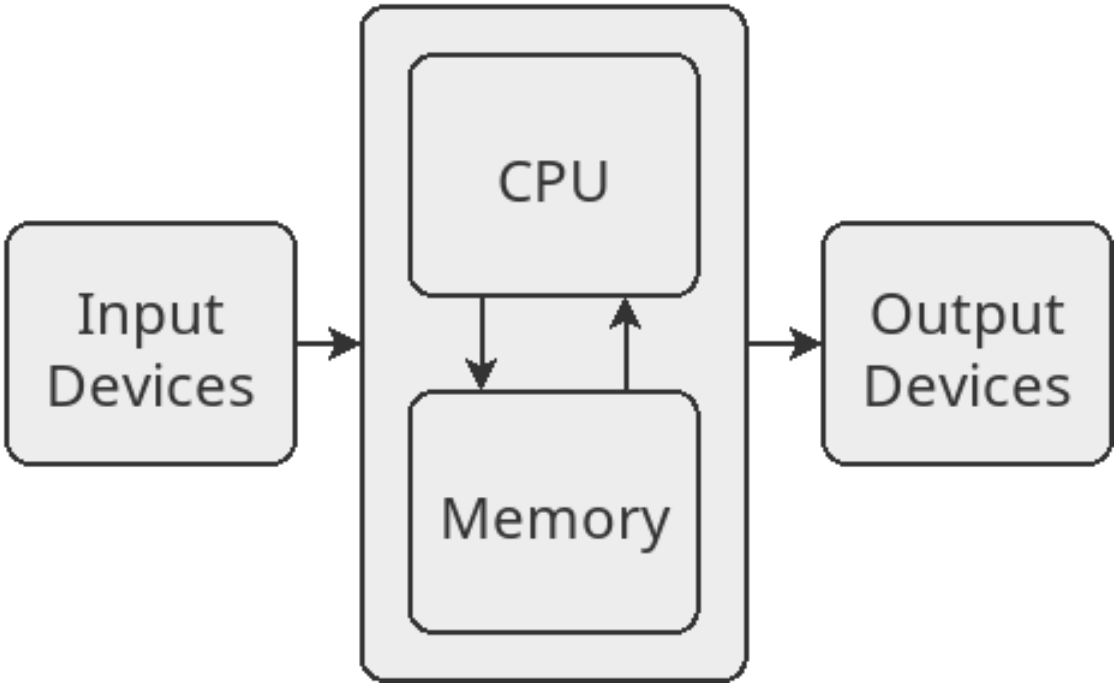

Figure 8: A functional view of a computer

The **memory** of a computer is responsible for storing numbers. Everything represented in a computer is done with numbers. For instance, when you send an email, the text that you write must be stored inside the computer as well. But, like everything else, this is done with numbers. The letters and punctuation symbols have specific numbers assigned to them. For instance when you type an 'A', the computer stores a 65. When you type a period, the computer stores a 46. When you type a space, the computer stores a 32[4].

Of course, the way that computers store numbers is in *binary*, which means they only use 1's and 0's. We will not talk about how to convert back and forth between binary and decimal numbers. Just know that any number can be stored in either binary or decimal, and they mean the same thing. Even images and sounds are stored inside a computer as numbers. We will talk about how that is done in later chapters.

Input devices include touch screens, mice, keyboards, game controllers and so on. These are connected to a computer system so that the user can influence the programs running on the computer. Likewise, **output devices** are connected to a computer so that the user can see what the computer is doing and see the result of the program which is running. These include monitors, speakers, printers, and vibration units (which can vibrate to provide the user feedback).

[4]These numbers are defined by the ASCII (American Standard Code for Information Interchange) code. Computer scientists do not need to remember these!

1.7 Programming Languages

So the computer executes simple instructions, reads and writes its memory, gets input from the user, and gives the user output. But where do the instructions come from? They are actually stored inside the computer's memory as numbers too! As an example of a computer instruction, we can look at one of the steps of Algorithm 1 to solve the simple guess the number game:

```
add 1 to g.
```

Let's look at what this step would look like as a real computer instruction[5]. But first, we would need to decide what memory location to store `G` in. Let's say we put it in location 7.

This instruction would tell the CPU to add 1 to memory location 7:

```
          operation     destination   input   amount
1110001   0100       0  0111          0111    00000001 0000
```

This is a *binary* number. We'll just point out some of the parts of this. The 0100 labeled "operation" tells the CPU what sort of thing it's doing. 0100 is the code for addition. So it tells the CPU to add instead of subtract, multiply, or anything else.

The first 0111 labeled "destination" tells the CPU where to put the result. 0111 is binary for 7. So it puts the answer in memory cell 7. The second 0111, labeled "input" tells the CPU what value to read in the addition. This is also 7. Lastly the 00000001 labeled "amount" tells the CPU how much to add. In total the instruction tells the CPU to read memory cell 7, add 1 to it, and put the answer back in memory cell 7.

The other, unlabelled parts just tell the CPU what type of instruction it is, and how to interpret the other fields. All together the instruction is the following binary number:

```
11100010100001110111000000010000
```

In decimal, this is equal to:

```
3800526864
```

So the way computers work is by reading in these numbers, which tell them what they are supposed to do. Part of the computer's memory is dedicated to storing these instruction numbers for the programs it runs. It reads the instructions one by one and carries them out. Programs stored as actual numbers like this are **machine code** programs.

[5]The instruction in this example is for ARM-based computer systems. These include almost all phones, tablets and other small computers. Most laptops and desktops use a different format, but the ideas are the same.

Figure 9: An assembler converts assembly code into machine language.

In case you are panicking right now, let me assure you that nobody actually writes programs this way! Giving a computer a program by writing a sequence of numbers like this would be tedious beyond belief.

Instead, we have created other languages which are easier for people to use. The first of these is **assembly** language, which is basically a human-readable version of machine code. Rather than write "0100" for add, and "0111" for memory cell 7, we just write them out. The instruction above written in assembly would look like this:

```
add r7, r7, #1
```

The computer can't run this instruction directly, it must be translated into machine code. That is done by a program called an **assembler**:

The assembler converts each line of assembly code into the corresponding machine code instruction. Then, the machine code program can be run on the computer system directly.

We aren't dealing with numbers directly with assembly code, but it is still too tedious for most people. In particular, we still have to keep track of where in memory our variables are stored. Knowing that memory cell 7 is storing our guess variable, and knowing where the other variables are becomes too hard as we begin to write larger programs.

Instead, we have developed **high-level** programming languages. These allow us to have names for our variables, instead of needing to remember which memory location they are in. Here is what the instruction above looks like in Python, the high-level language used in this book:

```
guess = guess + 1
```

High-level language code is also much more succinct than assembly or machine code. One line of code in a language like Python can do the same work as many lines of machine code. Of course this code must also be translated into machine code for the computer to execute. That is done with a program called an **interpreter**.

An interpreter takes code in a language like Python, and executes in line by line. For each line it sees, it gives one or more lines of machine code to the computer:

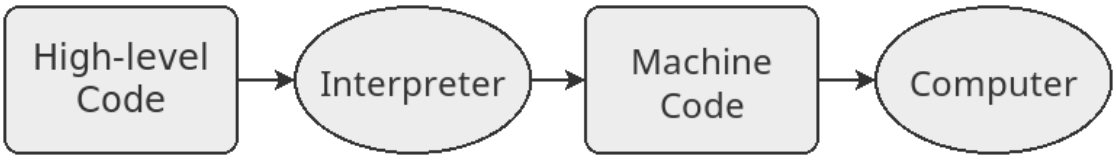

Figure 10: An interpreter translates high-level code into machine code.

Just like someone who interprets one spoken language into another, an interpreter program translates on the fly. As the high-level program is being run, its code is being translated for the computer to execute.

1.8 What do you do with Computer Science?

We have seen that computer science is primarily about algorithms, but this section will address how algorithms are used in a variety of areas to solve problems in the world.

Computer science is an interesting field because it frequently works together with other fields. For example:

- Computer scientists work with natural scientists in fields like physics, chemistry and biology to develop computer models of natural phenomenon. It is often impractical to carry out experiments in real life, so computer models can be used. For instance, there are programs which model the collision of two galaxies, the spread of a disease through a population, or the path of a river in the face of flooding. These models allow scientists to test hypotheses and compare potential solutions to problems more easily than they otherwise could.
- Social scientists work with computer scientists as well. For example, economists work with programmers to create programs which model the way markets work. Computer scientists also work with historians and anthropologists to create virtual exhibits for historical artifacts.
- Computer programs are also used in every area of industry. Programs determine the price of goods, they schedule flights, they manage factories and farms, and keep track of information in schools and hospitals. They are behind all of the web sites and apps that you use, and provide entertainment in the form of video games and special effects used in movies.

Some computer scientists work in “pure” computer science. These include people who work on operating systems or the interpreters that make languages like Python work. But the vast majority work with people in some other area. Because so many different fields rely on programs, a computer scientist can work in a variety of different areas over the course of their career. All of this makes computer science an interesting and rewarding field of study. No matter what you’re interested in, computer science can combine with it in some way.

Computer science also has a lower "barrier of entry" than most other fields. You don't need any special equipment or materials. You can use just about any sort of computer to write your programs (some people assume you need a powerful or specific type of machine, but that's not the case). It is also a field where you should feel free to experiment. Unlike a chemistry student mixing up chemicals, there's really not much you can do while programming that will cause any harm to you or your computer.

1.9 Comprehension Questions

1. What is an algorithm, and what role do algorithms play in computer science?
2. In the context of algorithms, what is a variable?
3. In the context of algorithms, what is a loop?
4. Why is it important that algorithms can be followed without understanding how they work?
5. Why is it important to analyse how long algorithms take?
6. What was special about the Analytical Engine developed by Babbage and Lovelace?
7. Why do most programmers use high-level languages instead of machine language?
8. Give an example of how another field or industry makes use of computer science.

1.10 Algorithm Exercises

1. Write an algorithm that can find the largest number in a list of 10 numbers. Begin by thinking about what information you will need to keep track of.

2. Below is Euclid's algorithm for finding the greatest common divisor between two numbers:

```
1. Set M to the biggest of our two numbers
2. Set N to the other number
3. If N is equal to 0, the answer is M.
4. Set R to the remainder of M divided by N.
5. Set M to N.
6. Set N to R.
7. Go to step 3.
```

   What does this algorithm give as the answer with the following inputs:

   a) 30 and 12

   b) 20 and 10

   c) 5 and 3

   Remember that you don't need to understand why this algorithm works to follow its steps (or indeed even know what a greatest common divisor even is).

3. In grade school you learned an algorithm for adding numbers with any number of digits. Try writing out the algorithm as a set of detailed step-by-step instructions. You can assume that the person following the algorithm can add any 1-digit numbers together (i.e. they can do 7+8 in their head, but can't add the whole number that way).

Chapter Summary

- Computer science is the study of algorithms. An algorithm is a set of directions for solving a problem. Algorithms must be precise enough that someone can use it to solve problems without needing to understand how it's working.
- A computer is a device which can carry out the steps of an algorithm automatically. The oldest computers could only solve problems they were designed for. Nowadays computers can run different algorithms at different times.
- For a computer to execute an algorithm, it must be programmed into a language that the computer can understand. Computers have a "native" machine language which is difficult for people to use. Most programs are written in an easier high-level language which is translated for the computer by an interpreter.
- Computer scientists work to write programs that solve a variety of problems for lots of different fields.

Chapter 2: Starting Out

Learning Objectives

- Understand what an interpreter, IDE, and shell do.
- Learn how to install Python on your computer.
- Know how to write and run simple Python programs.
- Become comfortable writing programs which print messages to the screen.
- Learn how to get user input into a program.
- Understand how to create variables, and the rules for using them.

2.1 Interpreters and IDEs

As we learned in the last chapter, computers only directly understand programs written in machine code. However, nearly all programs are written in a high-level language. For this to work, we need an interpreter program to translate the code into what the computer understands.

So in order to run our programs, we have to install the interpreter for the language that we want to use. Because we will be using Python, we need to install the Python interpreter. If we just give Python programs straight to our computer, it won't know what to do with them. We need to interpreter to run them.

We will also be installing *another* program along with an interpreter. This program is something called an **Integrated Development Environment (IDE)**. An IDE is a program that lets you type in the program you are writing. You could write your programs in any old program, but it's generally much easier to use a program geared just for that purpose. For example, an IDE will give you a button to pass your program to the interpreter, highlight keywords in the language and make it easier to see when you have errors.

The IDE we will be using here is a simple one called **Thonny**, which is easy to use and get started with. It also includes Python with it, so you only need to install one thing. The choice of IDE doesn't really matter too much, if you want to use a different one for some reason (for instance if you have one from another class), you can follow the rest of the book using that too.

The rest of this chapter will guide you through setting up Python along with our IDE for whichever type of computer system you have. Then, we will see how to use it to run some Python code!

2.2 Installing Python

Installing Thonny is slightly different depending on what sort of computer you have. Follow one of the following set of directions based on whether you have Windows, Mac, or Linux:

1. To install Python along with Thonny on **Windows** computers, go to the Thonny website. Hover over the Windows link in the download box at the upper right. Click the top link which is labeled "Installer with 64-bit Python". Choose to save the file. When the download is finished, run the installer program.

   Click next, select the agreement, choose where to install it, and wait for the installer to finish. Once it is done, you should have Thonny, along with its Python interpreter installed.

2. To install Python along with Thonny on a **Mac OSX** computer, go to the Thonny website. Hover over the the Mac link in the download box at the upper right. Click the link to the .pkg file and download it to your computer.

   When the download has finished, you should see the Thonny icon in a window. Drag this icon into your applications folder to copy it to your computer. You should then have it installed and be able launch Thonny from your Application menu to start programming.

3. While all recent versions of **Linux** come with Python, they do not come with Thonny. To install it, open up a terminal and run the following command:

   ```
   bash <(wget -O - https://thonny.org/installer-for-linux)
   ```

   Then hit Enter at the prompt to continue.

   That should download the latest version, and install it on your computer. You should be able to find Thonny amongst your installed applications. You could also launch it by running the command:

   ```
   ~/apps/thonny/bin/thonny
   ```

2.3 The Shell Window

You should now have Python and Thonny installed. When you run it, you should see a window something like this:

The main window has two main parts. The top is the file area. This is where you will type in the program that you will create. This is empty right now, and called "<untitled>".

The bottom area is called the **Shell**. This is a window where you can pass Python code to the Python interpreter. Any code you put in here will be run right away and the results will be given to you. Here is an example:

As you can see, when we put `3 + 4` into the shell, it gives us the answer, 7. Likewise when we put in the command `print("Hi!")`, it prints what we told it to. What's happening here is that these are small amounts of Python code. When we put them in, the shell window passes them to the Python interpreter, which runs them. Any results are

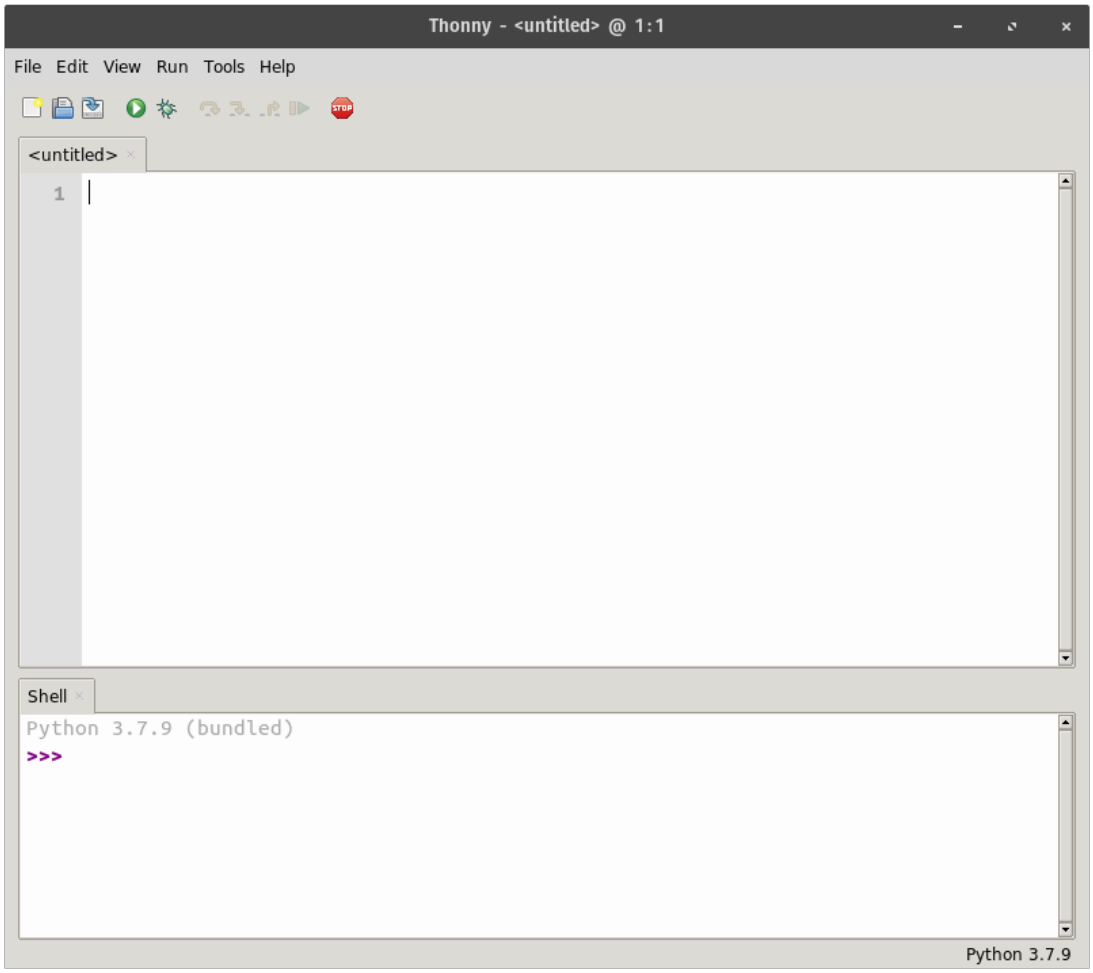

Figure 11: The main Thonny window

displayed back in the shell. It is called a “shell” because it sort of surrounds the Python interpreter and acts as our interface to it.

Generally, the top file area is for writing a program that you will run all at once. This window saves what you put there so you can run it over and over again as you work on a program. The bottom shell area is for trying things out and experimenting. Unless you copy and paste it some place else, the things you put into the shell window are not saved.

As you can see from the first example in the screenshot above, the shell can work as a calculator. Try putting a few other simple math expressions in and see how the shell gives you results back.

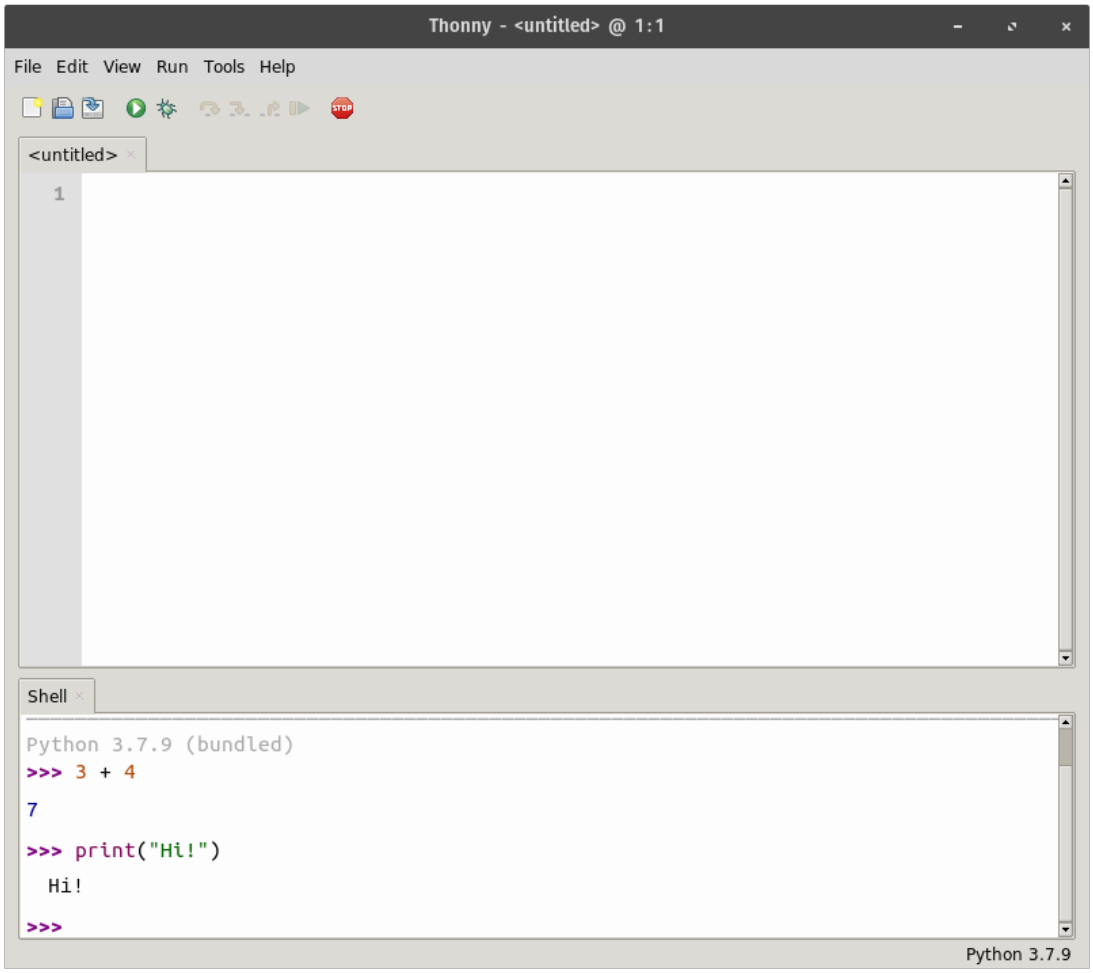

Figure 12: Some examples put into the shell window

2.4 Our First Program

Now we are ready to write our first program. The goal of the first program is just to print the text "Hello World" to the screen [1]. The code for this program is the following:

Program 2.1

```
# this is our first program
print("Hello World!")
```

[1] Having the first program print this message is something of a silly tradition in computer science. It dates back at least to the 1978 book "The C Programming Language".

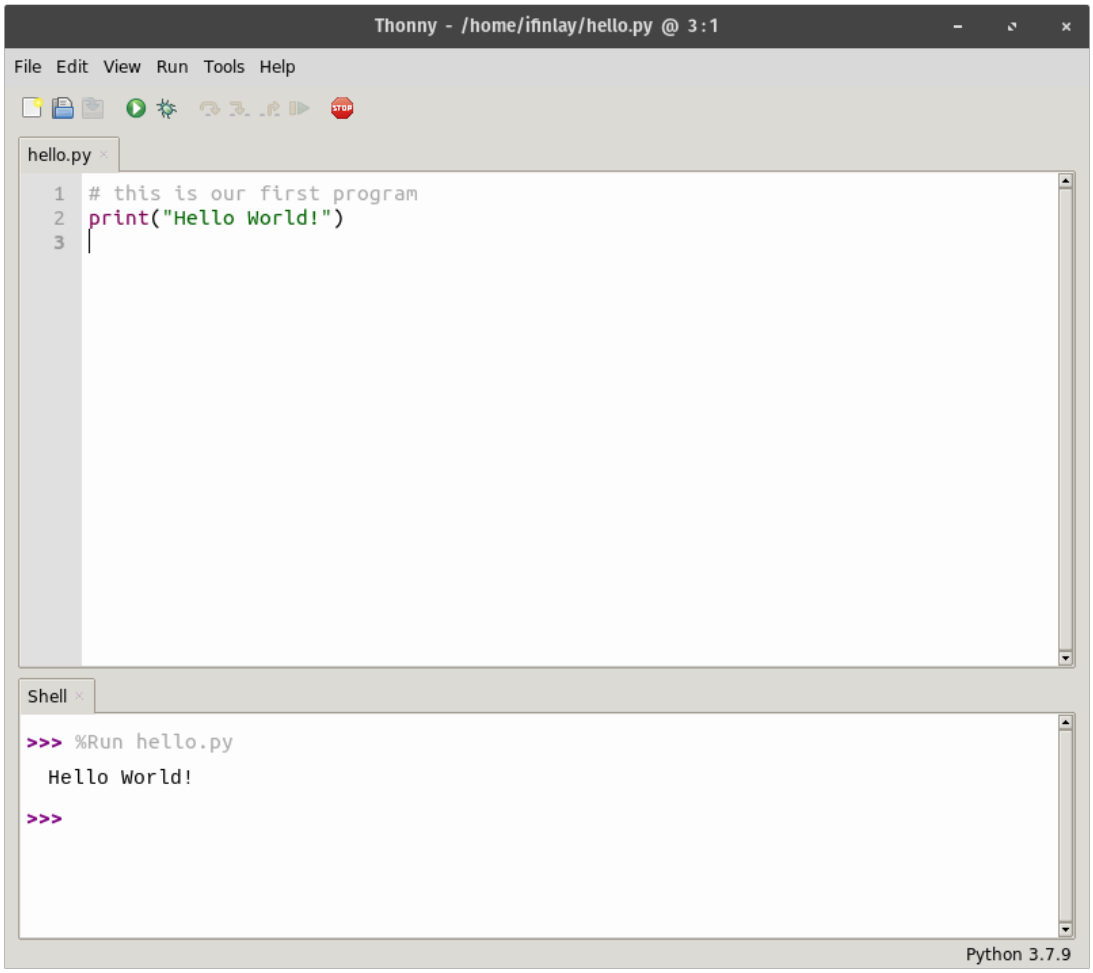

Figure 13: The results of running the program

You should type this program into the top window of Thonny. Then we can run the program. This can be done in one of three ways:

- Clicking the “Run” menu, then choosing “Run current script”.
- Clicking the green play button ( ).
- Hitting the F5 key.

