2: Chapter 2

2.1: Programming as a means of making systems

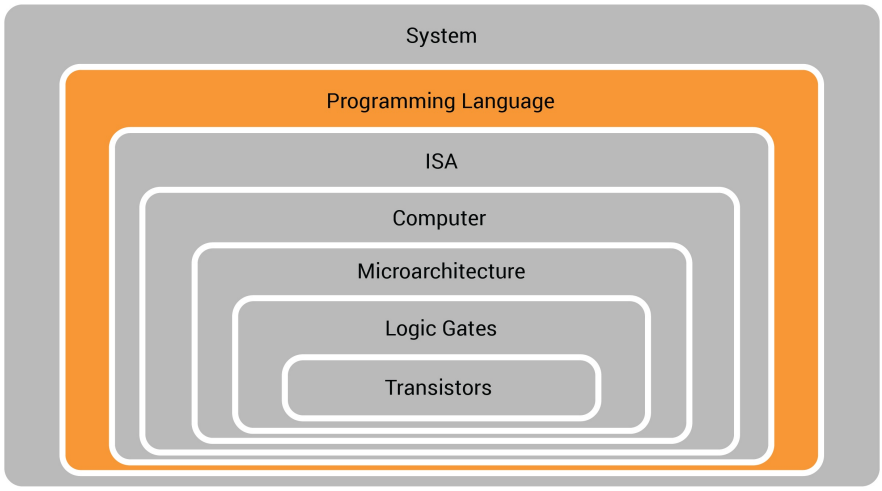

Figure 2.1.1: We are now going to study programming at a high level (low-level programming will happen at the ISA layer).

Let us start by situating ourselves within the tower of abstractions we laid out in the previous chapter: We are trying to understand computers--that is, devices that perform arithmetic, logical, memory, and I/O operations. Specifically, we are trying to understand two things:

- How to program a computer
- How to build a computer

Recall we said that to program such a device, we would provide it with sequences of commands from its ISA. But because the ISA provides only basic commands, we prefer to write commands in a more human-friendly form and then have a mechanical way of translating those commands into the basic ISA commands that the computer understands. A human-friendly way of specifying what the computer should do is called a **high-level programming language**.

Warning:

The computer doesn't itself understand the programming language, as mentioned. However at this level, we are actually not interacting directly with the computer at all, but rather with the compiler--the mechanism for translating a high-level programming language into ISA operations for the computer's consumption.

Nevertheless, we may sometimes in this chapter speak imprecisely by talking about the computer itself actually understanding the programming language, when really we mean the compiler--the abstraction currently sitting between us and the computer--is helping the computer understand it. That is, we are describing the interface of the compiler, and so from this perspective, programming the compiler looks like programming the computer.

In this chapter, we'll discuss the concept of a program and a programming language in general in 2.2, as well as strategies for writing effective programs. Then, starting in 2.3, we'll introduce an actual programming language: Python, and will walk through the process of writing Python programs to solve specified problems.

Most generally, **programming** is expressing to the computer:

- A sequence of steps for it to perform--also known as a **program**
- specified in way that the computer understands--also known as a **programming language**

When the computer subsequently executes the steps you gave it, it is said to be **running** or **executing** the program.

In general, you approach a computer with a human-level problem: You want to find out how much you spend on cookies each month, or maybe you want to make a Pacman clone. However a computer has no understanding of such things--It does not know what a cookie is and does not come with a command for "play pacman". But it does (at least, at the level we're thinking about it currently) provide you with a programming language--that is, you can tell it to do certain specific operations that are allowed by the language. Beyond that, a computer is a machine--if you give it a sequence of

commands in a programming language, it very simply does everything listed in that program, and nothing else. It has no understanding of--let alone sympathy for--its owner's 'intent'.

```
A classic parable--"If your husband were a computer":

"Honey, while you're out, could you stop by the store and buy one
carton of milk?  And if they have eggs, get 6."

A short time later the husband comes back with 6 cartons of milk.

The wife is perplexed.  "Why did you buy 6 cartons of milk?"

"They had eggs!"
```

This is the gap that a programmer has to bridge: Taking a high-level problem coming from the real world, and turning it into a sequence of steps that are allowed in a given programming language.