Before the program can run, it will ask you to save it. When saving your program files, you should put them some place where you will be able to find them again. You should also always name them something ending with the “.py” extension.

Once the program is saved, it will run. You should then see the results in the shell window:

Now that we have seen how to run the program, we will talk about the program itself a little bit. This program consists of two lines. The first line says:

```
# this is our first program
```

This line is a **comment**. Any line that starts with the # symbol is a comment in Python [2]. When the interpreter gets a comment line, it completely ignores it, and moves on to the next line. The sole purpose of comments is to leave little notes in the code, for any people reading. They are meant to explain things about how the program works. This program is so short and simple that the comment is not really needed, but as we work on more complex programs, they'll become more helpful.

The second line of the program says:

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

This is the line that actually tells the Python interpreter to do something. Python comes with lots of commands called **functions** built in that cause it to do different things. One of these is `print`. The parenthesis mark the things that will be printed. In this case, it's just the message "Hello World!".

Both the parenthesis and the quotation marks are needed for the program to work. You can change the message inside the quotation marks to whatever you want though. Try changing it so that it prints out your name.

2.5 When Things Go Wrong

We said that the parenthesis and quotes are needed, but what happens if we get rid of them? In these cases, the Python interpreter will not be able to figure out what to do with the code, and will give us an error message. For instance, if we get rid of the quotation marks, we get this:

Here the program did not run successfully. Instead, the shell gives us the error `SyntaxError: invalid syntax`. There is also an "Assistant" window which is a feature of Thonny to help us figure the error out. In this case, it's not terribly helpful. The specific problem here is that Python has no idea what to do with the exclamation mark since that doesn't mean anything in Python code.

If we get rid of the parenthesis instead, we get a different error:

Here the error is much easier to figure out. It actually tells us what is missing and even suggests a fix for it. Even though the two errors were pretty similar, one would be much easier to figure out.

[2] I used to insist this symbol be called a "pound" or "hash" symbol, and become annoyed when it was called a "hashtag", but I've accepted it. You can say that Python comments begin with hashtags.

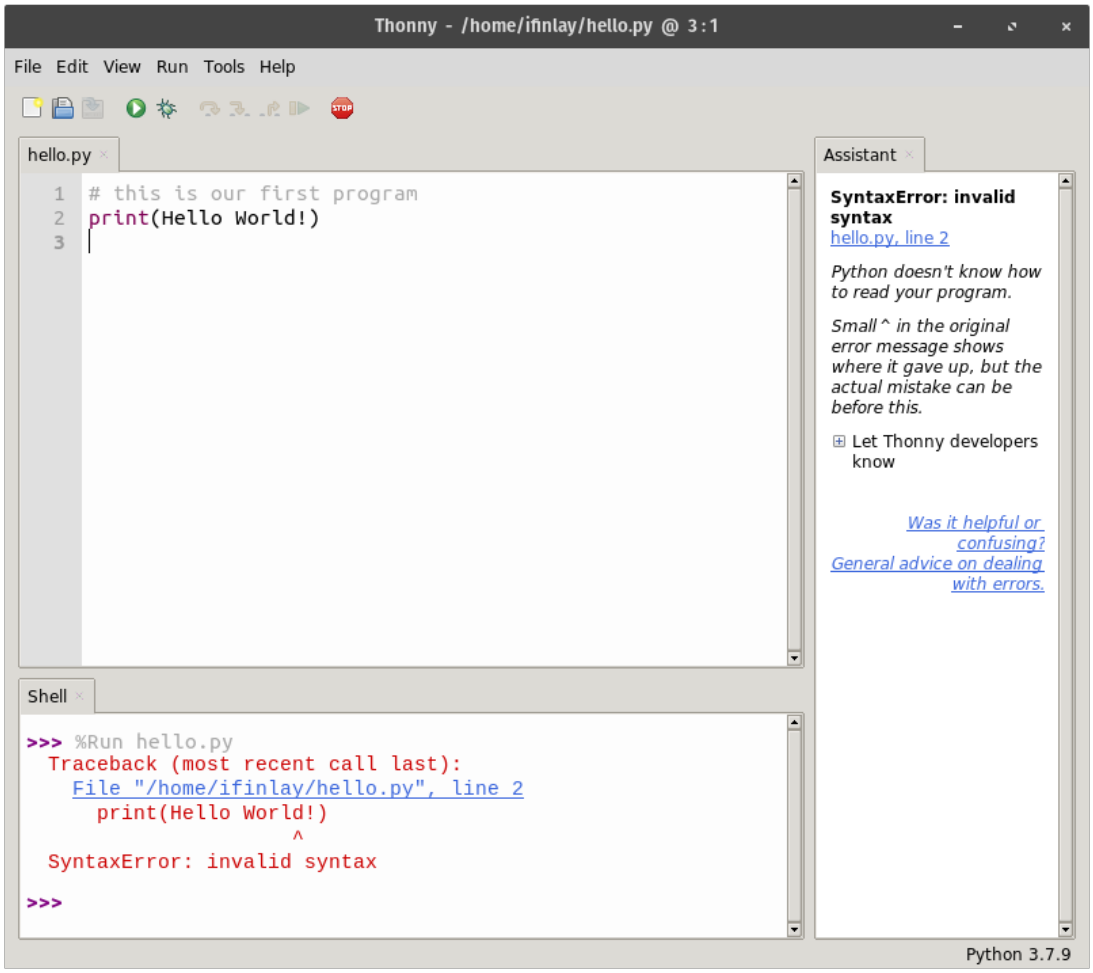

Figure 14: Missing quotation marks

Errors can be stressful as a beginning programmer. Even if your code is 99% correct, one mistake can prevent the interpreter from being able to figure it out. As you get more experience with programming, they get easier and easier to fix. In the meantime, you can always ask your friends or instructor for help.

2.6 Output

In our first Python program, we saw the `print` function which prints whatever message is put between the parenthesis. Here it is again:

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

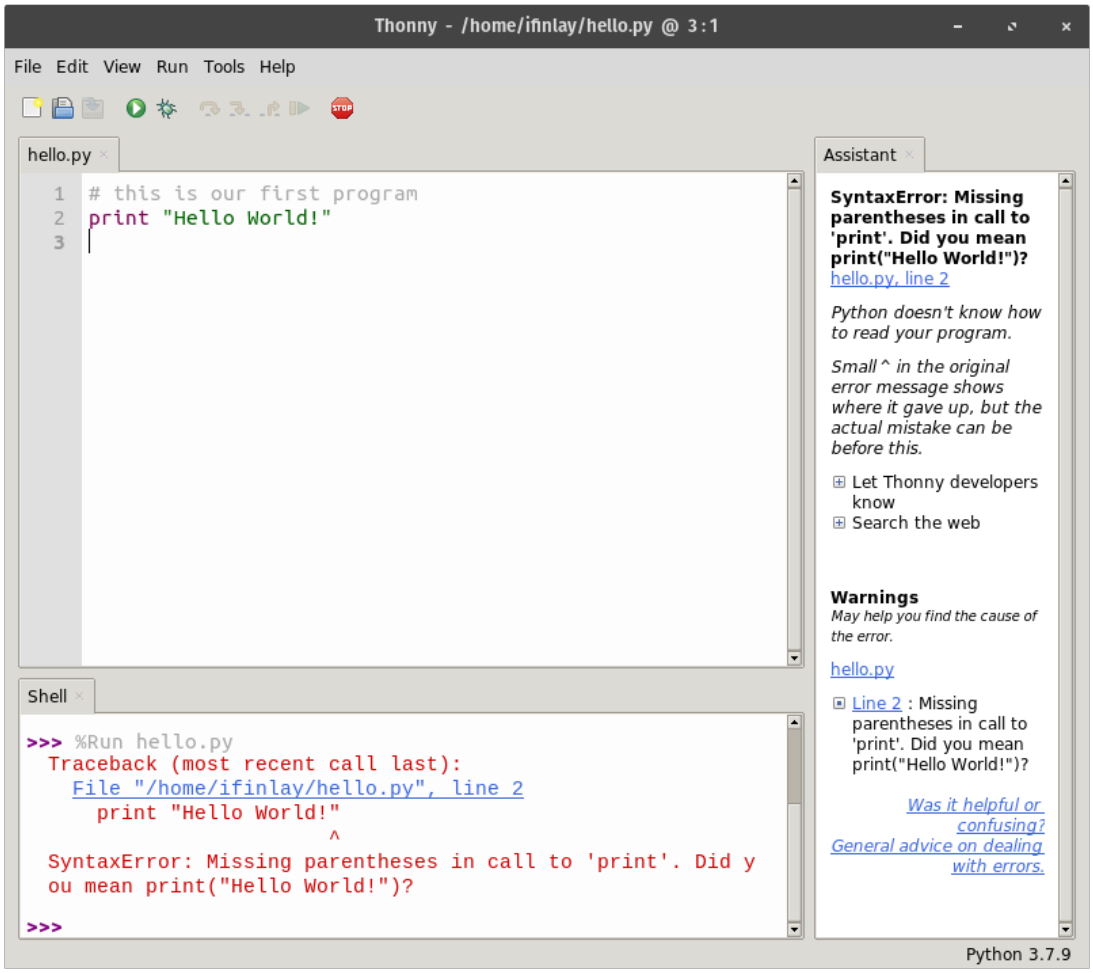

Figure 15: Missing parenthesis

`print` is the main way for doing output in Python. Of course, we can have multiple print statements in a program. The following program has three prints:

```
print("Welcome to this program!")
print("Hello World!")
print("Bye bye!")
```

When a program has more than one line like this, Python will do them one by one. This is an important point of programming. Unless we tell Python otherwise, it starts with the first instruction, then goes through them in order until it gets to the end.

So this program will print the first line, then the second, and then the third. The output of this program looks like this:

```
Welcome to this program!
Hello World!
Bye bye!
```

By the way, in this book, we will display code in the blocks with a light grey background, and what the programs output with a darker background like this.

As you can see, this program prints three lines of output, one for each of our three print statements. We will look at more things we can do with print statements in a bit, but first let's look at getting user input.

2.7 Input

We can also do input in Python, when we want to ask the user for information. Most programs take some sort of input, which allows us to control what the program is doing, or what values it is calculating with.

This can be done with the `input` function. Like `print`, `input` can take a message inside of parenthesis. In the case of input, this message is a question to give the user, called a **prompt**.

Here is an example of how `input` works[3]:

```
input("How are you feeling today? ")
```

When we run this program, it will print the prompt to the screen for us, and then wait for us to type something in. To give the program the input it's waiting for, we have to type into the shell window at the bottom of the screen. When you type something in and hit enter, it will take the input:

As you can see, Thonny colors what we are typing in blue, and what the program prints as black. Here the input we gave the program was the words "Pretty Good".

We can only type one line of text. As soon as we hit enter, Python moves on from the input instruction. In this case, there is no next instruction so the program finishes.

This program does not actually *do anything* with the input we give it. In the program above, whatever the user types in can't really affect the program at all. In order to do something with input, we must put it into a *variable*.

[3] Notice the space after question mark. That is not necessary, but it puts a space before the user can start typing, which I think looks neater.

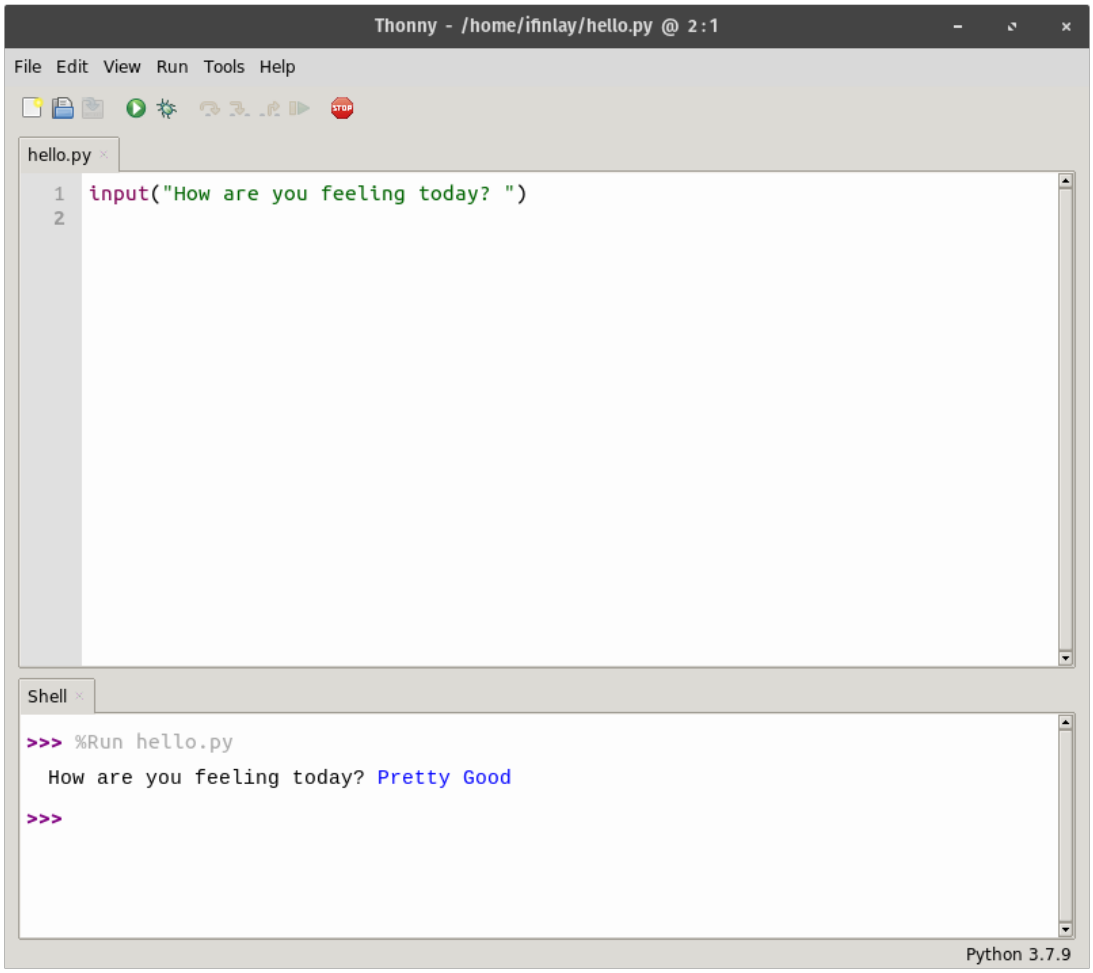

Figure 16: Giving the program input

2.8 Variables

We talked about variables briefly when we were talking about algorithms back in Chapter 1. Here we will talk about how to use them in Python.

Variables in programming are names that we associate with some piece of information. Variables let us refer back to something that was created earlier on in a program. They also let us save whatever the user inputs, so we can keep track of it.

The way that a variable is created in Python is by putting the name on the left hand side, then an equals sign, and finally the thing that you want to store in the variable. For instance, if we want to save our user's input in a variable, we could do it like this:

```
answer = input("How are you feeling today? ")
```

Now when we run this program, it will ask us the question, and wait for us to enter a response. It will then save whatever we give it into the variable called `answer`. We can now change the program so that it prints it back to us:

```
answer = input("How are you feeling today? ")
print("You said")
print(answer)
```

Here is an example of the output of this program:

```
How are you feeling today? pretty good
You said
pretty good
```

The text that we typed is in a different color so that you can see what the user types in this example. The white text is what the program itself is printing out.

There are a couple of things to note about this program. First, we have saved the input we typed into the variable called `answer`. We can then print this variable out on the third line of the program. This line of code is worth talking about:

```
print(answer)
```

Notice how this did not actually print the word "answer". When we print a variable, it doesn't print the variable's *name*, it prints the variable's *value*. Whatever got stored in the variable (which is whatever we typed in), gets printed here.

Also, notice how there are no quotation marks around "answer" in the print command. If we put quotation marks in, it *would* actually have printed out the word "answer". We have to use quotation marks to print some message out exactly, and no quotation marks when we want to get the thing stored in a variable.

There are some rules for naming our variables. The name of a variable has to be made of letters, numbers and underscore characters. They cannot begin with a number and cannot have spaces in them.

These are examples of legal variable names:

- `price`
- `price_in_dollars`
- `priceFor2`

And these are not legal:

- `full-price` (the - symbol is not allowed)

- `2_times_price` (can't start with a number)
- `price in dollars` (no spaces are allowed)

Variables also should not be named something that already means something in Python. That means that you should not name a variable `print` or `input`. There are lots of other names in Python that mean things and we will see them as we go.

Notice that Thonny colors `print` and `input` differently than other things. If the new variable you just made also shows up colored like this, then it means something special and you should pick another name!

2.9 More on Printing

In the program above, we printed our message on two different lines, which looks kind of weird. Instead, we can print it on one line, using just one print instruction. To do that, we can pass the message and the variable to print on one line, separated by a comma. That would look like this:

```
answer = input("How are you feeling today? ")
print("You said", answer)
```

When we run this program, it gives us this:

```
How are you feeling today? pretty good
You said pretty good
```

There is no limit to how many things we can print like this — we can just keep adding things and putting commas between them. Like if we want to also print "Bye!" so the user knows the program is done, we could add that in:

```
answer = input("How are you feeling today? ")
print("You said", answer, ". Bye!")
```

Now the program prints this:

```
How are you feeling today? pretty good
You said pretty good . Bye!
```

Notice that Python automatically puts a space between the things that we are printing. This is often helpful, but in this case makes the output look kind of weird since there is a space before our period. When we want to avoid this, we can also give the text `sep=""` to print. This tells Python to separate the things it's printing with nothing at all. Now the program looks like this:

Program 2.2

```
answer = input("How are you feeling today? ")
print("You said ", answer, ". Bye!", sep="")
```

And it will output the following:

```
How are you feeling today? pretty good
You said pretty good. Bye!
```

Note that we had to now put a space between "You said" and the variable because now there isn't one put in automatically. Some people aren't too bothered about details like this, but I like to get the spacing to look exactly right for the program's output.

2.10 Example: Greeting Program

Now let's create a slightly longer program which will need two variables. We'll talk about how the program will behave first, and then talk about how to write it.

We want the program to ask the user for two things:

1. Their name
2. What day of the week it is

It will then give them a personalized greeting wishing them to have a good day. For example, if we put in "Nicole" and "Thursday", then it would print this:

```
Hello Nicole!
Have a great Thursday!
```

However, if we put in "Tim" and "Monday" when the program asks our name and what day it is, then it will print this:

```
Hello Tim!
Have a great Monday!
```

This is an important point in programming — what the program does will depend on the input given to it. It means that we can't just write the program like this:

```
print("Hello Nicole!")
print("Have a great Thursday!")
```

If we did, then it works if your name is Nicole, and it happens to be Thursday, but it won't work in any other case. You also can't just replace "Nicole" and "Thursday" with your own name and day. If you do, it will work for you that day, but not in any other situation.

What we want is to have the program do the right thing in *every* situation. For that, we need to put the name and the day into variables. We will need one for each thing. One variable generally keeps track of just one piece of information.

We will start by asking the user their name and storing the result into a variable:

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

Next, we need to ask them for the other piece of information we need, the day of the week:

```
day = input("What day is it? ")
```

Now we have these two variables, which we have called `name` and `day`. The next step is to do the printing. Now we will use our variables so that whatever answers they gave to those questions will be repeated:

```
print("Hello ", name, "!", sep="")
```

This will print "Hello", followed by the user's name, and then an exclamation point, with no spaces in between (so the exclamation shows up right after their name).

Now it should greet them by name no matter what they put in. We can do the same thing to wish them a good day:

```
print("Have a great ", day, "!", sep="")
```

Below is the whole program, with a comment at the top. It's usually a good idea to put a comment at the top of your code explaining what the point of the program is.

Program 2.3

```
# this program gives the user a custom greeting
name = input("What is your name? ")
day = input("What day is it? ")

print("Hello ", name, "!", sep="")
print("Have a great ", day, "!", sep="")
```

Notice the program also has a blank line in it. Blank lines are ignored just like comments are. It's common in programs to put a blank line between different sections of code — kind of like paragraphs in a paper.

Below is an example run — though of course what it prints exactly depends on what you tell it!

```
What is your name? Mary
What day is it? Friday
Hello Mary!
Have a great Friday!
```

2.11 Comprehension Questions

1. Why do we need an interpreter for running programs written in high-level languages like Python?
2. What is an Integrated Development Environment (IDE)?
3. What is a comment in Python, and what is its purpose in a program?
4. What happens if there is an error in a program? Will the program be able to be run?
5. If a program has multiple statements, where does Python begin executing them?
6. Why do we almost always assign the result of a call to input into a variable?
7. What happens if Python sees a blank line in a program?
8. Why do you think Python doesn't allow spaces to be in variable names?
9. What's the difference between: `print(name)` and `print("name")`?

2.12 Programming Exercises

1. Change program 2.3 so that it also asks the user where they are from. Then, make it so the print also says where the user is from such as "Hello Mary from Pennsylvania" or "Hello Joe from England", before telling them to have a great day.
2. Write a program that asks the user for their name, major, and class (such as first-year, sophomore, junior, or senior). Then print out a message to them using those three pieces of information.

Chapter Summary

- An interpreter is a program that translates programs so that they can be executed by the computer. An IDE is a program that lets you write programs and passes them to the interpreter.
- The Thonny IDE can be installed on Windows, Mac, or Linux. It has a file window for writing programs, and a shell window for running commands interactively.
- Comments are lines starting with a # and are little notes that are put into programs.
- The `print` command is used to print things to the screen. You can pass multiple things to `print`, and it will print them one after the other.
- The `input` command is used to get input from the user. Input should normally be stored into a variable.
- Variables are used to keep track of information in a program. They are given a name of your choosing and can be referred to later.

Chapter 3: Types and Operations

Learning Objectives

- Understand the idea of type.
- Learn about the Python string type, and some of its operations.
- Learn about the two number types in Python: integers and floats.
- See how to get numbers as input from the user.
- Understand how to perform math operations on number in Python.

3.1 What are Types?

Just like us, programs need to keep track of information as they are solving problems or making decisions. They do this by storing data inside of variables. Last chapter, we saw how to create variables and also how to read input from the user and store it in variables. We didn't talk about this, but we did it using one type of data, called the string type.

Python actually provides multiple types of data. A **type** is something associated with a variable that determines what sorts of things you can do with it. To be able to use a variable, you need to know what type it has.

For example, you probably have a first name like "Anne", and a last name, like "Smith". In Python, these are called *strings*. You can do some things with strings like join them together, to get "Anne Smith". Or see how many letters they have. Other things don't make sense to do with strings. For example, you can't subtract strings — that just doesn't make any sense.

You also have a birth year, like 1998. You can do different things with that piece of information. For example, you *can* subtract it from other numbers. We can subtract 1998 from the current year to see how old you are.

Python has several different types for dealing with different kinds of information like this. In this chapter, we will look at strings and numbers. We will also talk about some of the operations we can do with these.

3.2 Strings

The first type that we will look into is the **string** type. A string is some text inside of a program. For instance a message, name, address, email, password, URL etc. are all textual and would be stored as strings in a Python program.

Strings can be created by enclosing text inside of double quotes:

```
name = "Anne Smith"
print(name)
```

Strings can also be enclosed inside of single quotes instead:

```
name = 'Anne Smith'
print(name)
```

There is usually no difference between the two ways of making strings. The exception is when a string *contains* a quote. For example, this program is just not going to work:

```
answer = input("Answer "yes" or "no" here ")
print(answer)
```

The problem here (which you can see from the way the code is highlighted) is that when Python sees the " at the start of "yes", it thinks that the string is over. Then it gets really confused because it can't figure out what to do with the yes. To fix this, we can use single quotes instead:

```
answer = input('Answer "yes" or "no" here ')
print(answer)
```

Now this is OK because Python knows that the string is not over until it sees the second ' symbol. This goes the other way too. For example this program has an error:

```
print('You can't enter a negative number!')
```

But this one is fine:

```
print("You can't enter a negative number!")
```

Personally, I always use the double-quotes unless the string I am writing needs to have double-quotes in it.

3.3 Joining Strings

Next we will talk about some of the operations that you can do with strings. The first one we will talk about is joining strings, which is also sometimes called concatenating[1].

This is done by putting two existing strings together with a + in between them. For example, the following program asks the user for their first and last name, and then joins them together to form a full name:

[1] Sometimes computer scientists come up with fancy words like this for very simple concepts. This is just one of many examples.

```
# read the names from the user
first = input("What is your first name? ")
last = input("What is your last name? ")

# join them together in a new variable
fullName = first + last

print(fullName)
```

Below is an example of this program being run:

```
What is your first name? Anne
What is your last name? Smith
AnneSmith
```

This program has three variables, which are all strings. The first contains the input the user gave for the first name ("Anne" in this example), the second contains the last name ("Smith"), and the third is created by joining the two names ("AnneSmith"), which is then printed out.

We can add as many strings together as we want. For example, we can make it so there is a space between the users first and last name by joining a space in between them. The program with this fix is here:

Program 3.1

```
# read the names from the user
first = input("What is your first name? ")
last = input("What is your last name? ")

# join them together in a new variable, with a space
fullName = first + " " + last

print(fullName)
```

3.4 String Length and Indexing

A string is a sequence of one or more *characters*. A character is just a single symbol of text, such as a letter, numeral, punctuation or space. For instance, the string string `"Hello World!"` has 12 characters: 10 letters, a space, and an exclamation mark.

We can get the length of a string by using the `len` function. The following program will get a string from the user and then print how long it is:

Program 3.2

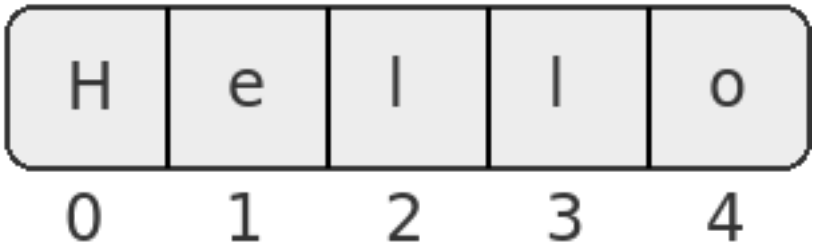

Figure 17: The indices of a string

```
# program to print the length of the user's input
response = input("Enter a string: ")
length = len(response)
print("That string has", length, "characters.")
```

Below is an example run of this program:

```
Enter a string: Hello out there!!
That string has 17 characters.
```

We can also get any individual characters out of a string if we want to. Each character in a string has an *index*. The index starts at 0 and goes up by 1 for each character. For instance, the string "Hello" has these indices:

Notice that, while there are 5 letters, the indices go from 0 to 4, and that 5 is not the index of any letter. To get just one character from a string, we put the name of the string, then the index we want inside of square brackets.

For example, we can write a program that prints out the first letter of our input like this:

Program 3.3

```
# program to print just the first letter of the user's input
response = input("Enter a string: ")
first = response[0]
print(first)
```

Here is an example of this program running:

```
Enter a string: Hello out there!!
H
```

This may not seem terribly useful, but getting the characters out of a string is sometimes necessary. For example, if a string has several things in it, like a first name and a last

name, you may need to look at each character to see where the first name ends and the last name begins.

3.5 String Methods

There are other things we can do with strings, but first we need to talk about a new term. We have seen several functions `print` and `input` which are called *functions*. There is a similar concept in Python called a **method** which is like a function, but is called on a variable. To call a method, you give the variable, then a `.` then the name of the method, and finally parenthesis.

For instance, strings have a method called `upper` which gives back a version of the string in all capital letters. We can call this method like this:

```
mesg = input("Enter a message: ")

allcaps = mesg.upper()
print(allcaps)
```

Here we get a string from the user with `input`. Then we call the `upper` method on that string. This gives us back a new string with all the lower-case letters swapped for capital ones. We then print that out. Here's an example run:

```
Enter a message: hello there.
HELLO THERE.
```

There is also a method called `lower` which gives us the all lower-case version. There are a few more helpful methods that we will see in examples later in this book. The important thing to take note of here is that these methods only make sense on strings. Numbers, or other types of data can't be lower-case or upper-case. So the type of thing you have is important.

3.6 Numbers

Many programs work with numerical data, so numbers are another important type in Python. As we will see, numbers come up a lot more in programming than you might think. Even things like games need lots of numbers to work. For example, the positions of everything on the screen are stored as numbers.

Python actually has a few different types of numbers for different situations.

The first is the integer, or `int` type. Integers are numbers that have no fractional component. For example, an int can be equal to 3, or 4, but not 3.5. Integers can be

positive or negative. Some languages have limits on how big an integer can be, but Python allows integers to be as big as they need to be[2].

We can create an integer just by putting a number into a program. We can also assign it to a variable to keep track of it. For example, to create a variable to keep track of the year someone was born, we could do so like this:

```
birthYear = 1998
```

Integers cannot have any fractional component. So if we make a number that does have a fraction in it, Python gives it a different type, called a `float`[3]:

```
size = 10.2
```

A float is a number which can have a fractional part. That fractional part might be 0 though. For example, if you make a variable equal to 3.0, then it will be stored as a float, which just happens to have the fractional part equal to 0.

You may wonder why not just have all numbers be floats, since they can have fractional parts, but also can be set to numbers without a fractional part. The reason is that floats don't store numbers exactly; they only store approximations of them. The reason for this is that some numbers, like Pi, would require infinite memory to store exactly.

We can see this effect by giving Python certain computations like the following (the »> indicates that line is typed into the shell window):

```
>>> 0.1 + 0.2
0.30000000000000004
```

Of course the answer should really be 0.3, but Python gives us a number that is almost, but not exactly, 0.3. This might seem like a flaw in Python, but it just comes from the fact that there are infinitely many real numbers between 0 and 1 — we can't store them all perfectly in a computer. That's just life. The small amount of imprecision we get with floats is normally not an issue.

As a rule of thumb, you should use integers when a number won't have a fractional component, because integers are exact. When a number might have a fraction, you have to use a float.

[2] Of course, a computer has a set amount of memory, and a big enough number could in theory need more memory than you have, but that isn't really an issue in practice.

[3] These numbers are called "floats" because they have a "floating" decimal point. That means the decimal can appear in any position, for example 10.0, 1.0, 0.1, and .01 are all valid numbers. Some languages (not Python) also have fixed point numbers, where the decimal can't move.

3.7 Number Input

To read in a float from the user, we can't just use the `input` function. This always gives us a string, even if the user types in a number. You can see this in the following shell example, where we read in a variable and then check its type. The `type` function returns the type of what you give it:

```
>>> num = input("Enter a number ")
Enter a number 7
>>> type(num)
<class 'str'>
```

This is probably confusing, because 7 obviously *is* a number. But strings can contain digits too. For instance, if you are reading in an address, it might be something like `"1301 College Avenue"`. So strings can contain numbers inside of them. This one just happens to contain only a number. Python doesn't automatically set the type based on what the user types in.

Now because it's a string, we can't really use it like a number. For example, if we try to add this variable to another number, we will get an error.

In order to actually get a number from the user, we need to convert it into either an int or float first. This can be done by putting either `int()` or `float()` around the input line, like this:

```
num1 = int(input("Enter an integer "))
num2 = float(input("Enter a float "))
```

The first example will read in the user's input and convert it to an integer. The second will convert it into a float. If the user puts in something that is not correct, the program will stop with an error. For example, if we put in "3.5" for the first line, or "banana" for either line, the program won't continue.

We'll learn how to catch these sorts of errors and handle them later on.

3.8 Doing Math

Once we have numbers inside a Python program, there are lots of things we can do with them. We saw that + can be used to join together strings. With numbers it adds them together. The following program reads two float numbers from the user and then adds them together, displaying the result:

Program 3.4

```
# this program adds two numbers given by the user
num1 = float(input("Enter a number: "))
num2 = float(input("Enter a number: "))
total = num1 + num2
print("The total is", total)
```

This program reads in the two numbers from the user, and stores them in variables called `num1` and `num2`. It then adds them together using the + operation. Because these are numbers, this results in getting the sum of the two. This is why types are important — if they were strings it would have joined them. The result of the addition is stored in the variable called "total" which is then printed to the screen.

Of course Python has other operations besides just addition. Below is a list of some operators that Python supports:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Using the + and - symbols for addition and subtraction is probably pretty familiar to you. Some of the others might take a little while to get used to.

Programming languages were made with standard American keyboards in mind, so the symbol for multiplication is the asterisk (*) instead of × or · symbols used more in math. Likewise we use the / symbol for division instead of the ÷ symbol. The symbols were picked to make them easier to type when writing code.

Notice that exponents are done with two asterisks **. A fairly common mistake is to use the caret symbol ^ instead. This could have been used, but actually means something else in Python[4].

Python follows the standard mathematical rules of precedence which can be overridden with parenthesis.

3.9 Example: Tips

As an example of working with numbers, let's write a program that can figure the amount that you should tip a server at a restaurant. When the bill comes at a restaurant, it's

[4]The ^ is used for an operation called XOR, which is used for dealing with binary numbers.

expected that diners will give a tip between 15 and 20 percent of the price of the meal. This can sometimes be tricky to figure out[5], so let's write a program to figure it out for us!

The program should start off by asking the user how much their meal cost. To do that, we will need to use `input` combined with the `float` function to read in a number that may have a fractional part:

```
cost = float(input("How much was the bill? "))
```

Next, we need to do a little math. If the user wants to give a 15% tip, then we should multiply the amount by 1.15. This will give a new number which is equal to the original amount, but is 15% more. We should save this number into a new variable:

```
total = cost * 1.15
```

One mistake beginning programmers sometimes make is to not put the answer to something like this in a variable. For instance they might write a line like this:

```
# wrong
cost * 1.15
```

This line of code doesn't really do anything. It multiplies cost by 1.15, but it doesn't save the answer anywhere. In order to make use of a result like this, we have to put it into a variable.

Now that we have the amount with a tip added in, we can print it for the user:

```
print("The amount with tip is", total)
```

This will print out the full amount with tip added in, which we have just calculated. The whole program with comments added is below:

Program 3.5

```
# read in the starting cost
cost = float(input("How much was the bill? "))

# figure out the cost with a 15% tip added
total = cost * 1.15

# print the result
print("The amount with 15% tip is", total)
```

Below is an example of running this program:

[5] Especially if the meal included a few drinks.

```
How much was the bill? 32.40
The amount with 15% tip is 37.26
```

This program would be quite helpful in figuring out how much to tip a server, but we can make it even better. What if someone wants to tip 20% instead? We could of course just change the `1.15` in the program to `1.20` instead.

But it would be even better if we asked the user how much they want to tip and then use that amount instead of a pre-determined amount.

So to start let's ask them **two** questions instead of just one:

```
# read in the starting cost and tip amount
cost = float(input("How much was the bill? "))
percentage = int(input("How much do you want to tip? "))
```

Now we have two variables. The first one, `cost`, will store the cost of the meal before tipping. The second one, `percentage`, will store the percentage the user wants to tip.

Now we need to do a little bit of math. The first thing we need to do is to divide the percentage they entered by 100. That way we can go from 15 to the .15 that we need. The word "percent" actually means divided by 100.

Next we need to add 1 to this number. That way we go from the .15 to the 1.15 we used in the first program. Except now it will work with any percentage.

Then we can multiply that number by the cost of the meal. Altogether, that gives us this line of code:

```
total = cost * (1 + percentage/100)
```

Notice that we used parenthesis to make sure to do the addition before the multiplication. The new and improved version of the program is below:

Program 3.6

```
# read in the starting cost and tip amount
cost = float(input("How much was the bill? "))
percentage = int(input("How much do you want to tip? "))

# figure out the cost with the tip added in
total = cost * (1 + percentage/100)

# print the result
print("The amount with tip is", total)
```

And here is an example of running it:

```
How much was the bill? 41.40
How much do you want to tip? 20
The amount with tip is 49.68
```

3.10 Rounding

One last thing before we close out this chapter. I sort of carefully chose the inputs to the last program so that the output looked right. If we had picked other things, it would not look as nice. For example:

```
How much was the bill? 37.21
How much do you want to tip? 17
The amount with tip is 43.5357
```

That just looks weird when we're talking about money. We always round money to the nearest cent. We would not expect to see something like this, so let's talk about how to fix it.

The best way is to round the result to 2 decimal places. This is done with the `round` function, which takes the number we want to round and how many decimal places to round to. We can try this in the shell

```
>>> round(43.5357, 2)
43.54
```

`round` always rounds to the *nearest* place. For instance `round(4.7, 0)` rounds up to 5, while `round(4.3, 0)` will round down to 4.

The program with this fix in place would look like this:

Program 3.7

```
# read in the starting cost and tip amount
cost = float(input("How much was the bill? "))
percentage = int(input("How much do you want to tip? "))

# figure out the cost with the tip added in
total = cost * (1 + percentage/100)

# round the answer to 2 decimal places
rounded = round(total, 2)
```

```
# print the result
print("The amount with tip is", rounded)
```

This is a pretty advanced program! It takes two pieces of input, uses four variables, and does some sort of tricky math. It even makes sure that the output looks like the user would expect. Even better it actually solves a real-life problem most of us can appreciate. If you followed what we've done here, then great job!

3.11 Comprehension Questions

1. Why is it important to understand the type of data stored in a variable in Python?
2. What is the difference between an integer and floating point number? When would you use one versus the other?
3. What does the + operator do with numbers and what does it do with strings?
4. What number gives us the first character in a string when used as an index?
5. What must we do with a number value read in with `input` before storing it in a variable?

3.12 Programming Exercises

1. Write a program to read in the length and width of a rectangle and print both the area and perimeter of the rectangle to the user.
2. Write a program to convert from feet to meters. There are 3.28084 feet in one meter. First read in the number of feet, do the calculation to find how many meters that is, and then print the result.
3. Write a program to read in the user's first name and last name, and print out their initials. For example, if the user puts in "Margaret" and "Jones" it should print out "M.J."
4. Write a program to print the average of 4 numbers that the user gives. You should read in the 4 numbers, compute the average, and then print the answer.
5. Imagine a snack bar that sells four different types of snacks: candy bars cost $1.00, bags of popcorn cost $.25, cans of soda cost $.75 and bottles of water cost $.50. Write a program to ask the user how many of each of these snacks they bought and tell them the total price of their order.

Chapter Summary

- A type is something associated with a variable that determines what things make sense to do with it.
- The string type is for storing text. Strings can be joined together, and you can get the individual characters out of them.

- There are two types of numbers in Python. Integers are for numbers which can't have a fractional part, and floats are numbers which can.
- The `input` function gives us strings by default. We can read in numbers by using the `int` or `float` function.
- Both types of numbers let us use math operators on them, to perform calculations.
- The `round` function is used to round numbers in Python.

Chapter 4: Making Decisions

Learning Objectives

- Learn how to use if statements to make programs that make decisions.
- Understand comparisons and how to make them.
- Meet the boolean type and the boolean operations.
- Learn how to use elif and else statements to create multi-way decisions.

4.1 Decisions in Algorithms

So far, our programs have all started on the first line and executed each line in sequence until the end of the program. However, many algorithms have the need to make decisions and do different things under different conditions.

For example, when we talked about algorithms back in chapter 1, we came up with the following for guessing a number when the player just answered "yes" or "no":

Algorithm 1

```
1. Set G to 1.
2. Ask if their number is G.
3. If it was, then we are done!
4. If it was not, then add 1 to G.
5. Go back to step 2.
```

Here the algorithm needs to make a decision based on the information it has available to it. If the guess was right, we do one thing. Otherwise we do another. Lots of algorithms have to do things like this.

Another example is adding numbers. When you are adding a column of numbers you have to make a decision on whether to carry or not. You check if the column totals more than 9. If so, you carry the tens place to the next column. If not, you don't.

In this chapter, we will talk about how to write Python programs that make decisions like this.

4.2 If Statements

The way that we can make decisions like this in Python is with an *if statement*. If statements in Python start with the word `if`, followed by some sort of condition, then a colon.

A **condition** is something that might be either true or false. For example, if we read in a number from the user, it might be bigger than 10, or it might not. To check this, we could use the following if statement:

```
if number > 10:
```

After this line, we put all of the code that we want to happen when the condition is true. For example, we can just print a message that the number is bigger than 10 with a program like this:

```
number = int(input("Enter a number: "))
if number > 10:
    print("That is bigger than 10.")
```

When Python runs this program, it will check if the condition of the number being bigger than 10 is true or not. If it is, Python will execute the print statement. If the condition is false, that print line will not be executed.

If we run this program and put in a number bigger than, we will see the message like this:

```
Enter a number: 50
That is bigger than 10.
```

However, if we put in something smaller than 10, the program will **not** run the print message, and we will see nothing:

```
Enter a number: 5
```

4.3 Indentation and Spacing

Notice that the print message in the program above was indented over to the right. Sometimes spaces don't matter in Python programs, but here these spaces before the print are actually really important. Python uses indentation to mark if lines are part of an if statement or not.

For example, we can add some more message to the program above, and whether they are indented or not will determine if they are "part of" the if statement:

```
number = int(input("Enter a number: "))
if number > 10:
    print("That is bigger than 10.")
    print("Good job putting such a big number!")
print("Bye bye!")
```

The first two messages, the ones which are indented, will only be executed when the number is bigger than 10. However, the message which says "Bye bye!" will be printed

every single time the program runs — whether the condition is met or not. That's because it's not indented, so it's not part of the if statement.

For this to work, we need to indent our code correctly so Python can tell if it's part of an if statement or not. Code should normally not be indented, but if it's part of an if, it should be. If you mix up indentation, Python will get confused. For instance, code like this will not work:

```
# error, messed up indentation!
number = int(input("Enter a number: "))
if number > 10:
    print("That is bigger than 10.")
  print("In between")
print("Bye bye!")
```

Here Python will be confused because it doesn't know what to do with the second message. Is it part of the if block or not? Python will give us an error in cases like this, and we won't be able to run the code until we fix the indentation.

4.4 Comparisons

In the example above we used the > operator to compare two numbers and decide which one was bigger. Of course we can also check if something is less than something else. Or if two things are equal or not. Python uses the following comparison operators:

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

The < and > operators will be familiar to most people from math. However Python is designed so its easy to type code on a regular keyboard. So instead of the and symbols, Python uses <=, >=, and !=.

Also notice that to check if two things are equal, Python uses **two** equal signs. That's because a single = is already used for something: assigning variables. It's a pretty common mistake to forget to put the second equal sign in.

4.5 Example: Checking Input

One common use of if statements is to check if the input users enter makes sense or not. For example, if we ask the user their age, they could enter a value that doesn't make

sense like -7, or 3000. We can use if statements to write a program which will check if the age given is too low or too high before going on to the rest of the program. We will also give them another chance to put in their age correctly.

For our purposes, let's say the age has to be between 5 and 125. Anything less than 5 we'll count as a mistake (we're assuming toddlers aren't going to use this program). Also, anything over 125 will be counted as too old (assuming the person using the program hasn't broken the lifespan record by three years).

We can do this with the following program:

Program 4.1

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

if age < 5:
    print("That is too young!  Try again.")
    age = int(input("What is your age? "))

if age > 125:
    print("That is too old!  Try again.")
    age = int(input("What is your age? "))

print("Your age is", age)
```

Because the lines of code which scold the user for entering bad data are indented under the if line, they will only happen when the conditions are true. If the user enters good data, they will never be done. For example if we put in 20, this happens:

```
What is your age? 20
Your age is 20
```

However, if we put in too low of a number, the condition in the first if statement *will* be true. When that happens, the scolding and getting a new age will happen:

```
What is your age? 2
That is too young!  Try again.
What is your age? 20
Your age is 20
```

Likewise, if we put in an age which is over 125, the second if statement will have a true condition:

```
What is your age? 750
That is too old!  Try again.
```

```
What is your age? 75
Your age is 75
```

There's one flaw in this program which is that if you give a bad answer *twice*, then it will just accept it. For example:

```
What is your age? 750
That is too old!  Try again.
What is your age? 750
Your age is 750
```

In order to fix this, we will need to learn how to repeat some code over and over again. That'll be the topic of the next chapter!

4.6 Booleans Types

Last chapter we talked about **types** in Python. So far we have learned about strings, integers and floats. Last chapter we saw how these work and how to use them. But when we use if statements, we are actually using another type.

It is called the *boolean* type. Numbers and strings can have many, many different values. In fact there are essentially infinitely many different values an integer or string can have. A boolean, on the other hand, can have only two different values: `True` or `False`. Boolean values are used to represent whether conditions are true or not. A condition like `age < 5` has a boolean value. Either the age is less than 5 (in which case it's true), or it isn't less than 5 (in which case it's false).

The somewhat unusual name "boolean", which Python shortens to "bool" comes from the English mathematician George Boole. Boole developed a form of math based on true/false values which became influential in the development of computers.

We often use booleans like the programs above. We do a comparison which has a true/false value, and put it right into the if statement. But booleans are normal types which means we can put them into variables too.

For instance, we can assign a variable to be `True` or `False`:

```
>>> done = False
>>> type(done)
<class 'bool'>
```

We could also put the result of a condition into a variable:

```
>>> valid = length >= 0
>>> valid
```

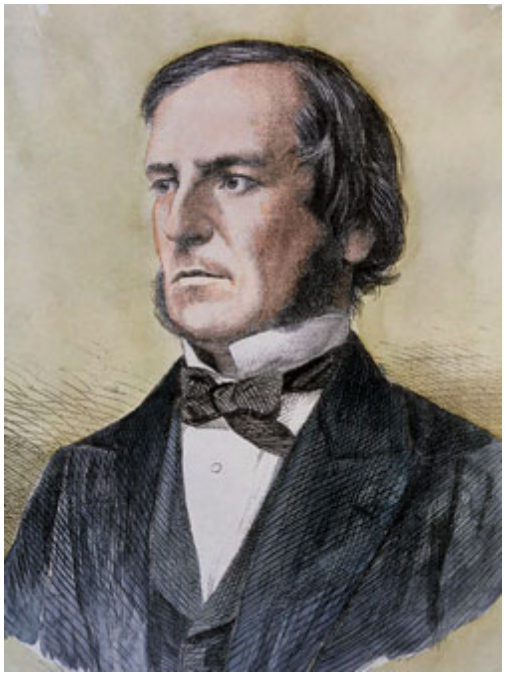

Figure 18: George Boole, circa 1860

```
True
```

Booleans are used for keeping track of information like whether certain conditions have been met, or whether events have occurred. We will find uses for using boolean variables like this next chapter.

4.7 Boolean Operations

Remember that a type determines what sorts of things you can do with something. If you've got a string, you can find the length, or join it to another string. If you've got an integer, you can add or subtract it. If you have a boolean, there are things you can do to it too.

Booleans actually have just three operations:

- not
- and
- or

The simplest of these is `not`. This operator just reverses the boolean you give it. If you have a boolean which is true, then you apply `not` to it, you get false. Here's an example:

```
>>> valid = length >= 0
>>> valid
True
>>> not valid
False
```

If `valid` is true, then `not valid` is the opposite of that: false.

The next operator is `and`. This combines two booleans and tells you if they are **both** true. For example, we could use this in an if statement for checking if our age is within the 5 through 125 range:

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

if age >= 5 and age <= 125:
    print("Your age is valid.")
```

Here we are combining up two boolean values. The first is the value we get from `age >=5`. This will either be true or false. The second is `age < 125`. When we combine them with `and`, we are going to end up with true if both parts are true. If either part (or both parts) are false, then we get false.

The last operator is `or`. Like `and`, it combines up two booleans and gives you a new one. Whereas `and` checks if **both** sides are true, `or` checks if **either** one is.

We could use this to write a program to detect when the age is *invalid*:

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

if age < 5 or age > 125:
    print("Your age is NOT valid.")
```

Notice how we use the or to combine up the two cases into one if statement. Before we had the boolean operators at our disposal, we had to do this with two separate if statements. Now, we can combine it into one line.

4.8 Two-Way Decisions

Oftentimes, we want to test a condition and, if it's true, do one thing, and if it's false, do another thing. For example, if we want to write some code which tells us how to get dressed based on the temperature, we could do something like this:

```
temp = float(input("What temperature is it out? "))
```

```
if temp < 80:
    print("You should wear pants.")

if temp >= 80:
    print("You should wear shorts.")
```

This program checks if it's less than 80 degrees and tells the user to wear pants in that case. It then checks the opposite: if it's not less than 80 out, then it must be greater than or equal to 80 degrees. Here it says to wear shorts.

It's so common to want to do this, that there is a simpler way, using an `else` statement. That would look like this:

```
temp = float(input("What temperature is it out? "))

if temp < 80:
    print("You should wear pants.")
else:
    print("You should wear shorts.")
```

Here, it will check if the temperature is less than 80. If so, it does the first print message, telling us to wear pants. If the condition was false, then it does the "else" part of the code — the print which tells us to wear shorts instead.

With an if/else statement, if the if condition is true, than the code under the if line is executed. Otherwise, the code under the else line is executed instead. It can never do *both* things.

As another example, imagine a roller coaster at a theme park which has requirements for riding it. Those who wish to ride must be at least 12 years old and also 54 inches. We can write a program to check both of these conditions and tell users if they can ride:

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

if age >= 12 and height >= 54:
    print("You can ride the roller coaster!")
else:
    print("You cannot ride the roller coaster :(")
```

4.9 Multi-Way Decisions

Sometimes we have more than one condition we want to check, and handle each one differently. We can do this using Python's `elif` statement. This stands for "else if", and allows us to chain together multiple conditions.

For example, imagine we want to expand our clothing-recommender program to include cold weather. We could do it like this:

```
temp = float(input("What temperature is it out? "))

if temp < 60:
    print("Wear a coat.")
elif temp < 80:
    print("You should wear pants.")
else:
    print("You should wear shorts.")
```

Here we recommend a coat when it's less than 60 degrees, pants when it's less than 80, and shorts otherwise.

The way this works is that Python checks each condition in order. Once it finds one condition that is true, it executes the statements under it and then goes to the end of the chain. This is important because it means **only one** of the parts will ever run. For example, in the above program, if we put in 50, it will do this:

```
What temperature is it out? 50
Wear a coat.
```

Here the first condition was true, so it tells us to bring a coat. Even though the second condition is true as well, it doesn't tell us to wear pants. As soon as one case in the chain is true, it stops.

There's no limit to how many `elif` statements we can include. If we want, we can expand it even more:

Program 4.2

```
temp = float(input("What temperature is it out? "))

if temp < 10:
    print("It's super cold, maybe stay home?")
elif temp < 30:
    print("Wear a snowsuit")
elif temp < 60:
    print("Wear a coat.")
```

```
elif temp < 80:
    print("You should wear pants.")
elif temp < 95:
    print("You should wear shorts.")
else:
    print("It's super hot, maybe stay home?")
```

It will still only give us one message, no matter what temperature we put in.

Next chapter we will continue talking about booleans and conditions. We have seen how they can be used to make decisions with if statements. Next we will see how they can be used to repeat code over again with loops.

4.10 Comprehension Questions

1. What is a condition in programming? Give an example.
2. What role does indentation play in if statements?
3. What is the difference between the = and == operators in Python?
4. What is a boolean type? What values can a boolean variable hold?
5. What does the `else` keyword do in Python?

4.11 Programming Exercises

1. The program at the end of Section 4.8 checks if the user meets the height and age requirement of a roller coaster. If they can ride it tells them so. If they can't it tells them they can't ride, but not why. Write a version of it that tells them either they can ride, they are not old enough, or they are not tall enough.
2. Write a program to tell a student their classification as a college student. Those with less than 30 credits are first-year students, those with less between 30 and 59 credits are sophomores, those with between 60 and 89 credits are juniors, and those with 90 or more credits are seniors. Ask the user how many credits they have and print what class they belong to.
3. You are helping organize a large event which people need to check into. To help make checking in faster, you are dividing people by last name. Those with a last name beginning with a letter A–F check in at line 1. Those with a first letter from G–L go to line 2. Those with M–R go to line 3, and those with a first letter from S–Z go to line 4. Write a program which will ask the user for their last name and tell them which line to go to.
4. At a certain university, students graduating in even years are said to be "Goats". Students graduating in odd years are said to be "Devils". Write a program to ask the user what year they are graduating, and tells them if they are a Devil or a Goat. Hint: remember the % operator gives us the remainder after dividing two numbers.

5. Write a program to check if a date is valid. You should ask the day and the month and then print whether it's valid based on that. For instance, 10/2 is a valid date, but 9/31 is not. You can ignore leap years for this.

6. Do exercise 5, but this time also ask the user whether or not it is a leap year, and factor that into the algorithm for determining if the date is valid or not.

7. Lots of companies pay time-and-a-half for hours worked over 40. Write a program to read in the number of hours an employee worked, and their pay rate. Then print out their pay for that period.

Chapter Summary

- Python if statements allow us to write programs which do different things according to a condition.
- The indentation of code is important as it indicates what lines we always do, and what lines are part of an if statement.
- We can use comparison operators to make decisions based on variables in our programs.
- The boolean type represents a true or false value. They can be combined with the boolean operators: and, or, and not.
- We can use the `else` statement to create a two-way decision. This makes the program do one thing when the condition is true, and a different thing when the condition is false.
- The `elif` statement can be used when we have more than two cases in our program. Python will do the first condition that is true.

Chapter 5: Going Back Again

Learning Objectives

- Learn how to write while loops to repeat code while some condition is true.
- Understand infinite loops, and know how to stop a program with an infinite loop.
- Learn how for loops work, and how to use them to loop through strings.
- Learn how to use the range function to create sequences of numbers that can be used with for loops.

5.1 Repeating Steps

Many algorithms are built on the concept of a **loop** where you repeat some steps of the algorithm multiple times. For example:

- On a shampoo bottle, it says "Lather, rinse, and repeat".
- In our "guess the number" algorithm, we have to keep on guessing until we guess the number right.
- In adding and subtracting algorithms you learned in grade school you must keep going for every digit of the number.

With loops we will be able to write programs that do these kinds of things, and repeat some part of the program over again.

As a first example, consider the "guess the number" game. We could in theory just use if statements to guess all the numbers:

```
if input("Did you guess 1? ") == "yes":
    print("Got it!")
elif input("Did you guess 2? ") == "yes":
    print("Got it!")
elif input("Did you guess 3? ") == "yes":
    print("Got it!")
elif input("Did you guess 4? ") == "yes":
    print("Got it!")
elif input("Did you guess 5? ") == "yes":
    print("Got it!")
elif input("Did you guess 6? ") == "yes":
    print("Got it!")
elif input("Did you guess 7? ") == "yes":
    print("Got it!")
elif input("Did you guess 8? ") == "yes":
    print("Got it!")
elif input("Did you guess 9? ") == "yes":
    print("Got it!")
```

```
elif input("Did you guess 10? ") == "yes":
    print("Got it!")
```

However, this is clearly kind of repetitive. It also isn't sustainable, if we wanted to write a program that would guess a number from 1 to 100, that would be a lot of typing! Instead we should use a loop.

5.2 While Loops

The simplest loop in Python is the while loop. The while loop looks a lot like an if statement. It starts with the word `while`, and then it has a condition, followed by a colon. Then there are some statements indented over. These statements are called the **loop body**.

The following is a simple example of a program with a while loop:

Program 5.1

```
number = 1
while number <= 10:
    print(number)
    number = number + 1
```

The way a while loop works is sort of similar to an if statement too. When this program is run, we start by setting the number variable to 1. Next we check if that variable is less than or equal to 10. If so we do the statements indented under the loop.

The difference is that, after we're done those statements, *we go back to the top*. It will check the condition another time. If it's *still* true, it does the loop again. It will keep on doing it over and over again until the condition is false.

In this particular case, the program will find that the condition is true the first time, so it will print the number, and then add one to it. Now the number is 2. This is still less than or equal to 10, so it does the loop body again. It keeps going on like this until the number variable is bigger than 10. So the output will be this:

```
1
2
3
4
5
6
7
8
9
```

```
10
```

We can now use a while loop to implement the simple version of the "guess a number" game as follows:

Program 5.2

```
# start by asking for 1
number = 1
answer = input("Did you guess 1?")

# keep doing the loop until the answer is yes
while answer != "yes":
    number = number + 1
    answer = input("Did you guess " + str(number) + "?")

# when we are done the loop it means we got it
print("Got it!")
```

This program will keep looping until the user enters "yes". At that point, the condition becomes false and so the loop exits.

There's quite a bit going on in this code, so let's go through it.

- First, we need to ask them the first question: whether their number was 1 or not. We do this so that the variable answer is defined for the start of the loop.
- We then have the loop condition which checks if their answer was not "yes". If it wasn't we go into the loop. If it was, then we got it on the first try, and skip over the whole loop.
- Inside the loop body, we add 1 to the number variable. This is how we change the thing we guess from 1, to 2, to 3, etc.
- We also ask them if the latest number is their guess or not. Doing so is a little tricky. Unlike `print`, the `input` function doesn't let us pass it numbers, only a string. So we have to convert number to a string with the `str` function, and join it to the rest of the question.
- The code after the while loop, which prints "Got it!", will only happen once the condition becomes false. For this program that means that answer *was* yes.

5.3 Example: Checking Input

We've talked about how to use if statements to check if user input is valid. For instance, this program will check if the user enters a negative number for their age:

```
# program that checks for bad data once
age = int(input("How old are you? "))

if age < 0:
    print("Hey, your age can't be negative!")
    age = int(input("How old are you for real? "))

print("You are", age, "years old.")
```

However, if someone puts in a negative number twice in a row, then the bad input will still get through. We could of course repeat the check again to make *really* sure:

```
# (silly) program that checks for bad data twice
age = int(input("How old are you? "))

if age < 0:
    print("Hey, your age can't be negative!")
    age = int(input("How old are you for real? "))

if age < 0:
    print("Hey, your age STILL can't be negative!")
    age = int(input("How old are you for real? "))

print("You are", age, "years old.")
```

Of course now, they can put in a negative number *three* times. Clearly this is not a great way of approaching this. A better way would be to use a loop. In this case, we will want to keep on asking them, over and over again, until they eventually put in valid data. Maybe this is the first try, or maybe it's the hundredth.

We can do it by simply replacing `if` with `while`. Now, the program will *keep* asking the user for data until it is greater than or equal to 0. Now the program looks like this:

Program 5.3

```
# program that checks for bad data over and over
age = int(input("How old are you? "))

while age < 0:
    print("Hey, your age can't be negative!")
    age = int(input("How old are you for real? "))

print("You are", age, "years old.")
```

Below is an example run of this program, where the user messes up by entering a negative age a few times in a row:

```
How old are you? -5
Hey, your age can't be negative!
How old are you for real? -2
Hey, your age can't be negative!
How old are you for real? -7
Hey, your age can't be negative!
How old are you for real? -1000
Hey, your age can't be negative!
How old are you for real? 27
You are 27 years old.
```

5.4 Infinite Loops

One danger when creating loops is that the condition might *never* become true. For example, there's a mistake in the program below which causes this:

```
number = 1

while number < 10:
    print(number)
    numer = number + 1

print("All done!")
```

Here we mistyped "number" as "numer". So when we run the addition, it doesn't change number to be bigger. Instead it makes a *new* variable. Because of our typo, the variable `number` never reaches 10, so the condition stays false forever. This is called an *infinite loop* and is a common programming mistake.

If you run this program, it will never stop running. It will just continue on forever. Or until you stop it, which is probably what you will want to do. You can do this in Thonny by hitting the stop button (), or by choosing "Stop/Restart backend" from the "Run" menu.

5.5 Example: Running Total

Let's write a program that will compute a running total of numbers. The way this will work is that you will put in numbers to the program, and it will add them all together, and show you the sum after each one. You can then stop the program by entering a 0 (there's no reason to add 0 normally since it won't change anything).

Below is an example of a run of this program, so you can see how it should work before we dive into some code:

```
What's the first number? 7
Running total is 7
Next: 12
Running total is 19
Next: -5
Running total is 14
Next:  2
Running total is 16
Next:  0
The total is 16
```

The code which solves this problem is given below:

Program 5.4

```
# get the first number
number = int(input("What's the first number? "))
total = number

# keep going until they enter 0
while number != 0:
    print("Running total is", total)
    number = int(input("Next: "))
    total = total + number

# print the final result
print("The total is", total)
```

We start by getting the first number from the user. The `number` variable is used to store the thing they just entered. The `total` variable is used to keep track of the running sum. It starts as the same as the number first entered.

The condition for this while loop is `number != 0`. So we will keep going as long as the number they entered wasn't 0.

Inside the loop, we do a few things. First we print out the total so the user can see it change for each number they enter. Then we get the next number. Lastly we add it to the total variable.

After the loop, we just print out the total sum.

This kind of program would be *impossible* to write without a loop of some kind. Even if we wanted to copy and paste a bunch of code, we couldn't because we don't know ahead of time how many numbers the user will want to add.

5.6 For Loops

There is another type of loop in Python called a **for loop**. A for loop is similar to a while loop in that it lets you do some piece of code over and over again. But it is different in that it loops through every element in a *sequence* of some kind.

The only type of sequence we have seen so far is a string. A string is a sequence of characters (which could be letters, digits, punctuation, etc.). Below is a for loop that just prints each character one by one:

Program 5.5

```
name = input("What's your name? ")

for letter in name:
    print(letter)
```

The for loop looks a bit different from the while loop. Instead of the condition, we have the word `for` followed by a variable name. The variable name above is `letter`. Then we have the word `in`, followed by the sequence we are using. In this case, that's the string `name`.

When you run a for loop, it makes the variable (`letter` in this case) equal to each thing in the sequence one-by-one. It then runs the loop body on it. So if we enter, let's say "Amy" as the name, then it will first set `letter` to "A". It will then run the loop body on "A". Next it will run the loop body again, but with `letter` equal to "m". Lastly it will run the loop body with `letter` as "y". Then it will stop.

The result of running this program can be seen below:

```
What's your name? Amy
A
m
y
```

The result of this program is that it prints the name out vertically.

Every for loop could be replaced by a while loop that does the same thing. In the example above, we could have written a while loop to count up to the length of the string, and used indices to get the letters out.

But for loops have a few benefits when looping through a sequence:

- It's much harder to accidentally make an infinite for loop.
- For loops are a little easier to read. They make our intention of looping through a sequence more obvious.

5.7 The range Function

For loops can be used to loop through any *sequence*. A string is just one type of sequence. There are several others in Python. The next one that we will look at is a *range*.

The `range` function is used to create a sequence of numbers. For example, we could use `range` to make a sequence of numbers from 1 through 10. Or a sequence of numbers from 25 down to 5. We can then combine `range` with a for loop to write code that does something for each number in the sequence.

If we pass `range` 1 number, it will give us a sequence from 0 up to (but not including) that number. For example, if we pass 10:

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

Then range will give us a sequence of numbers going from 0 through 9. Instead of starting at 1, range starts at 0 — just like string indices[1]. If we run this program, we will get:

```
0
1
2
3
4
5
6
7
8
9
```

Just like the for loop with a string, this for loop sets our variable (called `i`) to each thing in the sequence one by one. It then runs the loop body once for each value.

If we don't want to start on 0, we can also pass range a starting point. To do that, we have to pass two numbers. The first is the start and the second is 1 past the ending point.

For example, if we want to make a range of numbers from 5 through 10, including both end points, we could do it like this:

[1] It might seem weird at first, but we count starting at 0 in computer science. The reason has to do with the fact that memory addresses are calculated based on an offset past the start of something. So the first symbol in a string is stored 0 bytes past the beginning.

```
for i in range(5, 11):
    print(i)
```

This program will print the numbers from 5 to 10:

```
5
6
7
8
9
10
```

Lastly, we can pass *three* numbers into `range`. The first two are the same as before. The last number we pass in will be used as *step* between each. For example, if we want to go through even numbers from 2 through 10, we could pass a step of 2:

```
for i in range(2, 11, 2):
    print(i)
```

This gives us the following:

```
2
4
6
8
10
```

The step is just the amount that you add to each number to go on to the next one.

We could also pass a negative number for the step to go *backwards*:

```
for i in range(10, 0, -1):
    print(i)
```

This gives us the following:

```
10
9
8
7
6
5
4
```

```
3
2
1
```

5.8 Example: Temperature Table

As an example of a for loop with a range, let's write a program which gives us a table of Celsius temperatures with their equivalent Fahrenheit temperatures. Rather than read in one, and print out the other, we will just print a whole table. That way the user can see how the two relate.

One part of this is converting one temperature from Fahrenheit to Celsius. We can do this by subtracting 32 from the Fahrenheit temperature and then multiplying by $5/9$.

The next part is doing this for a bunch of temperatures. To be helpful, let's make the range of temperatures start at the lowest Fahrenheit temperature it's likely to be. Here in Virginia, it's rare that it gets below 0°, or above 100° Fahrenheit. We can therefore use 0 as the starting point to range, and 101 as the ending point (remember it has to be just *past* the value we want to end at.

It will also be nicer if we don't print *every* temperature from 0° to 100°. 100 lines of output will probably be too much. Instead, we will go in increments of 5°. To do this, we can just pass 5 for the last thing to `range`.

The program, then, is given below:

Program 5.6

```
# loop from 0 to 100, going 5 at a time
for far in range(0, 101, 5):
    # do the conversion, and round it
    cel = (far - 32) * 5/9
    cel = round(cel, 2)

    # print one line of the table
    print(far, "degrees F =", cel, "degrees C")
```

Here our loop variable is called `far` (short for Fahrenheit). It is given all the values in the sequence of numbers we make with range. First it's 0, then 5, then 10, all the way to 100.

For each time through the loop, we do all of the commands on the loop body. This converts to a Celsius temperature, rounds it, and prints out the two temperatures that are equivalent.

The output from this program is given below:

```
0 degrees F = -17.78 degrees C
5 degrees F = -15.0 degrees C
10 degrees F = -12.22 degrees C
15 degrees F = -9.44 degrees C
20 degrees F = -6.67 degrees C
25 degrees F = -3.89 degrees C
30 degrees F = -1.11 degrees C
35 degrees F = 1.67 degrees C
40 degrees F = 4.44 degrees C
45 degrees F = 7.22 degrees C
50 degrees F = 10.0 degrees C
55 degrees F = 12.78 degrees C
60 degrees F = 15.56 degrees C
65 degrees F = 18.33 degrees C
70 degrees F = 21.11 degrees C
75 degrees F = 23.89 degrees C
80 degrees F = 26.67 degrees C
85 degrees F = 29.44 degrees C
90 degrees F = 32.22 degrees C
95 degrees F = 35.0 degrees C
100 degrees F = 37.78 degrees C
```

There are other types of sequences that for loops work with. We will see a few more as we go. We will also see lots more examples of solving problems with loops. Almost all algorithms use looping in some fashion.

5.9 Comprehension Questions

1. What is the purpose of writing loops in a program?
2. How is a while loop different from an if statement in Python?
3. What is an infinite loop? Are they usually a good thing?
4. How are for loops different than while loops?
5. What is the purpose of the range function?
6. When would you choose to use a for loop vs. a while loop?

5.10 Programming Exercises

1. Write a program that reads in a number from the user, and a message. The program should then print the message out "number" many times. For example, if they enter 3 and "Hello", then the program should print "Hello" three times.

2. Write a program which will let the user print the length of strings, for as many as they want. It should ask them to enter a string and then print the length of it. It

should keep doing this until they enter an empty string (with a length of 0). Then the program should end.

3. Your younger sibling is working on learning their times tables. They want help with specific numbers, so have asked you to write a program to print one column of a times table. For example, if they want help with 7, your program would print:

```
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
...
7 * 10 = 70
```

   Your program should read in the number they want the times table column of (7 of this case). It should then output this column showing the products of the number with 1 through 10.

4. Write a sum calculator. Begin by asking the user how many numbers they wish to enter. Then write a loop that executes that many times. Each time, read in a number from the user and add it to a running tally. At the end, print out the sum of all the numbers they entered.

5. Ask the user to enter a number and then tell them how many digits are in that number. This can be done by dividing the number by 10 until the number is less than 1, counting how many divisions you did to make that happen.

6. The Fibonacci series is a sequence of numbers. The first two numbers in the series are both 1. Each number after that is the sum of the previous two numbers. So the series starts as "1, 1, 2, 3, 5, 8, 13, 21...". Write a program to print the first 50 numbers in the series.

Chapter Summary

- Many algorithms use looping, which is repeating some steps of the algorithm multiple times.
- While loops are based on a condition. They check if the condition is true. If it is, they run the code in the loop body. They then check the condition again. They will keep doing this until the condition is false.
- It is possible to make a loop where the condition will never become true. This is called an infinite loop, and usually is a mistake.
- A for loop is used to do something with every part of a sequence.
- A string is a sequence. If you want to do something with every character in a string, a for loop works best.
- We can also create sequences of numbers with the `range` function. `range` can be given a starting point, an ending point, and the amount to increase or decrease by.

Chapter 6: Algorithms

Learning Objectives

- Learn about nested control structures.
- Be introduced to techniques for solving problems with algorithms.
- Learn about pseudocode and flowcharts.
- See more examples of problems and algorithms.

6.1 Overview

In the last couple of chapters we looked at if and else statements, and looping. These statements are examples of **control flow** statements, because they control the flow of the program — that is they change the order the other statements run in.

In this chapter we will look at some more examples of these control flow statements, and learn some new things we can do with them. In particular, we will talk about *nesting* if and else statements and loops together. Doing this will allow us to write more complex problems and solve some problems we couldn't otherwise solve.

With these tools, we can also start to tackle some more interesting problems. So this chapter will also talk about some techniques for solving problems. Lastly we will look at a handful of example problems, and talk about how we can go about breaking them down before giving code to solve them.

6.2 Nesting Control Statements

So far our programs have only used an if and else statement, or a loop at one time. But to make more complex programs, we can start to combine them up together. We can do that by *nesting* them together. For example, we can put an if/else statement inside of a loop. Or a loop inside of an if statement. Or even a loop inside of another loop.

As a first example, let's look at a program to read numbers from the user and tell the user if each number is even or odd. The user would be able to enter as many numbers as they want, and enter 0 to quit.

To do this, we will need a while loop to keep reading in the numbers. We will also need an if/else statement to check if the number is even or not. The if/else can't be *after* the loop, because it needs to check every single number read in. Instead it has to be *inside* the loop.

The code for doing this is below:

```
# read the first number
num = int(input("Enter a number: "))
```

```
# keep going while it's not 0
while num != 0:
    # do the even/odd check
    if num % 2 == 0:
        print("Even")
    else:
        print("Odd")

    # get the next number
    num = int(input("Enter next number: "))
```

The `%` operator is new, so we need to explain that first. What this does is checks the remainder of a division. For instance if we divide 13 by 5, we get 2 with a remainder of 3. So `13 % 5` is equal to 3. If the remainder when dividing by 2 is 0, it means there is no remainder, so the number is even.

Here the if/else is *nested* inside of the loop. Every time the loop is done, we check the if condition and do either the if part or the else part. Remember that the indentation is what tells us what part of the code is part of the loop. Since the if and else statements are indented, they are part of the loop. That's what *nesting* means in computer science — that something is part of something else.

There is no restriction on nesting like this. We could instead put a loop inside of an if statement, or a loop inside of another loop. We could even put a loop inside of an if statement which is part of another loop. We will see more examples of nested control structures as we go along.

6.3 Example: Guess the Number

Now that we know how to nest if statements with loops, we can finally tackle a Python version of the "Guess the Number" algorithm we looked at way back in Chapter 1. The algorithm is given again in pseudocode:

```
1. Set min to 1.
2. Set max to 100.
3. Set G to (max + min) ÷ 2 (rounding down if needed).
4. Ask if their number is G.
5. If it is, then we are done!
6. If the guess was too high, set max to (G - 1).
7. If the guess was too low, set min to (G + 1).
8. Go back to step 3.
```

Notice that we have a loop (steps 3 through 8) with an if/elif/else statement inside of it (steps 5 through 7). So our program will need to nest these statements too.

The Python code to solve this problem is below:

```
# set initial values for our variables
min = 1
max = 100
done = False

while not done:
    # ask them if this is their number or not
    G = int((max + min) / 2)
    answer = input("Is your number " + str(G) + "? ")

    # respond to their answer
    if answer == "yes":
        done = True
    elif answer == "too low":
        min = G + 1
    elif answer == "too high":
        max = G - 1
    else:
        print("Answers are 'yes', 'too low', or 'too high'.")

# when we exit the loop, we have guessed the number
print("Got it!")
```

There are some things to point out about this program. First, we are using a boolean variable, called `done` to keep track of whether or not to exit the loop. The variable starts off as False, and we set it to True when we find the user's number. Because our condition tells us to keep looping while we are not done, the loop will keep going until we guess right.

Another thing to point out is that we solve the problem of rounding down by calling the `int` function. We have used this function to change a string (like "12") into an integer (like 12). It can also be used to change a float number (like 12.5) into an integer (like 12). The `round` function could be used to round to the *nearest* whole number, but not to always round down like we want here.

We are also calling the `str` function to convert the variable G into a string. The reason for this is that `input` doesn't allow us to pass multiple things to be printed like `print` does. We have to pass 1 string. We do this by joining the different parts of our questions, but we can only join strings with the + operator — not integers. So we use `str` to convert from a number (like 12) into a string (like "12").

And crucially the if statement chain for testing if we got our guess right or not is nested inside the while loop. You can tell this because it is indented over. We need to do this because we need to do the test for *every* guess we make, not just one.

Here is an example run of this program:

```
Is your number 50? too high
Is your number 25? too low
Is your number 37? too high
Is your number 31? too low
Is your number 34? yes
Got it!
```

6.4 Example: Password Strength

Let's look at another example now. Many websites require user passwords to meet certain standards. For example, our university has the following requirements for passwords:

- Be at least 8 characters long
- Include at least one upper-case letter
- Include at least one lower-case letter
- Include at least one digit

We already know how to check if the string is long enough with the `len` function. The other ones are a bit trickier though. The approach we will take is to loop through the entire password one character at a time. For each character, we will check if it is one of the three things we need to look for.

In order to do this, we will need a couple new string methods. These are `isupper`, `islower`, and `isdigit`. These each return true if the string contains only upper-case letters, lower-case letters or digits respectively. We'll call these on each letter to see what sort of character it is.

In doing this, we also need to keep track of whether *any* of the symbols in the password are in one of these three categories or not. We will do this by having a boolean variable for each category. For instance, we can have a variable called `upper` which starts at false. Then, when we see an upper-case character, we will set it to true.

The code for solving this problem is given below:

```
# read the password from the user
password = input("Enter password: ")

#  check the length first
if len(password) < 8:
```

```
    print("Password must be 8 characters or more.")
else:
    # length OK, now check contents
    upper = False
    lower = False
    digit = False

    # check each chatacter to see if it's in these categories
    for char in password:
        if char.isupper():
            upper = True
        elif char.islower():
            lower = True
        elif char.isdigit():
            digit = True

    # now check if all three categories are met or not
    if not upper:
        print("Password must contain an upper-case letter.")
    elif not lower:
        print("Password must contain a lower-case letter.")
    elif not digit:
        print("Password must contain a digit.")
    else:
        print("Password is accepted!")
```

This is the longest program we have seen so far! It also has a few nested statements, so let's go through it carefully so we can be sure to understand it. The program starts by reading in your password. Next it does the check to see if it is at least 8 characters. If not, it gives you a message saying it's not long enough.

The rest of the program is in the `else` statement. That means the rest of the checks only happen when the password *is* long enough. We start by making one variable for each category we have to check, and setting them all to false. We will assume we don't have any of these until we see one.

Next, we loop through every character in the string with a for loop. For each character, we check if it's a upper-case letter, lower-case letter or digit. In each case we set the corresponding variable to true.

When we are done going through the loop, we will have checked every single character in the password. If any of our three boolean variables is still false, that means that type of thing must not have been present in the password.

We then finish up by checking those three variables in an if/elif/else statement. If any one of them was false, we scold the user saying their password needed one of those

characters. In the else clause, we know that their password met all the criteria, so we declare that it is accepted.

6.5 Example: Times Tables

Now we will look at an example of nested loops. That would be one loop nested inside of another loop. An example of a problem we could solve with nested loops is the problem of printing out a times table. A times table is a table which shows what one number multiplied by another is. You probably had to memorize this table in grade school.

Let's say we want to print a 10 by 10 times table like the following:

```
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

To do this, we will need two nested loops. The first loop will loop through each of the ten *rows* of the table. Then the inner loop will go through each of the ten *columns* of the table. Then in that loop we print one number. The basic algorithm would look like this:

```
for each row:
    for each column:
        print this row number times this column number
```

Because the column loop is nested inside the row loop, it will be done in its entirety for every row. With ten rows and ten columns, we will get a total of 100 numbers printed out.

The Python code for this program is below:

```
# read the size of the table
size = int(input("What size table would you like? "))

# loop through each row
for row in range(1, size + 1):
    # loop through each column
    for col in range(1, size + 1):
```

```
        # print one 'cell' of the table
        print(row * col, end="\t")
    # go to the next line after this row is done
    print()
```

There are a couple things to notice here. First, we have our nested loops. Each loop goes through the numbers 1 to 10. We also have to give them a different variable (`row` and `col` in this case). We couldn't use "i" for both because then one would overwrite the other.

Also we pass `size + 1` as the end point to the `range` function. That allows us to loop more or less depending on how big of a table the user requested.

The print that outputs the number uses the two variables `row` and `col` multiplied together. We don't want each number to be on a line all by itself, so we have to tell Python not to end with a new line. Instead we pass `end="\t"`. The `"\t"` is a *tab character*. By putting a tab after each character, we make sure the numbers line up nicely in columns. We could have used a space and it would have worked, just looked a little messier.

Then we have the funky looking line that calls `print()` with nothing at all between the parenthesis. The purpose of this is to go down to the next line. Each time we print a number, we end with a tab. Then after one row is done (after the inner for loop has finished), we need to go down to the next line so the next row has room. That's what that second print accomplishes.

Nested loops can be tricky because you have to keep track of where you are in both at the same time.

6.6 Breaking Down Problems

As we mentioned in Chapter 1, computer science is not really the study of computers — it is the study of algorithms. The main thing that computer scientists do is come up with algorithms to solve various problems. Usually they write the algorithms in a programming language like Python, but coming up with the algorithm in the first place is usually the hard part.

Like any intellectual skill, learning to develop algorithms is something that takes a bunch of time and practice. That said, there are some techniques for breaking down a problem so that you can go about solving it. This section will give some advice for doing this.

The following are steps that I think are good to go through when tackling a new problem:

1. **Identify the inputs**

   Most problems have some kind of information which is given to you as input. Your algorithm is generally going to have to use this input in some way, so listing out what inputs you will need is a good starting point.

2. **Identify the outputs**

   For most problems, there is also some kind of solution that you are looking for. If we don't have the output we are looking for firmly in mind, it will be hard to hit upon the right algorithm. Listing the outputs our algorithm is expected to produce will make sure we know our goal.

3. **Solve a few examples by hand**

   Next you should solve a few examples of the problem just by hand. If you can't solve the problem with example inputs, then you have no hope of coming up with an algorithm that can solve it in general. When doing this, it's good to try examples of the different situations that could arise.

   By working through a few concrete examples, you are doing two things. First you are making sure that you really understand the problem before diving into an algorithm. Secondly, you are going through the steps that your algorithm will need to go through which will give you insight into writing it.

4. **Write the basic steps**

   Based on what you learned working the example problems, you can now sketch out the algorithm. But rather than jump right into Python code, it's often helpful to start with righting down the basic steps in English first.

   The main reason for this is because it's easier to focus on just the algorithm and not the details of a programming language. For example, you don't need to worry about deciding whether things should be int or float, or making sure parenthesis line up right.

   This English like description of an algorithm is sometimes called "pseudocode" because it is sort of like computer code, but not really in any actual language. The main benefit of this is that it lets us focus on the algorithm without being distracted by details of Python syntax.

5. **Test the steps**

   Another benefit of using pseudocode is that we can now test out the algorithm in its simple English-like steps before coding it. We should follow the steps a couple of times to make sure that it gets the right answers. If we made any mistakes, or if it is wrong in some cases, we can fix it before we spend the time making it a program. Algorithms in pseudocode can be easier to fix than programs in a full programming language.

6. **Write the actual code**

   Once we are confident our basic algorithm is working, we can start putting it into actual code in a programming language (we'll use Python of course). Now instead of focusing on the problem itself, we will focus on the language's syntax and finding all the functions we need.

7. **Test the code**

   Now we can test our code out to see if it gives us the answers that we expect. We should be able to plug in the inputs for the examples we solved by hand and get the right answers.

   Another benefit of splitting our problem-solving steps into two parts is that we should not have to fix problems with the actual algorithm any more (those should have been caught in step 5). Any problems now should be with translating our pseudocode steps into actual Python code.

Experienced programmers will often skip steps 4 and 5, and jump right into writing code when they have an idea of how to solve a problem. The main reason for this is that they are so comfortable with the programming language they are using. An experienced programmer can translate steps into code as they go. For a beginner this is harder. And even experienced programmers will often fall back to writing the steps in English first when they run into an especially tricky problem[1].

6.7 Breaking Down Problems Example

As an example of applying these steps, let's solve the problem of figuring out how much somebody is paid given their hourly rate and the number of hours they worked. If the hours worked are over 40, then the person is paid "time and a half" for their overtime.

It's worth pointing out that the problem of solving one specific case of this problem (like if you worked 30 hours and make 12 dollars per hour) is just a math problem. It becomes a computer science problem when we want to solve it *in general*. With the right algorithm, we could solve any case of this problem at all — no matter what the inputs are, our algorithm will give us the right answer.

Let's go through the steps outlined above:

1. **Identify the inputs**

   For this problem we have two pieces of input: the hourly rate and the number of hours worked.

2. **Identify the outputs**

   In this case, we only want one output, which is the amount of money earned.

3. **Solve a few examples by hand**

   For this problem, we should solve an example where the person doesn't get overtime and one where they do. Let's start with the case where they don't get overtime. Say

[1] When I personally am writing programs, I will often start by writing my "basic steps" as comments in my program files. Then I read through and make sure the steps make sense before jumping into the code. Then, I add the code, but leave the comments in to explain how the code is working. I find this method works well.

they work for 30 hours, and make 12 dollars per hour. In this case, we should just multiply the two numbers together to get $30 \times 12 = 360$.

In the case where the employee *does* get overtime, we have to do something extra. Let's say they work 45 hours and make 10 dollars per hour. Now we need to give them 10 dollars for each of the 40 regular hours they are working. We also need to pay them extra for the hours they work over 40. In this case that's 5 hours at 15 dollars per hour. So in total we have $40 \times 10 + 5 \times 15 = 475$.

4. **Write the basic steps**

   We should use the steps we took solving the problem above to guide us in writing them out in pseudocode. We will have identified the two main cases in this problem, and might come up with something as straightforward as the following:

```
1. Read in the hours
2. Read in the wage
3. If they worked 40 hours or less, pay will be (hours * wage)
4. Otherwise, pay will be:
     40 * wage +  (hours - 40) * (wage * 1.5)
5. Print out the pay
```

5. **Test the steps**

   In this case, we can run through these steps with a couple of example inputs to make sure they work. We can even use the ones we worked through by hand to make sure they get the same answer we arrived at.

   If something does go wrong here, we should modify the algorithm at this point before moving on.

6. **Write the actual code**

   Now that we have an algorithm that we are pretty confident with, we can go through and translate it into Python code. In this case, we can write something like this:

```
hours = float(input("How many hours did you work? "))
wage = float(input("What is your hourly wage? "))

if hours <= 40:
    pay = hours * wage
else:
    pay = 40 * wage +  (hours - 40) * (wage * 1.5)

print("Your pay will be", pay)
```

7. **Test the code**

   Finally we can test this code to make sure the results it gives matches what we expect. We should be able to run this code and give it the inputs we tested in step 3 to make sure it gets the same result that we got.

Because this problem isn't *terribly* difficult, many of you could probably have jumped straight into code. The point of this section is to give an example of solving problems this way. These steps will be helpful to at least think about when you encounter trickier problems.

6.8 Flowcharts

Another tool that computer scientists use to solve problems before jumping into code is the flowchart. A flowchart shows the steps of an algorithm, just like pseudocode does. A flowchart however makes the decisions in the algorithm really clear by having arrows that show what steps happen when.

Below is an example of a flowchart for the overtime program:

The flowchart shows the algorithm in a slightly more graphical way. Each box in the chart displays one step of the algorithm. The nice thing about a flowchart is that it makes the control flow statements really clear. In this case, you can see the decision (which is typically shown with a diamond shape in a flowchart), with branches for the "yes" and "no" cases.

Flowcharts also make loops easy to spot. We can make a flowchart for the problem of reading a valid age from the user. The steps in the following algorithm keep asking the user for an age until they put in something that's not negative:

You can see the loop in this algorithm because of the arrow that's pointing back to a previous step (that's the arrow pointing from the "Read age again" step back around to the decision). Making flowcharts can be helpful as you are working on an algorithm, as they can help you understand the order that the steps need to be done in.

6.9 Comprehension Questions

1. Explain what is meant by "nesting" control structures such as if statements and loops.
2. How can you determine whether a line of code is part of a loop, or comes after the loop?
3. If we have a loop that executes 10 times, then nested within that is a loop that executes 3 times, how many times does a print statement within the inner loop execute?
4. What is pseudocode and how does it aid in developing algorithms?
5. How do flowcharts help in understanding and solving programming problems?

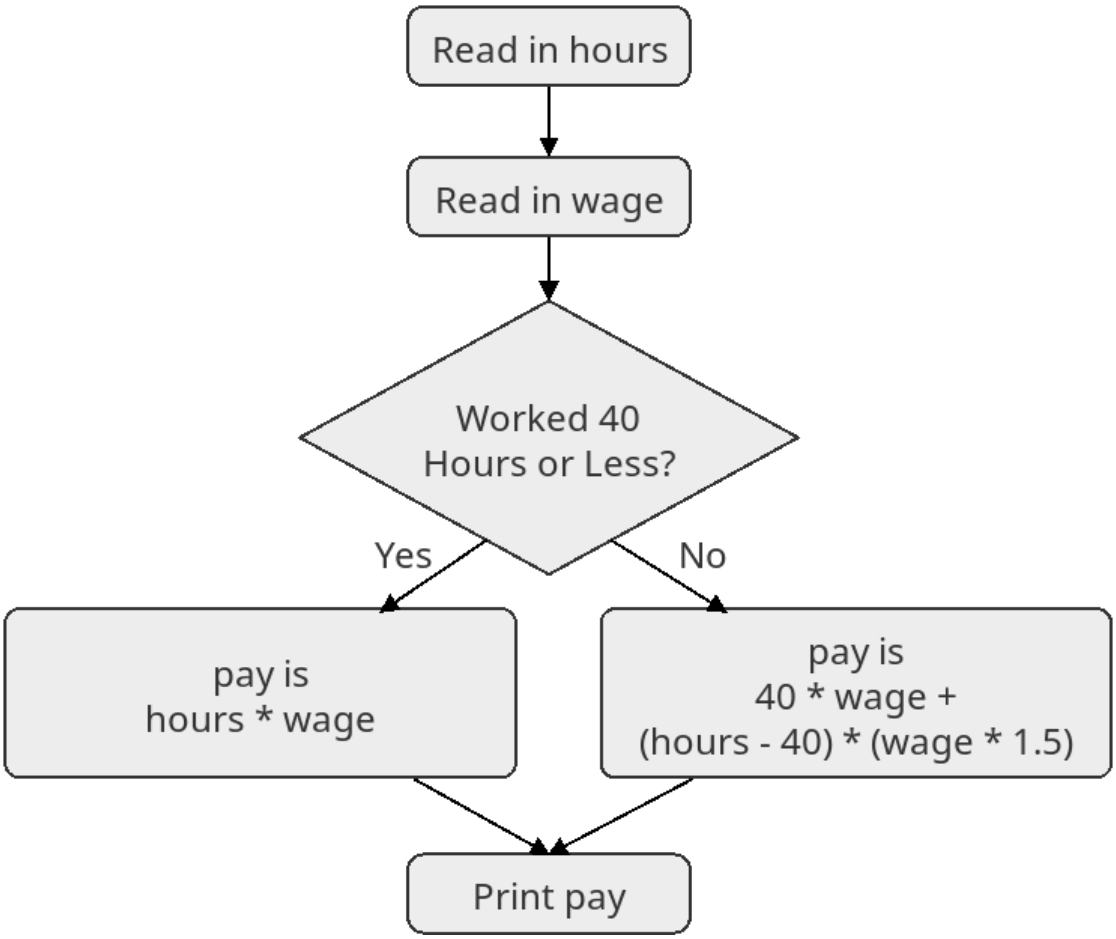

Figure 19: A flowchart showing the steps in the pay calculator algorithm

6. Why is testing each step of an algorithm important before implementing it in code?

6.10 Programming Exercises

1. Write a program which counts how many times a character appears within a string. Begin by reading in a string from the user. Next, read in a single character (which is just a string you can assume has length 1). Loop over the string and count how many characters match the one you are looking for, and report that to the user at the end.

2. It's harder than you probably think to determine if a year is a leap year or not – it's not just every 4 years. The rules are:

   ```
   1. If the year is divisible by 400 it's a leap year
   2. Else if it's divisible by 100 it's not a leap year
   ```

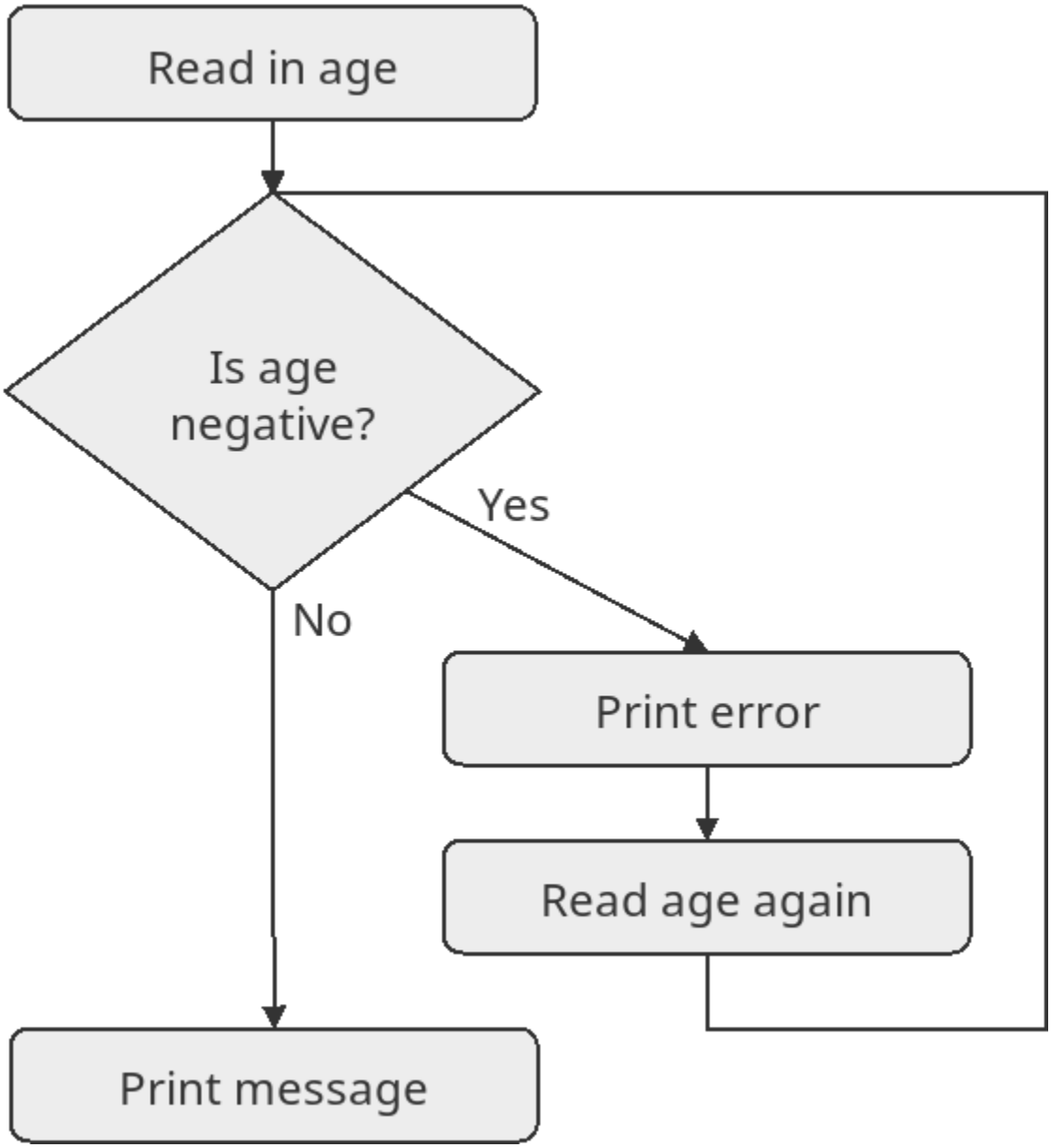

Figure 20: A flowchart showing the steps of reading a valid age from the user

```
3. Else if it's divisible by 4 it is a leap year
4. Otherwise it's not a leap year
```

Write a Python program to read in a year and tell the user whether it's a leap year or not.

3. Write a program to find the largest number in a sequence. Start by asking the user to tell you how many numbers they will provide. Next write a loop that runs that many times. Inside the loop, ask the user to give you a number and keep track of which one is the biggest. At the end of the program, you should print the largest number in the sequence the user entered.

4. The Collatz Conjecture states that if we take any integer N , and repeatedly do the following steps to it:

   ```
   If it's even, divide it by two
   If it's odd, multiply it by 3 and add 1.
   ```

   Then we will eventually hit 1. For example if we start with 5, we go though the following steps:

   ```
   5 (starting number)
   16 (times by 3 and add 1)
   8 (divide by 2)
   4 (divide by 2)
   2 (divide by 2)
   1 (divide by 2)
   ```

   This conjecture was raised by mathematician Lothar Collatz in 1937. Every number anyone has ever tried has eventually gotten to 1, but mathematicians have not been able to prove if this will work for all numbers or not.

   You should write a program which reads in the starting number and prints out the numbers in the sequence until you hit 1.

5. An infinite series is a list of numbers with some repeating pattern to them. One famous infinite series is the following: $\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64}$ Write a program to read in a number, called N, from the user. Then add up the first N numbers in this series. For example, if the user enters 3, it should add $\frac{1}{1} + \frac{1}{2} + \frac{1}{4}$ and print the answer.

Chapter Summary

- If, elif and else statements, along with our loop statements form Python's "control flow" statements. These allow us to control the order that program instructions are done in.
- These control flow statements can be combined, or nested, any way we like. We can put a loop inside of an if statement, or an if statement inside of a loop for example. This ability allows us to solve more complex problems.

- When tackling a new problem, there are some steps to go about solving the problem:
    1. Identify the inputs
    2. Identify the outputs
    3. Solve a few examples by hand
    4. Write the steps needed in plain English
    5. Test these steps and look for problems with them
    6. Translate your algorithm into code
    7. Test the code you've written
- You can use pseudocode or flowcharts to work on the general solution to a problem before diving into code.
- To get better at algorithmic problem solving, you need to practice solving problems!

Chapter 7: Using Libraries

Learning Objectives

- Understand what a library is.
- Learn how to import libraries into a Python program.
- Learn some of the functions of the math and random libraries.
- Understand how to find libraries that are available for Python, and learn how to use them.
- Learn how to install 3rd-party Python libraries in Thonny.

7.1 Overview

Python comes with some built-in functions that we've used in our programs. These include things like `print`, `input`, `round`, and `len`. We can use any of these directly in a Python program without needing to do anything special.

Python also comes with lots of **libraries**, that we can use in our programs. A library is essentially a collection of functions that we can use in our own programs to help us. Python has ready-made libraries for doing all sorts of things like solving math problems, getting random numbers, handling dates and times, and even working with websites, databases, and graphical interfaces. Python does not let you call on all of these automatically, but the libraries included with Python are easy to access, and we will see how to do that in this chapter.

There are also lots of other libraries that Python does not come with, but which can be downloaded from the Internet. These allow programmers to use code that was created by other programmers all across the world. Sharing work with other people and building off of what others have done is a big part of computer science. This makes it possible to make cooler things than if everyone had to work alone.

Lastly in this chapter, we'll talk about how to look up documentation on libraries so you can learn how to use them.

7.2 The import Statement

In order to use a library, we must put an `import` statement in our program. This tells Python to load the library so that we access the functions inside of it. For example, we will start by looking at the math library. In order to use this library, we could do the following:

```
import math
```

After doing the import, we will be able to call upon any of the functions inside the math library. One thing that the math library comes with is the `sqrt` function. This finds the

square root of a number. For example, the following programs asks the user to enter a number and then prints the square root of it:

```
import math

num = float(input("Enter a number: "))
root = math.sqrt(num)
print("The square root of", num, "is", root)
```

Here we start with the `import` statement, and then later use `sqrt`. The import of math has to be before we actually use anything from the math library. Usually people put import lines all at the top of a program, so we make sure things are imported before we use them.

Also note that we don't just call `sqrt` directly, we have to put the name of the library first, followed by a `.`, followed by the name of the function we're using. The reason for this is to help keep things organized. The math library has over 50 things in it, and if we are using lots of libraries in one program, it could be hard to know what things are coming from where. When we see `math.sqrt`, we know that the sqrt function comes from the math library which we imported above.

7.3 Another form of import

Sometimes we want to use some functions from a library and we don't want to have to put the name of the library and the `.` before the name of the functions every single time. For example, imagine we are writing a program that had to do lots and lots of square roots. It might get kind of tedious to write `math.sqrt` every time instead of just `sqrt`.

Python allows us to do that by using a different form of the import command, which looks like this:

```
from math import sqrt
```

This tells Python that we want to use the `sqrt` function from the math library, and that we want to do it without having to write `math.` before it every time. Now our program to calculate square roots might look like this:

```
from math import sqrt

num = float(input("Enter a number: "))
root = sqrt(num)
print("The square root of", num, "is", root)
```

Notice that we now *can* just say `sqrt` instead of having to put `math.sqrt`. However, now we can't use anything from the math library *except* for sqrt. If we wanted to be able

to call upon multiple things from the math library (like the `sin` function for calculating the sine of an angle), we could add it to the from statement like this:

```
from math import sqrt, sin
```

Now we could call upon the `sqrt` or the `sin` functions, both without using `math.`. But we couldn't call upon any of the other things in the math library.

If we really want to be able to use *everything*, from math, and we don't want to type `math.` for any of it, we can do so like this:

```
from math import *
```

The `*` means everything. So now we can call upon the `sqrt` and `sin` functions, along with everything else in math, without needing the `math.` beforehand.

Generally it's a good idea to stick with the first version of import most of the time. That makes your code clearer since everyone can tell what library everything we use is part of. There are cases, however, when one program uses lots and lots of things from one library where using the other form is more convenient.

7.4 Documentation Pages

So the math library has lots of things we might want to use in it. But how do we find out all what's in there? And how do we learn how to use those things? The answer is that we read the documentation. The Python website has pages of documentation for every one of the libraries that it comes with. The page for the math library is available here.

Just like reading the manual for an electronic device that you purchase, reading programming documentation can be a bit overwhelming. The math page includes *all* of the information you would need to use the math library. Usually you have a specific question, and will need to hunt through the documentation page looking for the answer.

Luckily, the pages are broken down into categories, and each function is labelled. For example, the description listed for `sqrt` looks like this:

math.**sqrt**(*x*)

Return the square root of *x*.

To read this, we would see that the name of the function is `sqrt`. We would see that it's part of the math library, so we need to put `math.` before using it (unless we use the from style of import). We also see that there is one thing inside of the parenthesis. Then the description of the function says that it returns to us the square root of that thing (which it calls `x`).

Another important question is "How does one know what libraries Python even comes with?" The answer to that question is that Python also includes a list of all the libraries that it comes with. You can find that page here. It starts listing the "built-in" things such as `print`, `len`, etc. The it lists all of the things you can import into Python, grouped in categories.

7.5 Example: Password Entropy

As an example of using the math library, let's look at the problem of figuring out how difficult a password would be to guess based on how long the password is, and how many characters might be in it. This is related to the problem we looked at last week of checking that a password met the length and character requirements. But now we are seeing how good those requirements themselves are.

Cryptographers use a concept called password entropy, which is a numerical measure of how difficult a password would be to guess. This is based on how many available characters might be in the password, and the length of the password. The more possible characters might be in the password, the harder it is to guess, and the longer the password is, the harder it is to guess. It is calculated as:

$E = \log_2(C^L)$

Where:
C = the number of possible unique characters available
L = the length of the password
E = the password entropy

In order to write this program, we need to somehow calculate a logarithm with a base of 2. Python's math library actually has a function that does exactly this. You can read about it here. We can use this function to write a program to calculate the password entropy:

```
import math

# read the input
chars = int(input("How many characters are available? "))
length = int(input("How long is the password required to be? "))

# perform the calculation
entropy = math.log2(chars ** length)

print("The entropy of this is", entropy)
```

Here we only had one thing in the math library we wanted, so we stuck with the normal `import math` line to pull in the math library. Then when we needed to do the logarithm, we call the function we need with `math.` before it.

Let's try the program out. First we will figure out the password entropy of a 4-digit pin number:

```
How many characters are available? 10
How long is the password required to be? 4
The entropy of this is 13.287712379549449
```

The higher the number, the harder the password would be to guess. Let's try the scheme of needing 8 characters for a password, and pulling from lower-case letters, upper-case letters, and digits:

```
How many characters are available? 62
How long is the password required to be? 8
The entropy of this is 47.633570483095
```

The 62 comes from the 26 lower-case letters, plus the 26 upper-case, plus the 10 digits. This password scheme is stronger because it has a much higher entropy value!

7.6 The random Library

Another very useful library that is included with Python is the `random` library, for getting random numbers. You can read all about it on its documentation page here.

Probably the most useful thing in it for us now is `randint` which we can use to get random integers between two values. For example, the following program gets the starting point and ending point from the user, and then prints a random number between those two (including both ends):

```
import random

a = int(input("Starting point: "))
b = int(input("Ending point: "))

num = random.randint(a, b)
print("Your random number is", num)
```

Random numbers are used for all sorts of things in programs. They are used for games that give you random interactions (for example, some games give you random "loot" when you defeat enemies). Also some algorithms use random numbers as an important part of how they work.

In actual fact, computers cannot really give us truly random numbers. The values that `randint` provides actually are produced from mathematical sequences. They *seem* random, but are in fact not. These numbers are called "pseudo-random" for that reason[1].

Now that we can use random numbers, we can write a simple "Rock, Paper, Scissors" game. In this game, two players each pick one of the three possible throws from the name of the game. There are then rules for determining the winner based on the two throws where:

- Rock beats scissors
- Scissors beats paper
- Paper beats rock

In order to write this program, we will do three main things:

1. Get the user's throw by asking them in an input statement
2. Get the computer's throw by choosing a random number
3. Compare the two and determine a winner

The code for this program is below:

```
import random

# get the user's throw
user = input("Rock, Paper, or Scissors? ")

# pick the computer's throw randomly
num = random.randint(1, 3)
if num == 1:
    comp = "Rock"
elif num == 2:
    comp = "Paper"
else:
    comp = "Scissors"
print("Computer throws", comp)

# if they threw the same thing, it's a tie
if user == comp:
    print("Tie!")

# if the user throws something that beats the computer
elif user == "Rock" and comp == "Scissors" or \
     user == "Scissors" and comp == "Paper" or \
     user == "Paper" and comp == "Rock":
```

[1]There have been cases where programmers use this fact to figure out the algorithms behind "random" gambling games like slot machines and predict when the games will produce a payout.

```
    print("You win!")
else:
    print("You lose!")
```

This program does something we have not seen yet, which is to break a line of code into pieces. The condition which tests to see if the user wins is very long. We could write it all on one line, but here we split it into three lines. Because Python cares about indentation, we have to tell it that those three lines are actually still part of one long condition. We do that by putting the \ symbol at the end of the line. That tells Python that the line continues below.

This program would not work (or at least not be very fun) without random numbers. We pick the random number to be between 1 and 3. We then use the random number to decide which of the three things the computer should play. Here is an example run where we were lucky enough to win:

```
Rock, Paper, or Scissors? Paper
Computer throws Rock
You win!
```

7.7 Third-Party Libraries

The math and random libraries both come with Python. If you have Python installed, you can import them and start using them right away. In this section we'll talk about how to use libraries that Python *doesn't* come with. These are sometimes called "third-party" libraries since some other person besides you and the language designers created them.

Python programmers who want to share a library they created with the world do so on the Python Package Index (PyPI). This is a common repository for sharing code in a placer everyone can find it. Most of the widely used libraries on PyPI are very complex, with many of them having entire textbooks devoted just to them.

As an example of one simple third-party library, we will talk about the "art" library, which you can read about on the PyPI art page. This library solves the very important problem of printing messages in fancier "ASCII art" fonts. To be able to use it, we first must install it. Luckily, Thonny makes this very easy.

Start by clicking "Tools" on the toolbar, and then "Manage Packages...". That pulls up a window that looks something like this:

Now we can search PyPI for the art package. There are several results, so click on the one that just says "art":

That brings you to this page. Click "Install" to install the package:

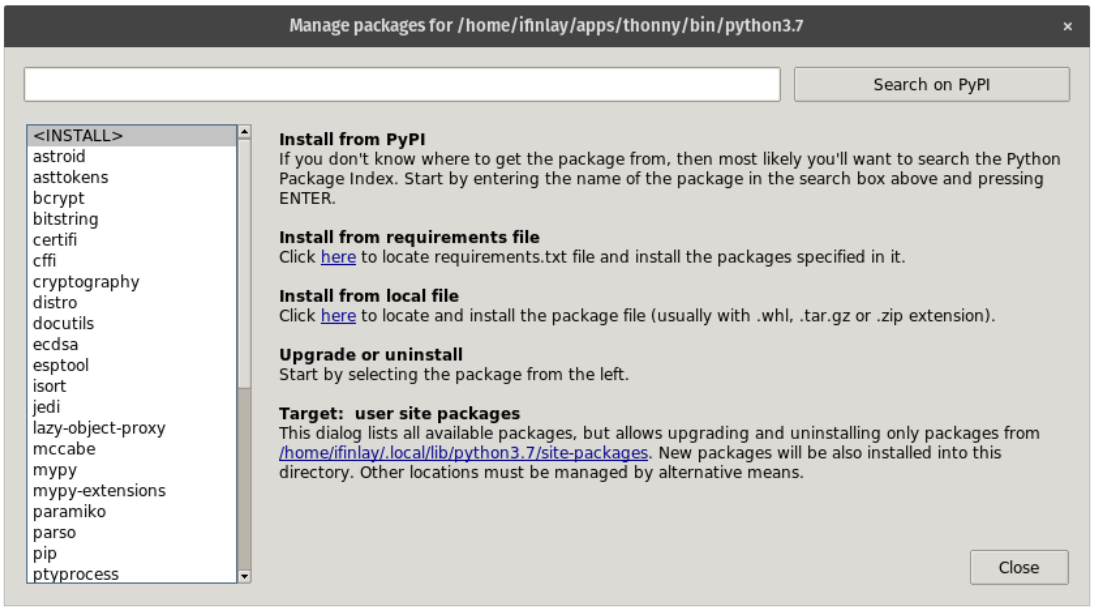

Figure 21: The Manage Packages Screen

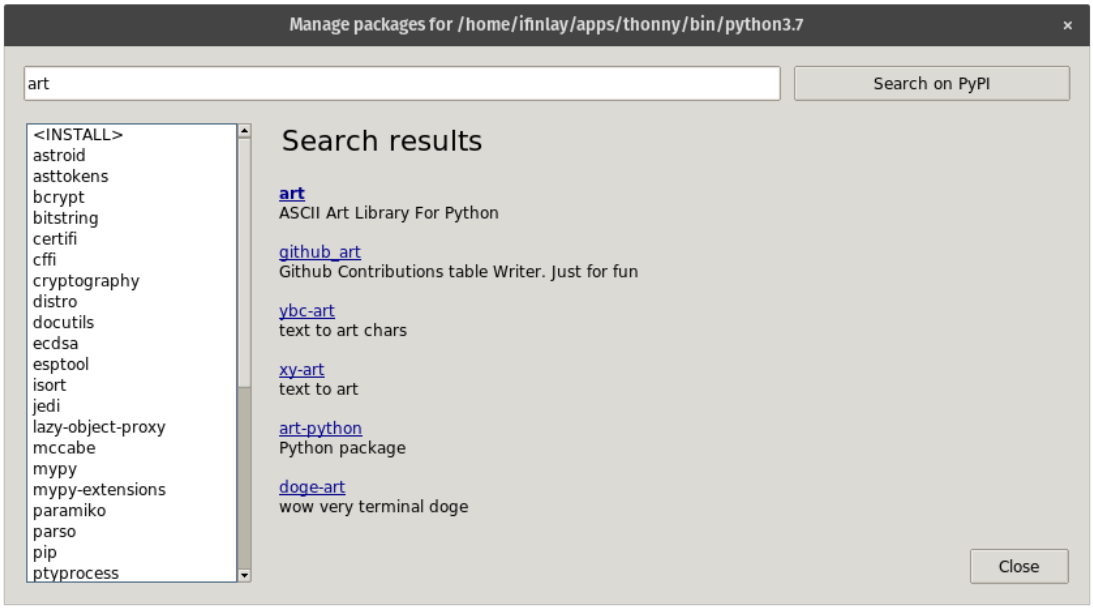

Figure 22: The Search Results for “art”

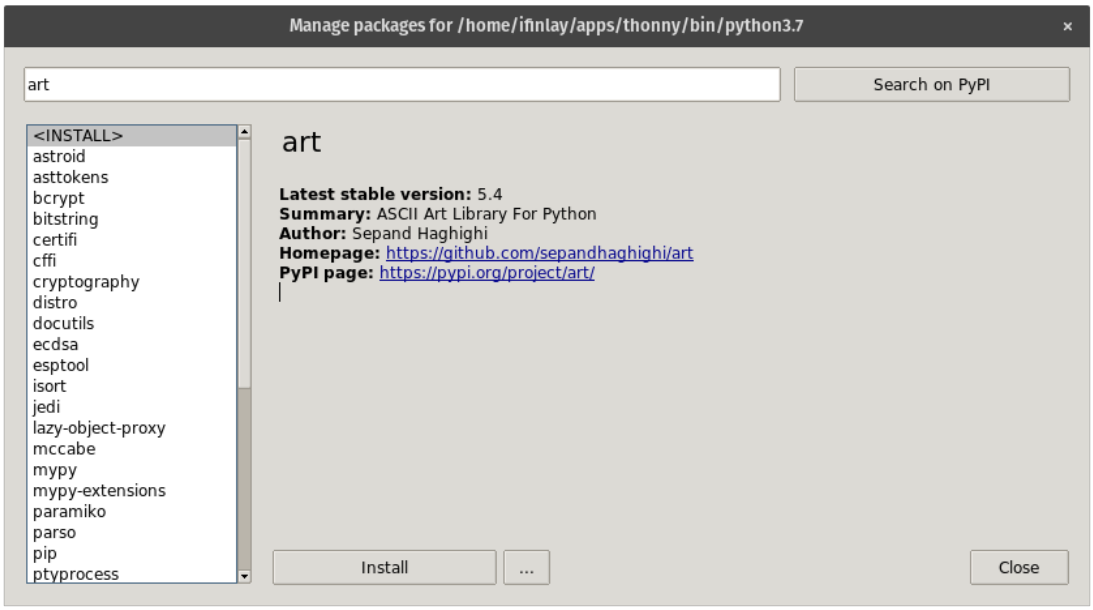

Figure 23: The art Package Screen

We should now have the library and be ready to use it. It comes with several functions you can read about on its home page. For now let's just call the `art.tprint` function which takes a string and prints it out in a fancy ASCII art font:

```
import art

# print the message in a fancy way
art.tprint("Hello!")
```

That gives us this exciting output:

```
 _    _          _  _                _
| |  | |  ___  | || |  ___    | |
| |_| | / _ \| || | / _ \ | |
|  _  ||  __/| || || (_) ||_|
|_| |_| \___||_||_| \___/ (_)
```

7.8 Using Libraries

The functions and things that can be used in Python come in three broad categories:

1. Python has some "built-in" things that can be used without having to import anything. Those can be seen on this page, and include `print`, `len`, `round` and so on.
2. Python also comes with lots of libraries pre-installed, but which you need to import to be able to call upon. These include the math and random libraries that we have seen in this chapter. You can find a list of these on this page. We will see a few more examples as we go on.
3. Lastly, anyone can create their own Python libraries. There are thousands of Python libraries that programmers have created and shared with others on the Internet. Thonny allows you to download them from the Python Package Index. Once downloaded, you can import them and use them in your programs.

One of the cool things about coding is that you can build upon the work of others. It would be overwhelming (if not downright impossible) to build a big program from scratch and have to write everything in it yourself. Instead, you can use things that come with Python or written by others as a part of your program.

7.9 Comprehension Questions

1. What is a library in Python and why is it useful?
2. What is the difference between `import math` and `from math import *`?
3. How can we find out what is available in a library like math?
4. What are third-party libraries in Python, and how can they be installed and used in a program like Thonny?
5. What does the random.randint function do?

7.10 Programming Exercises

1. Use the random library we aw this chapter to write a guess the number program in which the *computer* thinks of the number and the *user* tries to guess it.

2. Create a simple math quiz game where the program generates two random numbers and asks the user to compute their sum, difference, or product. Provide feedback based on the user's answer. The user should be able to choose ahead of time how many problems they want. At the end, give the user a report of how many they got right and what percentage.

3. The calendar library that comes with Python has useful functions for dealing with dates. Use the `weekday` function to write a program which asks the user to enter a date (including month, day, and year) and telling them what day of the week that date was. For example, your program will be able to tell us that October 31, 2024 falls on a Thursday or that January 1, 1731 was a Monday.

4. The time library has functions for dealing with time. One of the most commonly used is the `sleep` function which pauses for a set amount of time. Go back to one

of your previous programs and insert some pauses to give the user time to read output before moving on to the next line.

5. There is a 3rd party Python library called colorama which can be used to colorize our print messages. Go back to one of your previous programs and use this library to provide colored output.

Chapter Summary

- Python comes with some functions available by default, such as `print`, `range`, and `len`. Other things are included with Python, but we have to import libraries to access them.
- The import statement allows us to tell Python we want to use a library. The basic and most common form of it imports the library, but we must specify the name of the library and a `.` before each thing we use from it.
- There is another form of import we can use which starts with the keyword "from". This form allows us to import things from a library without needing to put the name of the library and the `.` before each thing.
- Each library has a page of documentation which shows you all of the things in the library and teaches you how to use them.
- Programmers use libraries to make programming easier and can even share their own code as libraries for others to build off.

Chapter 8: Lists

Learning Objectives

- Understand the purpose of list variables in Python.
- Learn how to create lists of data, and index them to get values out.
- Learn how to loop through lists with for loops.
- Be able to add values to a list.
- Learn how to read in lists from the user.

8.1 Storing Lots of Variables

Imagine that we were writing a program where we wanted to store a large amount of data. For example, imagine we were writing a program to figure calculate your grade in a class. We might want to store all of our grades into the program. Let's say we have 10 quiz grades in the class. We might write code like this:

```
quiz1 = 88
quiz2 = 94
quiz3 = 76
quiz4 = 100
quiz5 = 92
quiz6 = 89
quiz7 = 95
quiz8 = 85
quiz9 = 79
quiz10 = 99
```

Then we might want to add up all of these grades to figure out our average:

```
total = quiz1 + quiz2 + quiz3 + quiz4 + quiz5 + quiz6 + \
    quiz7 + quiz8 + quiz9 + quiz10
average = total / 10
```

This hopefully seems a little bit tedious and repetitive. We have to make 10 different variables and do the same thing with all of them. Here, it's not *too* bad with 10. But imagine if we had even more numbers we wanted to keep track. For instance, imagine you're *teaching* a course and wanted to store all 10 quiz grades from 25 students. That would be a lot of variables!

By the way, notice how the line ends in a \ character. This allows for long lines to be continued without Python thinking that the statement is done. Without this, this line of code would be too long to fit on one page! Generally super long lines of code are a sign there might be a better way to do something.

There is a better way of doing this which is to use a **list**. A list is a collection of multiple pieces of data that are stored together in one variable.

8.2 Creating Lists

To create a list, we can put the values in the list between square brackets, separated by commas. For example, we can store our 10 quiz grades in a list like this:

```
quizzes = [88, 94, 76, 100, 92, 89, 95, 85, 79, 99]
```

We could also create a list of strings:

```
names = ["Bob", "Alice", "Joe", "Mary"]
```

Each of these lists stores *all* of the values inside them. Lists provide a convenient way to store a bunch of things together with one name to access them.

8.3 Accessing List Elements

Once we have created a list, we can access each thing in the list. To do this, we can use the position of each element we want to access. Like strings, the first thing in the list is at position 0. The second element is at position 1, and so on. Starting at position 1 is a common mistake in programming.

In order to access an element, we use the name of the list, then the position inside of brackets. So to access the first quiz grade in the above list, we would use:

```
print(quizzes[0])
```

To access the last name above, we can say:

```
print(names[3])
```

When we use the position number to access the things inside of a list, we say that we are *indexing* the list.

If we use an index that is too big for our list, Python will give us an error message:

```
>>> print(names[4])
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in
    print(names[4])
IndexError: list index out of range
```

8.4 Example: Dates

Let's say we want to write a program that converts a numerical date into one using words for the month. For example, we can put in 3 for the month and 25 for the day and it will print out "March 25". We could do it with if statements like this:

```
if month == 1:
    print("January", day)
elif month == 2:
    print("February", day)
elif month == 3:
    print("March", day)
# and so on...
```

But this can actually be done with less code (and more efficiently) using lists. We can make a list of all the month names. We could then *index* the list with the month number. We will have to subtract 1 from the index because month numbers start at 1, but list numbers start at 0.

```
# get our input
month = int(input("What is the month? "))
day = int(input("What is the day? "))

# make a list of all the names of months
names = ["January", "February", "March", "April", "May",
         "June", "July", "August", "September", "October",
         "November", "December"]

# get the name of this month by indexing
monthName = names[month - 1]

# print our output
print("It is", monthName, day)
```

This program works by taking in the month as a number. It then subtracts one from this number and uses it to index the list. So if the month number is 5, it subtracts 1 to get 4. It then uses 4 as the index to get the name "May" out of the list.

Below is an example of this program running:

```
What is the month? 6
What is the day? 22
It is June 22
```

8.5 Looping Through a List

One super common thing to do with a list is to loop through everything in the list and do something with each thing in it. For example, we might loop through a list of our quiz scores to add them up, or loop through a list of names searching for one name in particular.

We can do this by using a for loop in Python. We have seen for loops for looping through strings and sequences of numbers using range(). They also work for lists.

For example, if we want to print all of our quiz grades, we could write code like this:

```
quizzes = [88, 94, 76, 100, 92, 89, 95, 85, 79, 99]

for quiz in quizzes:
    print(quiz)
```

The for loop assigns each thing in the list to `quiz`, one by one, and executes the lines under the for loop on it. In this case, it will print all of the quiz scores to the screen one by one. By changing the code in the loop, we can do different things with each item.

To add all of the quiz scores together, we can do the following:

```
# our list of quiz scores
quizzes = [88, 94, 76, 100, 92, 89, 95, 85, 79, 99]

# figure out the total score
total = 0
for quiz in quizzes:
    total = total + quiz

# find the average and print it
average = total / len(quizzes)
print("Your average score is", average)
```

The `total` variable here is worth talking about a little bit. We set it to 0 before the loop and then also set it inside the loop. What we set it to there is `total + quiz`. So the first thing it's equal to is 0, then we set it to that 0 plus the first quiz value, so it becomes 88. Then the next time through the loop, we set it to the 88 it is now plus the 94, resulting in total storing 182. This kind of "accumulation" loop is common in coding.

This is much less code than having to add all 10 variables by typing them all out! We also can add more quiz scores to the list without having to redo the code to find the average. This example also shows using the `len` function to find the length of a list — this works the same way it does for strings.

8.6 Example: Smart Guess the Number

When we first looked at using loops, we saw an example of a guess the number program that started at 1, and then went on guessing up to 10. To refresh your memory, this code looked like this:

```
# guess 1 the first time
guess = 1
answer = input("Did you guess " + str(guess) + "? ")

# keep guessing the next number until they get it
while answer != "yes":
    guess = guess + 1
    answer = input("Did you guess " + str(guess) + "? ")

print("Got it!")
```

This program does in fact eventually guess the number the user is thinking of, but it doesn't guess them in a very human-like manner. Only a robot would guess in order like that. Another way is to guess the numbers randomly, instead of in order:

```
import random

# guess a random number
guess = random.randint(1, 10)
answer = input("Did you guess " + str(guess) + "? ")

# keep guessing in order until we get it
while answer != "yes":
    guess = random.randint(1, 10)
    answer = input("Did you guess " + str(guess) + "? ")

print("Got it!")
```

Now the program no longer guesses the numbers in order, but it still doesn't guess them very well. Now each guess is random with no memory of what the previous guesses were. It could even guess the same number twice in a row. What we'd like is for the number to guess the numbers from 1 to 10 in random order without repeating itself.

We can do this using a list of numbers to guess and then "shuffling" the list. The idea here is that we will make a list to store all of the numbers 1 through 10. Then we call the `random.shuffle` function which takes a list and shuffles it randomly.

We then loop through this shuffled list and guess the numbers in it one by one. Now the program will guess them in a random way, but it won't ever guess the same number twice.

The code to do this is below:

```
import random

guesses = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(guesses)

for number in guesses:
     if input("Did you guess " + str(number) + "? ") == "yes":
          print("Got it!")
          break
```

There's something else new in this program which is the `break` command. This exits out of the loop immediately when we run it. By doing the break when we get the number right, we cause the loop to exit early, when it might not have gone through all of the numbers yet. Without this, the program would keep asking us until it went through all 10 numbers.

An example run of this program is below:

```
Did you guess 6? no
Did you guess 8? no
Did you guess 4? no
Did you guess 2? no
Did you guess 5? no
Did you guess 1? no
Did you guess 7? yes
Got it!
```

8.7 Splitting Input

One very helpful thing we can do with strings is to call their `split` method, which splits the string into parts by some separator (which we get to choose). This can allow us to go through each word of a sentence, for example.

The following program does just that. It asks the user to input a sentence. It then calls the `split` method to split it based on spaces. This gives us back a list variable (which we call words). We can then do what we want to with those individual words. Here we just print them out.

```
# read in a whole sentence
sentence = input("Enter a sentence: ")

# split it into words (separated by spaces)
```

```
words = sentence.split(" ")

# loop through each one
for word in words:
    print(word)
```

Below is an example run of this program:

```
Enter a sentence: the quick brown fox jumps over the lazy dog
the
quick
brown
fox
jumps
over
the
lazy
dog
```

We can use this to improve our quiz grade averaging program. In this program we "hard-coded" our quiz scores into the program with the following line of code:

```
quizzes = [88, 94, 76, 100, 92, 89, 95, 85, 79, 99]
```

Let's say that our program was so super helpful that we wanted to share it with our friends. We would not necessarily want them to have to edit our code to put in their own grades. So instead we should make it so the program asks you for the quiz grades.

There are a couple of ways to do this, but one is to make the program read them in on one line and then split them up. We can use any kind of separator we want to separate the numbers. Let's say we want to use a comma.

The following program does this by asking the user to enter their quiz grades, splitting them into individual things in a list, and then looping through them.

```
# read in a whole sentence
quizzes = input("Enter quiz grades separated by commas: ")

# split it into parts
quizzes = quizzes.split(",")

# loop through each one getting total
total = 0
for quiz in quizzes:
```

```
    total = total + int(quiz)

# print average
print("Average is", total / len(quizzes))
```

One wrinkle here is that our list is actually storing strings, because that's what the `split` method returns to us. So when we do the adding we have got to convert the numbers to int first.

Below is an example of this program being run:

```
Enter quiz grades separated by commas: 92,78,88,70,100,94
Average is 87.0
```

Now we are reading in the list from the user, and looping over it to calculate the average.

8.8 Adding to a List

So far we have looked at making lists all in one go, either by getting the list contents from `split`, or by listing the things inside brackets, like this:

```
names = ["Bob", "Alice", "Joe", "Mary"]
```

But what if we want to add an item to a list that already exists? This can be done with the `append` list method. For instance, this code will add two new names to the list:

```
names.append("Joe")
names.append("Beatrice")
```

This allows us build a list as we go, rather than create the whole thing at once. As we will see, there are lots of cases where being able to add to a list is handy.

The `append` method adds items on to the *end* of the list. If we wanted to add an item somewhere else, we can use the `insert` method instead. This method takes two parameters. The first is the index we want to insert at. The second parameter is the item we would like to insert.

The following code starts by making an empty list, and then inserting some names into it at different positions:

```
names = []
names.insert(0, "Bob")
names.insert(0, "Alice")
names.insert(1, "James")
```

```
print(names)
```

This example will print `['Alice', 'James', 'Bob']`.

When "Bob" is inserted at position 0, it's the only one, so the list is `["Bob"]`. Then, when "Alice" is inserted at location 0, the list becomes `["Alice", "Bob"]`. Finally, when "James" is inserted at location 1, he is inserted between Alice and Bob to make the list `["Alice", "James", "Bob"]`.

If we care about the *order* of the list, insert lets us choose where to put new items.

8.9 Reading in a List

We've seen one way to read in a list in Python, using the `split` method. Here we ask the user to enter all the values on one line, with some separator like a space or comma. This works well sometimes, but there are some downsides. One is that it reads it all in as strings, and another is that it might be inconvenient if there's a lot of items to read.

Another way to input a list from the user is to read in each value separately and then add them to the list one by one. With this approach, we can read them in as numbers.

The following program attempts to do this:

```
numbers = []
while True:
    item = int(input("Enter a number: "))
    numbers.append(item)
    print(numbers)
```

The only problem with this code is that it is an infinite loop. We need to have some way of knowing when to stop!

There are two ways that this could be done:

1. Ask the user up front how many items they would like to enter, and then loop that many times. The following example does this:

```
quizzes = []
count = int(input("How many quizzes are there? "))

for i in range(count):
    item = int(input("Enter a quiz grade: "))
    quizzes.append(item)
```

2. Have a certain value reserved to mean "I'm done now". A value used in this manner is called a *sentinel* value. If we are entering numbers that should always be positive, like quiz grades, then we can use -1 as the sentinel.

   An example doing it this way is below:

```
quizzes = []
item = int(input("Enter a quiz grade: "))

while item >= 0:
    quizzes.append(item)
    item = int(input("Enter a quiz grade: "))

print(quizzes)
```

Note in this example that we need to read in a quiz grade twice. The first time is done before the loop to make sure that the `item` variable is defined before we test it in our while condition. Then we read it again inside the loop to make sure it can happen for every quiz the user wants to enter.

8.10 Comprehension Questions

1. What is the main purpose of using lists in Python?
2. How can you access the third element in a list named `items`?
3. If you have a list pets equal to `["dog", "cat", "bird"]`, how do you add "fish" to the end of the list?
4. What will `len(pets)` return after executing the append operation in the previous question?
5. Is it more common to loop over a list with a for loop or a while loop?
6. What happens if you try to access an index from a list that does not exist? Like it we try to read `pets[99]` in the list above?

8.11 Programming Exercises

1. Write a program to read in a list of numbers and print one randomly selected element.
2. Write a program to read in a list of strings and then one string to search for. Then loop through the list and tell the user if the string they are looking for is in the list or not.
3. Write a program which reads in a list of numbers from the user and then prints all of those numbers which are greater than 50.
4. Write a program which reads in a list of strings from the user. Your program should then print out the list of strings backwards. There are two main ways to do this. You could put the strings into the list backwards right when you read them in. This

will build the list backwards, and then you can print it out forwards. Or you could put the strings into the list in order, then loop through them backwards when you print them. You can use a for loop with range to go through the indexes in reverse order.

Chapter Summary

- Lists store multiple pieces of information together in one variable. This is helpful when you have lots of related things to store.
- You can create a list all at once, by putting the things in the list inside square brackets, separated by commas.
- You can also add things into a list that was already created using either `append`, which adds to the end, or `insert` which can put something into the middle of a list.
- You can get individual items out of a list using brackets with an index inside of them. Like strings, the indices start at 0.
- We can loop through lists using for loops, which go through each item in the list one by one.
- There are different ways to read in a list from the user. We can read a bunch of items in one line, and use `split` to split them up, we can also read them one by one and add them to the list as we go.

Chapter 9: Functions

Learning Objectives

- Understand some of the benefits of splitting a program into multiple functions.
- Learn how to create our own functions.
- Understand the purpose of parameters and how to write functions that use them.
- Understand the purpose of return values and how to use them.
- Learn what "scope" is and how it affects variables in functions.

9.1 What is a Function?

We have been calling functions in Python ever since the very first "Hello World!" program:

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

Here we are calling the `print` function and asking it to print the given message to the screen for us. Every function has a job to do. The job of the print function is to print thins to the screen. When we "call" a function, we ask it to do its job.

Functions can have **parameters**, which are the things between the parenthesis. Parameters allow us to pass data to a function to control how it works. The print function prints out each of its parameters.

Consider the following function call:

```
import math
x = math.sqrt(144.0)
```

The `math.sqrt` function takes a number as a parameter. The job it has to do is to calculate the square root of the parameter. Unlike the print function, the sqrt function *returns* a value. We use parameters to give information to a function, and return values let functions give information back to us. So here, 144.0 is the parameter, and the return value will be 12.

Not all functions return values. If we try to print the return value from the `print` function, we'll get the value "None":

```
>>> x = print("Hello")
>>> print(x)
None
```

So parameters allow us to pass information to functions. Some functions take no parameters, some take one, and some take more. Likewise some functions return a value back and some do not.

9.2 Why Write Functions?

In addition to calling functions that already exist, we can create our own. There are many reasons to do this. The first is to split our code up into more manageable pieces. Just like books are divided into paragraphs and chapters, programs can be divided up into functions to make them easier to understand and write.

Another reason is to decrease repetitive code. Imagine you were writing a Program that reads in the size of a rectangle so it can compute the area and perimeter. It needs the width and height to do this, and both numbers should be greater than zero. We should make sure that's the case, so our code to read these values might look like this:

```
# read the width
width = int(input("Width: "))
while width <= 0:
    print("Please enter a positive number.")
    width = int(input("Width: "))

# read the height
height = int(input("Height: "))
while height <= 0:
    print("Please enter a positive number.")
    height = int(input("Height: "))
```

This code is pretty repetitive. The input line, while condition and loop bodies are all basically the same. The only things different are the variable name being used and the input prompt.

Having repetitive code in programs is not ideal. For one, we have to write more code. A basic tenet of programming is to not do more work than you have to. Another, even bigger, reason is that now we have to *maintain* the code in two places. If we want to, say, make it so that it doesn't crash the program when a letter is accidentally entered, then would have to make that change in *two* places. As we write larger programs that take longer to debug, having repetitive code like this is a bad idea.

If we were to write a function to read in a positive number, then we can just call upon it whenever we need to, and know that it already does this job. That might look like this:

```
# read height and width
width = readPositive("Width: ")
height = readPositive("Height: ")
```

We pass in the prompts, because that was the one thing that's really different. Of course we have to create the `readPositive` function for this to work. So we will see how to make functions next.

9.3 Writing a Function

So now that we know we want to write functions, we need to know how to do so. Functions are created using the following syntax:

```
def functionName(parameters):
    line 1
    line 2
    # etc ...
```

The keyword `def` starts a function, it stands for define. Function names follow the same rules as variable names: any letter or underscore, followed by any numbers of letters, numbers and underscores. Parameters are specified the same as in a function call: as a list separated by commas inside of parenthesis. Then the colon signifies the start of the function. Each line of code in the function must be indented over, as in a loop or if statement.

As an example, we can define a function that prints out a greeting:

```
def greet():
    print("Hello!")
```

Here the function's name is `greet`, and it takes no parameters. The body of the function is the code that will be run when the function gets called. This function doesn't return anything either.

If we run this program, nothing actually happens. Defining a function does not run the code inside it. It just says what *would* happen should the function ever actually be called. So in order for this program to do something, we would need to call the function after defining it, like this:

```
# define the function
def greet():
    print("Hello!")

# call the function
greet()
```

Calling the functions we make works the same as calling a functions that come with Python. Now the program will first define the function, and then actually call it, so the print statement will run and we should see this:

```
Hello!
```

Functions must be defined before they are called. So this program will only work if the definition of greet is before the call to it. If we flip them around, Python would give us an error message.

9.4 Parameters

Suppose we want to personalize the greet function so that it greets people by name. To do this, the function will need to know the name of the person it is supposed to greet. To send information into a function, we use parameters. In this case, we'll add a parameter to the function that for the person's name:

```
def greet(name):
    print("Hello,", name, "how are you?")
```

Now this function takes one parameter called `name`. Inside the function, we can refer to `name` when we need that information. In this case, we just print it out. Notice that the function isn't *setting* the `name` variable anywhere. The function just assumes it has been set to something already.

When we call the function, we pass in the actual value that should fill in for the parameter. For example, we add three calls to this function in the program below:

```
def greet(name):
    print("Hello", name, "how are you?")

greet("Alice")
greet("Bob")
greet("Claire")
```

This program outputs the following:

```
Hello Alice how are you?
Hello Bob how are you?
Hello Claire how are you?
```

Parameters let us pass information into the function to change the way it works. The function above just prints out its name variable, but that will be different based on what was passed in.

We have to get the number of parameters right, or Python won't be able to call the function. If we pass the `greet` function zero parameters, or more than one parameter, then Python will complain:

```
>>> greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() takes exactly 1 argument (0 given)

>>> greet("Too", "Many")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() takes exactly 1 positional argument (2 given)
```

As another example, let's write a function that will print a countdown from a starting point. Sometimes we will want the function to print out a countdown starting at 10 and going down to 1. Sometimes we will want to start at just 5 instead.

Because sometimes we want the starting point to be 10 and sometimes we want it to be 5, that has to be a parameter. We can then use that parameter, which we will call `start`, to control the for loop that will do the counting down. The code for this function is below:

```
def countdown(start):
    for i in range(start, 0, -1):
        print(i)
    print("Done!")

countdown(10)
countdown(5)
```

We are calling the `countdown` function with two different values for the start parameter. The output of this program appears below:

```
10
9
8
7
6
5
4
3
2
1
Done!
5
4
3
2
```

```
1
Done!
```

9.5 Return Values

Now that we have seen how to pass information *to* functions using parameters, we will look at how to pass information back *from* functions using return values.

A return value is a value that is produced by a function call. We have seen functions that produce return values such as `input` and `math.sqrt`. We can return values from our own functions using the `return` statement. This statement takes the value we want to return. The value we return can be any value in a Python program such as a constant, variable or expression.

For example, let's say we want to write a function to calculate the area of a rectangle. To do this, the information we need is the width and height of the rectangle. Because we need to know this to do our task, these should be parameters into the function. The result of our work will be the area, so this is what we should return back. The function could look like this:

```
def rectangleArea(width, height):
    area = width * height
    return area
```

The return statement sends back the value of the `area` variable which was computed. When we call this function, we'll need to save the value it gives us into a variable. If we don't than the result it gives us will be lost (just like we have to store the value that `input` gives us into a variable). The following bit of code calls the function above and then prints the result of it:

```
a = rectangleArea(3, 5)
print("The area of a 3 by 5 rectangle is", a)
```

We can also take the output of a function and pass it directly as a parameter to another function. For instance, we can print the area of a rectangle to the screen directly by calling it from inside of print:

```
print(rectangleArea(5, 7))
```

Here the rectangleArea function is called with parameters of 5 and 7. This function is going to return the value 35 which is then passed directly as a parameter to the print function. The upshot is that the 35 will be printed directly to the screen, without being stored in a variable first. We have actually been doing this same thing with the `input` function for a while in code like this:

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

Here the `input` function's return value (a string) is passed directly to the `int` function, which then returns us the integer version of that string.

A function can also have multiple `return` statements for different cases. For example, we can write a function to convert a number grade into a letter grade:

```
def numberToLetter(grade):
    if grade >= 92:
        return "A"
    elif grade >= 89:
        return "A-"
    elif grade >= 87:
        return "B+"
    elif grade >= 82:
        return "B"
    elif grade >= 79:
        return "B-"
    elif grade >= 77:
        return "C+"
    elif grade >= 72:
        return "C"
    elif grade >= 69:
        return "C-"
    elif grade >= 67:
        return "D+"
    elif grade >= 60:
        return "D"
    else:
        return "F"
```

Here we have a `return` statement for each condition in the function. When Python executes this function, the first condition that is true will have its return statement executed.

While we can have multiple `return` statements, we can only ever execute one of them for each function. As soon as we reach the `return` statement, we leave the function and don't do anything else. For example, this program has a statement after the `return`:

```
def rectangleArea(width, height):
    area = width * height
    return area
    print("Hello!")    # will never be printed
```

This will never print the "Hello!" message, because Python leaves the function as soon as the `return` is done. No statements after that will be executed.

9.6 A couple more examples

Another benefit of functions is they sometimes make code easier to read. We have seen that we can check if a number is even or odd by using the modulus operator. To refresh your memory, we can write a program to tell the user if their age is an even number like this:

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

if age % 2 == 0:
    print("Your age is even")
else:
    print("Your age is odd")
```

What's going on is that the modulus operator divides the age variable by 2, and checks the remainder. If it's 0, then the age is evenly divided by 2, so it must be even. How this code works is not exactly clear. To help make this easier to read and understand, we can write functions called `isEven` and `isOdd` to perform these calculations:

```
def isEven(num):
    if num % 2 == 0:
        return True
    else:
        return False

def isOdd(num):
    if num % 2 == 1:
        return True
    else:
        return False
```

These functions both take a number as a parameter. They then do the modulus trick to determine if the number is even or odd and then return a boolean True or False value. Now we can call them whenever we want to know if a number is even or odd, instead of having to write out the modulus expression directly.

As another example, let's say we want to write our own version of Python's `max` function? Given a list of numbers, it should return the biggest item out of it.

This function will take the list as a parameter. The return value should be the biggest item found in the list. We can do this with the following function:

```
def max(values):
    biggest = values[0]
    for v in values:
        if v > biggest:
            biggest = v
    return biggest
```

Here we loop through the list and keep track of the biggest item that we have seen so far. Once we've gone through the whole thing, we return the biggest.

Finally, let's write the code for the `readPositive` function we talked about earlier. To do this, we will make the prompt for the user a parameter, then read values until one is positive, and then use a `return` to send it back to the user. This might look like this:

```
def readPositive(prompt):
    value = int(input(prompt))
    while value <= 0:
        print("Please enter a positive number.")
        value = int(input(prompt))
    return value
```

We could even make a more general version of this function which gets a number as input from the user between any lower and upper limits. The limits could be passed into the function along with the prompt.

9.7 Scope

Scope refers to the parts of code that can access things like variables. Before we used functions, we could access any variable any place in our program, after it was created. But when we create variables inside of a function, they can only be accessed inside of that function. For example, the `hello` function below makes a variable called `message`. We are allowed to access that variable inside the `hello` function, but not outside of it:

```
def hello():
    message = "Hello!"
    # this is OK because we're inside the function
    print(message)

# this is not allowed
print(message)
```

This program produces an error because, in the last print statement, we are not inside of the hello function. Therefore the variable `message` cannot be accessed. Even if

the function hello() is called, and `message` is created, we still cannot access `message` outside of the function.

```
def hello():
    message = "Hello!"
    print(message)

# call the function
hello()

# still not OK and will cause an error
print(message)
```

The scope of the `message` variable is only inside the hello function because that's where it was made. If we made `message` outside of the function, this would be OK:

```
# message is not in a function, so its scope is the whole program
message = "Hello!"

def hello():
    # this variable CAN be called here
    print(message)

# this is OK too
print(message)
```

Here, `message` is set up outside of the function and *can* be accessed inside of functions in the program.

Variables created outside of a function are called **global** and variables inside of a function are called **local**. So the scope of a global variable is the whole program, and the scope of a local variable is just the function it's in.

The reason Python works this way is to try to keep programs more organized. If our program has a bunch of functions in it, that all make several variables, things would become messy if all of those variables could be accessed anywhere in the program. By keeping the scope of a variable just to the function it was made in, things stay organized. If you want to use some value from a function in other parts of your program, you need to use a `return` to send it back.

9.8 Designing Programs with Functions

When writing programs that are long and complicated, it's a good idea to break the program up into functions. We start the program by thinking about what pieces we need. We then make functions for each of the pieces and write them one by one. This is

sometimes called a “divide and conquer” approach because we divide the program into parts and “conquer” them one by one.

In general each function should do one specific job. If a piece of code does more than one thing, you should consider splitting it up into multiple functions.

9.9 Comprehension Questions

1. What is a function in Python, and why is it beneficial to use functions in your programs?
2. What is the difference between writing a function and calling it?
3. What term is used to describe passing information *into* a function?
4. What term is used to describe passing information *back from* a function?
5. What does the term “scope” refer to in the context of functions?
6. What happens if you don’t pass all of the parameters a function expects?

9.10 Programming Exercises

1. Write a function to convert a Celsius temperature to Fahrenheit. The parameters should be the Celsius temperature and the return value the Fahrenheit one. The formula for the conversion is $F = C \times \frac{9}{5} + 32$.
2. Write a function called “getInt” based on the readPositive function above. The function should take the minimum and maximum values the user can put in. It should then ask the user to enter a number until they put in something between that range, and return it when they do
3. Write a function called “printStart”. It should take a list as the first parameter and a number, called “n” as the second parameter. It should print the first “n” items of the list to the screen.
4. Write a program which calculates a few properties of rectangles, each being done with a separate function. The three functions you should write are:

   area This function should take the height and width of the rectangle and return the area of the rectangle. which is the product of the two.

   perimeter This function should take the height and width of the rectangle and return the perimeter, which is the sum of all four sides.

   isSquare This function should take the height and width and return a boolean indicating whether or not the rectangle is also a square. Have the program get the height and width of the rectangle and call each of these functions, printing the results.
5. Write a function which takes a list of numbers as a parameter and returns the smallest value found in the list.

6. Write a function called “filterNegatives”. It should take a list of numbers as the parameter. It should return a new list of numbers with only the positive numbers from the original list. So if you gave it [1, -3, 5, -6, 8, 12, -2] then it would return back [1, 5, 8, 12].

Chapter Summary

- Functions allow us to split programs into independent parts. This makes programs easier to write and test, and keeps code more organized.
- Parameters allow us to pass information into functions, which allows us to customize the way that they work. When we call the function, we supply it with the values it needs.
- Return values allow functions to pass information back to the code that called them. To use the return value, you normally store it into a variable.
- When a variable is created inside of a function, it can only be used inside of that function. We say the scope of that variable is inside that function which is called local. When a variable is created outside of a function, its scope is global which means it can be used anywhere.
- When working on a big program, functions allow us to split the work into smaller units and make solving a big problem more approachable.

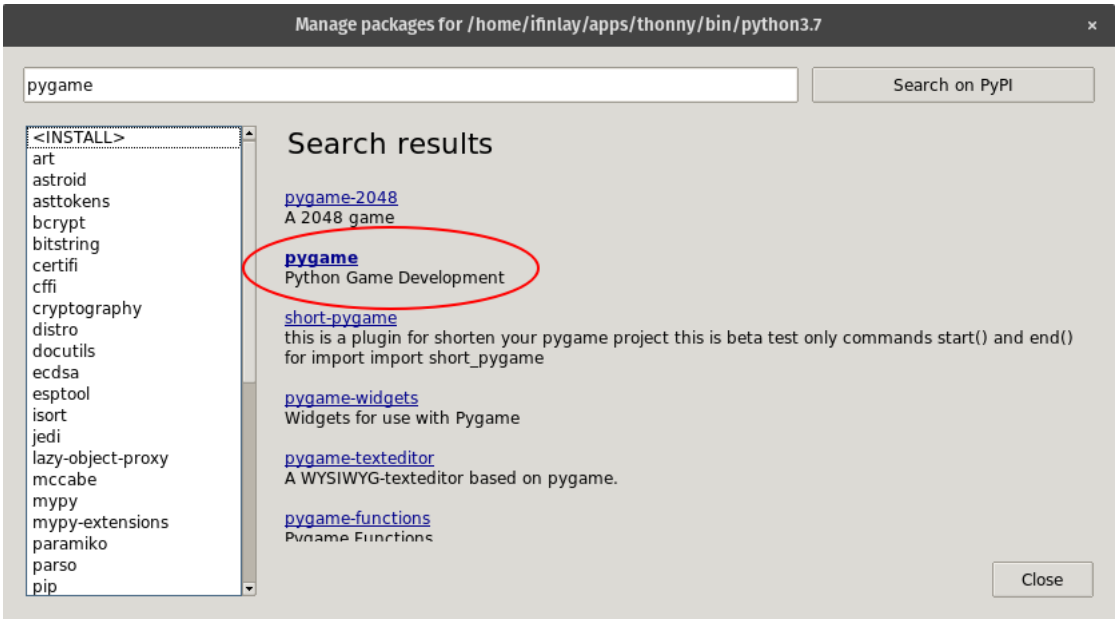

Figure 24: Searching for PyGame

Chapter 10: Graphics

Learning Objectives

- Learn how to create graphics windows
- Understand the coordinate system used in graphics programs
- Learn how colors are created in a computer program
- Be able to create various graphics objects such as points, circles, lines and rectangles
- Learn how to handle mouse and keyboard input in a graphics program

10.1 Installing the PyGame Library

As we talked about in Chapter 8, we can add libraries to Python that other people have built and incorporate them into our own programs. One of these is the *PyGame* library which allows us to make graphical programs. We can use it to make games (hence the name), but it can also be used for other graphical programs such as simulations, or just having more colorful output.

To install it with Thonny, choose “Tools -> Manage Packages” from the main menu. Then search for “pygame” in the window that comes up:

The one we want is just called “PyGame”, so click on that. Next click the Install button:

It will take a moment or two to download. When it is finished, you should have PyGame installed and be ready to write and run graphical programs.

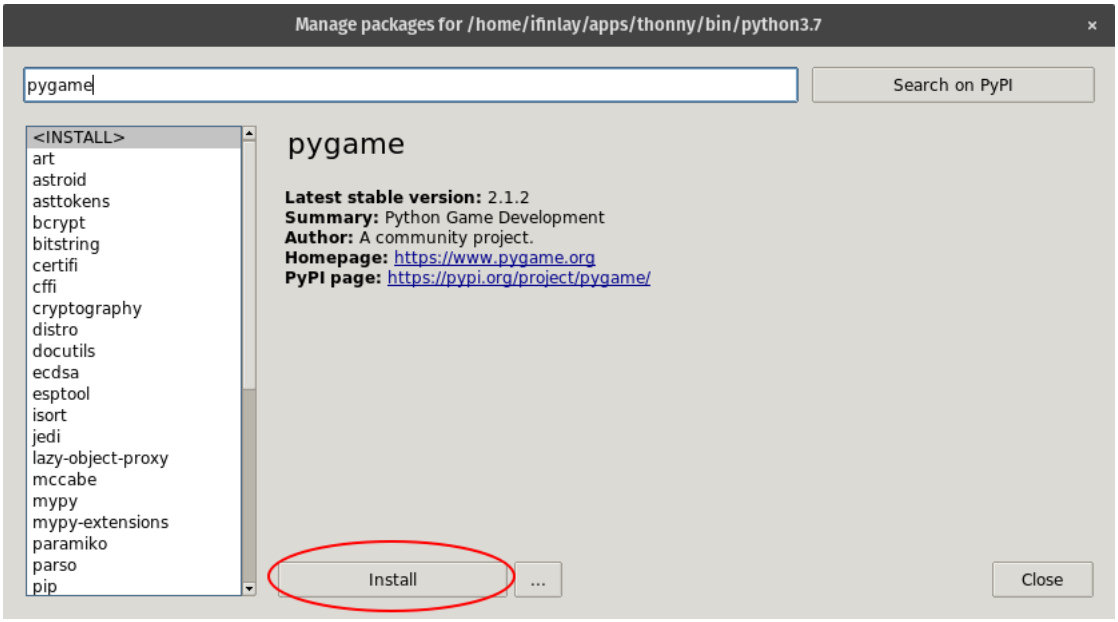

Figure 25: The PyGame Package Page

10.2 PyGame Basics

The first thing we need to do is import the PyGame library. Recall from Chapter 8 that this can be done with the `import` command:

```
import pygame
```

Next, we need to call the PyGame `init()` function which sets up the library, initializing it so we can call upon the other functions:

```
pygame.init()
```

Now we can create a window. All of the programs we have written up to this point have printed their results to the shell window. The shell is purely textual, so to draw shapes, images, and other graphical things, we need a window to put them in. We can create a window with this line of code:

```
window = pygame.display.set_mode([800, 600])
```

This `set_mode` function creates the window for us. Its parameter is a list containing two numbers. The first number is how wide the window should be and the second is how tall it should be.

These numbers refer to **pixels**, which is short for "picture elements". All computer screens are made up of tiny lights that can show up as different colors. These are called pixels. Your computer monitor or laptop screen may have more or less pixels in it. A pretty typical display these days might be 1,920 pixels wide and 1,080 pixels tall. In this case, our 800x600 window takes up around a quarter of the screen.

The window is where all of the things we draw will show up. Before we can start talking about drawing things into the window, we need to discuss how color is represented in computer programs.

10.3 Colors

Just like everything else in computers, colors are ultimately stored as numbers. The way this works is that we have three numbers for every color: one for the amounts of red, green, and blue in the color[1]. Each of these three numbers ranges from 0 to 255. The 255 limit may seem random, but that is the largest number that can fit in one byte. 0 means there is none of that color at all, and 255 means that color is turned up to the max.

For example, we could make a color with 100 for red, 240 for green and 215 for blue. This would be a light aquamarine sort of color like this . The easiest way to explore colors like this is probably to Google "color picker" and play with Google's built in color choosing tool. At the bottom left, you will see the label "RGB" which indicates the three color components. This will let you find the numbers for any color you want[2].

Colors are put into programs with *tuples* which are sort of like lists in Python except they use parentheses instead of square brackets. The other difference is tuples can't be changed once they are created, while lists can. We can assign our aquamarine color into a variable like this:

```
aqua = (100, 240, 215)
```

The numbers are always listed in the order of red, green, and then blue. Here are some definitions for other common colors:

```
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
```

[1] In school you learned that red, yellow and blue are the three primary colors. That's true in a *subtractive* model of color, such as you get when mixing paints together. But there are other models of color, such as the *additive* model wherein you combine colored lights. In this model red, green, and blue are the three primary colors. Shining a red light and a green light together actually makes a yellow light!

[2] With this color scheme, there are actually over 16 million colors we could create. The human eye can only discern about 10 million different colors. So while theoretically there's really no limit on the number of colors, we can practically make any color we can imagine.

```
white = (255, 255, 255)
black = (0, 0, 0)
grey = (128, 128, 128)

yellow = (255, 255, 0)
orange = (255, 128, 0)
pink = (255, 0, 255)
purple = (128, 0, 255)
```

10.4 Our First Graphical Program

Now that we can create colors, we can create a complete program using PyGame. The code for it is given first, then we will discuss it:

```
import pygame

# setup PyGame
pygame.init()

# create a window
window = pygame.display.set_mode([800, 600])

# set some colors
white = (255, 255, 255)
orange = (255, 128, 0)

while True:
    # fill the window with a white color
    window.fill(white)

    # draw an orange circle in the window
    pygame.draw.circle(window, orange, (400, 300), 100)

    # flip the window
    pygame.display.flip()
```

We start by importing the PyGame library, calling the `init()` function for it, and creating our window like before. Next we define two colors for use in the program, a white for the background and an orange for a circle. Of course we can change this to be whatever we want!

The rest of the program takes place in a while loop. Graphics programs pretty much always have a main loop in them that keeps the program running. If we didn't have a

loop, the window would pop up briefly when the program is run and close again when the program ends. Here the while loop is an infinite one. To end this program, you'll have to hit Thonny's "stop" button. Ideally, the program would end when the user hits the 'X' to close the window instead. We'll see how to do that in a little bit.

Inside the loop, we do three things. First we fill the window with a white color by calling `.fill()` on the window and passing the color we want it filled with.

Secondly, we draw an orange circle in the middle of the window. This is done using `pygame.draw.circle()` which takes 4 parameters. The first is what window we want the circle drawn into. Next is the color we want the circle to appear as. After that we give it the coordinates of where the center of the circle should appear. We talk about coordinates in the next section, but (400, 300) is the middle of our 800x600 window. Finally we pass 100 as the radius of the circle.

The third thing we do inside the loop is to "flip" the display. PyGame, like most graphical systems, uses a technique called **page flipping**. This means there are actually *two* graphical areas we create: the one being shown to the user and the one being drawn on. When we draw things like the orange circle, it gets drawn to the "back" display which isn't visible. Then when we are done drawing everything, we flip the displays so the user sees the new scene all at once. This prevents the user from seeing a half-drawn scene which won't look right.

10.5 Coordinate Systems

Before we go much further, we need to talk about the coordinate system used in computer graphics. What does it mean when we put in (400, 300) for the circle's location?

Graphics systems typically use a coordinate system that is different from that used by mathematics. In graphics, the origin, (0, 0) is at the upper left corner of the window. The first coordinate, the X coordinate, increases as we go from left to right. The second, or Y, coordinate increases as we go from top to bottom (which is normally the opposite in math).

This image illustrates the coordinate system:

This image shows the coordinates of each of the four corners of the window. Notice that, like string indices, we start counting at pixel 0 and not pixel 1. Here, the window is 400 by 300 pixels large. That means there are 120,000 pixels.

10.6 Events

Having to end the program through Thonny is not ideal. The user should be able to end the program by clicking the 'X' in the window bar like usual. To do this, we need to look for the quit **event**. An event is just something that might occur while our program is

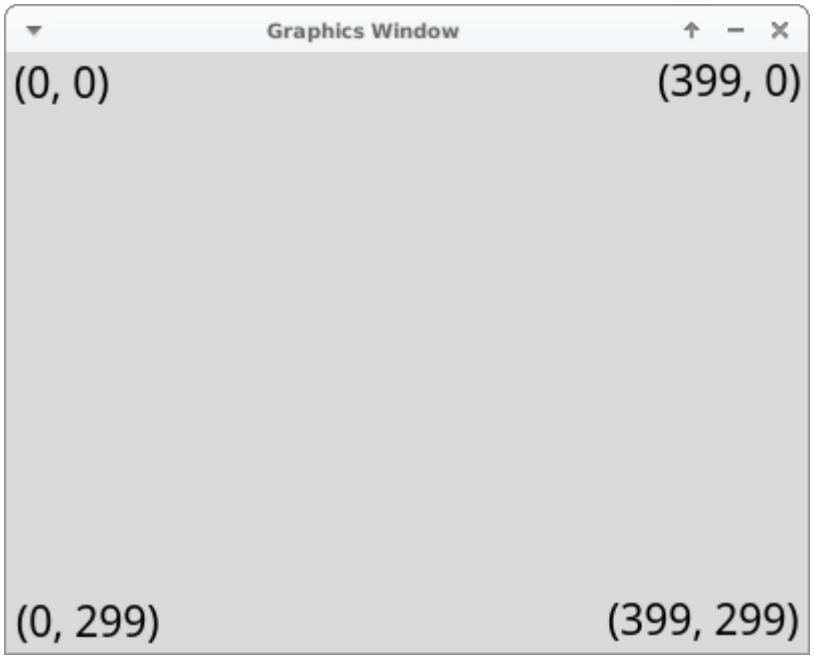

Figure 26: The graphics coordinate system

running, like the user clicking the close button, pressing a key on the keyboard or clicking the mouse.

By checking for these events, we can make our program respond to what the user is doing. PyGame has a function called `pygame.event.get()` which returns back a list of all the events that have occurred since the last time we called it. We can then loop through this list and respond to the events that we get.

This version of the program checks for the quit event. It uses a boolean variable to keep track of whether the program is still running. When the quit event is encountered, this variable is set to False. We also call `pygame.quit()` at the end of the program.

```
import pygame

# setup PyGame
pygame.init()

# create a window
window = pygame.display.set_mode([800, 600])

# set some colors
white = (255, 255, 255)
orange = (255, 128, 0)
```

```
running = True
while running:
    # check for events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # fill the window with a white color
    window.fill(white)

    # draw an orange circle in the window
    pygame.draw.circle(window, orange, (400, 300), 100)

    # flip the window
    pygame.display.flip()

pygame.quit()
```

Now the program should close the window and stop running when the user clicks the close button on the window bar.

10.7 Keyboard and Motion

We can get user input from the keyboard or the mouse using this method as well. To check if a key is pressed, we can detect the `KEYDOWN` event. Then we can check *which* key was pressed by checking `event.key`

For example, we could let the user also quit the program by hitting the Escape key. To do this, we can add an `elif` to the code checking what type of event was received. If it was a `KEYDOWN` event, then we can check which key was pressed. If it was Escape, we can also set `running` to `False`. The full list of keys is available here. The code for implementing this:

```
for event in pygame.event.get():
    # check what type of event it was
    if event.type == pygame.QUIT:
        running = False
    elif event.type == pygame.KEYDOWN:
        # check which key got pressed
        if event.key == pygame.K_ESCAPE:
            running = False
```

Another thing we can do with this is to let the user move the orange circle around with the arrow keys. To do this, we can start by storing the position of the circle in variables. That way we can change those variables when the keys are pressed:

```
# the position of the circle
x = 400
y = 300
```

Then when we draw the circle, we'll use these variables for the position:

```
pygame.draw.circle(window, orange, (x, y), 100)
```

Notice that we still need the extra parentheses because the position is passed as a tuple. Now if we change the x or y variables, the change will affect where the circle gets drawn to. We can do this by adding in code to change these variables when the arrow keys are pressed. The full code for this would look like this:

```
import pygame

# setup PyGame
pygame.init()

# create a window
window = pygame.display.set_mode([800, 600])

# set some colors
white = (255, 255, 255)
orange = (255, 128, 0)

# the position of the circle
x = 400
y = 300

running = True
while running:
    # check for events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            # check which key got pressed
            if event.key == pygame.K_ESCAPE:
                running = False
```

```
            elif event.key == pygame.K_UP:
                y = y - 10
            elif event.key == pygame.K_DOWN:
                y = y + 10
            elif event.key == pygame.K_LEFT:
                x = x - 10
            elif event.key == pygame.K_RIGHT:
                x = x + 10

    # fill the window with a white color
    window.fill(white)

    # draw an orange circle in the window
    pygame.draw.circle(window, orange, (x, y), 100)

    # flip the window
    pygame.display.flip()

pygame.quit()
```

We add or subtract 10 pixels from one of x or y when an arrow key is pressed. Note that when up is pressed we subtract 10 pixels. That's because Y gets smaller towards the top of the window (which can be confusing at first).

If you were using this approach to keyboard input and movement to make something like a game, it would not really be ideal. Rather than have to press they right key over and over to move the circle to the right, we might like to *hold* the right key to have it continually move right.

To do this, we will need to look for both the key pressed and key released events. When an arrow key is pressed, we can set the circle in motion. When the arrow key is released we stop its motion. To this end, we'll create two new variables for the circle. One keeps track of its speed in the x direction and the other is the speed in the y direction. These start off at 0 to mean the circle isn't moving to start off.

```
xspeed = 0
yspeed = 0
```

We'll also make a variable for how fast the circle goes when it is moving:

```
movementSpeed = 1
```

Next we will check for both types of key events (presses and releases) and set the `xspeed` and `yspeed` variables based on that:

```
# a key was pressed down
elif event.type == pygame.KEYDOWN:
    if event.key == pygame.K_ESCAPE:
        running = False
    elif event.key == pygame.K_UP:
        yspeed = -movementSpeed
    elif event.key == pygame.K_DOWN:
        yspeed = movementSpeed
    elif event.key == pygame.K_LEFT:
        xspeed = -movementSpeed
    elif event.key == pygame.K_RIGHT:
        xspeed = movementSpeed

# a key was resleased
elif event.type == pygame.KEYUP:
    if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
        yspeed = 0
    elif event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
        xspeed = 0
```

When an arrow key is pressed down, we set the `xspeed` or `yspeed` variable to either positive or negative `movementSpeed`. Which one depends on which key it was. For the right arrow key, we want it to move right so we set `xspeed` to positive `movementSpeed` which will indicate the circle moves up in the X coordinate.

When an arrow key is released, we set the `xspeed` to 0 when it was the left or right keys, which stop its side to side movement. And if the up or down keys are released it stops moving up or down, so we set the veritcal `yspeed` to 0.

Now the only thing to do is to take the speeds and use them to change the position of the circle. We can do that with this code:

```
# move the circle
x = x + xspeed
y = y + yspeed
```

This adds the speed into each of the two coordinates. So the way this works is the user presses an arrow key, let's say the right key. This sets `xspeed` to positive 1. Every time through the loop, the 1 is added into `x`. So each time through it moves 1 pixel to the right. When the user releases the right key, `xspeed` is set back to 0. Then the `x` coordinate will no longer have anything added to it, so the circle stops.

The full code for this example follows:

```
import pygame

# setup PyGame
pygame.init()

# create a window
window = pygame.display.set_mode([800, 600])

# set some colors
white = (255, 255, 255)
orange = (255, 128, 0)

# the position of the circle
x = 400
y = 300

# the current speed of the circle
xspeed = 0
yspeed = 0

# how fast the circle can go when moving
movementSpeed = 1

running = True
while running:
    # check for events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        # a key was pressed down
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                running = False
            elif event.key == pygame.K_UP:
                yspeed = -movementSpeed
            elif event.key == pygame.K_DOWN:
                yspeed = movementSpeed
            elif event.key == pygame.K_LEFT:
                xspeed = -movementSpeed
            elif event.key == pygame.K_RIGHT:
                xspeed = movementSpeed
```

```
        # a key was resleased
        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_UP or event.key == pygame.K_DOWN:
                yspeed = 0
            elif event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
                xspeed = 0

    # move the circle
    x = x + xspeed
    y = y + yspeed

    # fill the window with a white color
    window.fill(white)

    # draw an orange circle in the window
    pygame.draw.circle(window, orange, (x, y), 100)

    # flip the window
    pygame.display.flip()

pygame.quit()
```

10.8 Drawing More Things

Besides just circles, we can draw other shapes as well. The following program draws a number of shapes to a window:

```
import pygame
import math

pygame.init()

window = pygame.display.set_mode([800, 600])

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Fill the background with light grey
    window.fill((240, 240, 240))
```

```
    # a red rectangle
    pygame.draw.rect(window, (255, 0, 0), pygame.Rect(50, 50, 150, 100))

    # a green ellipse
    pygame.draw.ellipse(window, (0, 255, 0), pygame.Rect(400, 100, 300, 200))

    # a blue polygon
    pygame.draw.polygon(window, (0, 0, 255), [(100, 300), (200, 300), (250, 3

    # an orange line
    pygame.draw.line(window, (255, 128, 0), (100, 500), (400, 550), 4)

    # a purple arc
    pygame.draw.arc(window, (150, 0, 180), pygame.Rect(500, 400, 100, 100), 0

    # Flip the display
    pygame.display.flip()

# Done! Time to quit.
pygame.quit()
```

The rectangle is drawn with:

```
pygame.draw.rect(window, (255, 0, 0), pygame.Rect(50, 50, 150, 100))
```

Just like when drawing the circle, the first parameter is the window to draw the rectangle to and the second is the color it should be. The third parameter is the rectangle to draw. The `pygame.Rect` function creates a rectangle given four parameters. They are:

1. x coordinate
2. y coordinate
3. width
4. height

Drawing an ellipse is done in a similar way. We used this code to draw a green ellipse:

```
pygame.draw.ellipse(window, (0, 255, 0), pygame.Rect(400, 100, 300, 200))
```

The parameters here mean the exact same thing. The only difference is that the ellipse will be drawn inside the rectangle shape which is given.

Drawing a polygon is done with this line of code:

```
pygame.draw.polygon(window, (0, 0, 255), [(100, 300), (200, 300), (250, 350)
```

Again the first two things are the window to draw to and the color. Then the function takes a list of points which give the coordinates of the points in the polygon.

A straight line can be drawn with code like this:

```
pygame.draw.line(window, (255, 128, 0), (100, 500), (400, 550), 4)
```

After passing the window and the color for it, we pass two points which give the ending points for the line. The last parameter is optional and gives the thickness of the line.

Finally we can draw a curved line, called an arc with:

```
pygame.draw.arc(window, (150, 0, 180), pygame.Rect(500, 400, 100, 100), 0, ma
```

We pass the window and the color first. Next comes a rectangle which bounds where the arc will be drawn (just like an ellipse). The last two parameters are angles measured in radians. These two angles specify the start and end angle for the arc.

10.9 Drawing Images

In addition to drawing these shapes, we can also load images from a file and draw them to the window too. The first step in doing this is to load the image file from your computer. We can do this with the `pygame.image.load` function:

```
picture = pygame.image.load("campus.png")
```

For this to work, the image file has to be in the same folder as your Python program is saved to. PyGame can load several types of images including PNG and JPG. Once the image is loaded into a variable, we can draw it using code like this:

```
window.blit(picture, (0, 0))
```

Instead of passing the window into a draw method, like for drawing shapes, here we call `blit` on the window itself[3]. The second parameter is where in the window the upper-left corner of the image should appear. Here we pass the origin (0, 0) so that the picture fills the whole window. You can download campus.png to run the program.

10.10 Comprehension Questions

1. How do computers represent color?
2. What is a tuple and how is it different from a list?

[3]The term "blit" is an old phrase in graphics and game programming. Its origin is an acronym for BLock Information Transfer. To blit something is to transfer it quickly from one part of memory to another. When you draw an image, you copy the image data onto the screen.

3. What does “page flipping” refer to?
4. Where is the origin point in a computer graphics window?
5. What is a pixel?
6. How does PyGame handle events like the user clicking a mouse or quitting?

10.11 Programming Exercises

1. Write a program which draws your initials on the screen using lines. If you have a “curvy” initial, you don’t have to make it appear round. For example, an S can be made out of just three lines. But if, you want to be ambitious, you can use the pygame.draw.arc function to make curves.
2. Write a program where you draw a randomly colored rectangle to the screen.
3. Modify the last program so that whenever the user clicks the rectangle it changes color.
4. Write a program in which ball moves around the screen. When it hits the edge, you should have it bounce back. This can be done by keeping track of four variables: the ball’s X and Y coordinates, and the *change* in the ball’s X and Y coordinates, which we can call DX and DY. Each time through the main loop, add the DX into the X and DY into the Y. If we need the ball to move right DX will be positive. If it needs to bounce back to the left, we set DX to be negative.
5. Expand the previous program to be a simple Pong game, by adding two rectangles which represent the player and computer paddles. One paddle should be moved when the up and down keys are pressed. The other can move however you like: up and down forever, randomly, or by tracking the ball.