In this chapter, we'll do three things:

1. In 2.2, we'll discuss the first step in crossing this gap: Algorithms. An algorithm is a human-readable description of computer-friendly steps intended to solve a high-level problem. There are a few considerations that come into play when designing an algorithm--how "good" is a particular algorithm for solving a problem, and what does "good" even mean for algorithms.
2. Then, in 2.3, and 2.4, we'll take the next step: Once we've got the algorithm--the computer-friendly steps that solve the problem--we need the program. That is, we need to write those steps in a computer-friendly way. So we'll introduce a specific programming language--Python, talking only about the rules about what instructions are allowed in a valid Python program (also known as Python's syntax, and discussed in 2.4), and what specific valid instructions actually do (Python's semantics, discussed in 2.5).
3. Finally, in 2.5 and 2.6, we'll go through examples of ever-increasing complexity that put this whole process together: Starting with a general problem, constructing an algorithm that solves it in a computer's terms, and then turning that algorithm into a program so that the computer can find the actual solutions for us.

2.2: Algorithms: Expressing solutions in human language, but in terms of the computer's primitives

Summary:

The first step in programming a computer with the solution to some given problem is to figure out what are the steps it will need to perform. Each step must consist, fundamentally, of only operations of the four basic sorts that a computer can perform.

It is at this point--while writing the algorithm and in particular before writing any actual computer code--that we can assess our solution's efficiency as well as identify in what situations our solution would not work. This can consist either of rare special cases of the problem in question--called edge-cases, or of simply scaling up the size of the problem until our solution wouldn't be able to handle it within a reasonable amount of time.

Suppose you've written a computer game, which has 3 players, each of whom has a score. At the game's end, you'd like to determine a winner. To a human, looking at the three scores and picking out the largest one is completely straightforward. After all, "pick the largest of three numbers" seems like an atomic operation in its own right--one which we never have to think about breaking down into a more precise, step-by-step procedure.

However such a step-by-step procedure is exactly what a computer needs in order to accomplish this same goal. It doesn't have a "find the largest number" operation built-in. Instead, its available operations are basic: Things like "add two numbers", "compare two numbers", and the like.

First of all, we know the computer can store numbers. In fact, as we will see later, you can store numbers under names of your choosing, so suppose we've stored the three players' scores under the names score1, score2, and score3 respectively. Then a procedure for determining the winner might look like:

```
If score1 is greater than score2 and score1 is also
greater than score3: Display the text: "PLAYER 1 WINS!"

If not, then if score2 is greater than score1 and score2 is also
greater than score3: Display the text: "PLAYER 2 WINS!"

If not, then if score3 is greater than score1 and score3 is also
greater than score2: Display the text: "PLAYER 3 WINS!"
```

An algorithm for determining the winner of a 3-player game given the scores

This procedure can now believably be broken down into simple arithmetic operations: Comparing two numbers is an operation the computer can do, and the simple logic operation "if both this and that are true" is another, so indeed, the above does comprise a computer-friendly procedure. It is not yet a program because it was still written in English, rather than in a language the computer understands, but it is a complete description of computer-friendly steps for accomplishing what we need done. Recall that such a precisely specified procedure we called an algorithm.

There are two things that might give you pause about this algorithm as written: namely, it's handling of ties, and its apparent inefficiency.

2.2.1: Edge-cases

Summary: Edge-cases are situations that are unusual or unexpected, but may nonetheless occur. A simple example of this is a tied score in a game. Sloppy algorithms may work under normal circumstances but break down when faced with certain edge-cases.

Definitions: edge-case

When you sit down to write an algorithm, you should have some understanding of what you can and cannot expect of the inputs. For instance, try to follow through the above algorithm if

```
score1 = 10, score2 = 10, score3 = 7
```

It doesn't print anything at all! Such inputs are sometimes called **corner-cases** or **edge-cases**. They are not generally what your algorithm will have to deal with, but they can still happen in theory, and when they do, they can give you a rather bad day if you haven't accounted for them. Indeed, failing to deal with even a single sneaky corner-case is a common cause of serious problems in real-world programming.

For now, for simplicity, we'll assume the rest of the game is set up so that any tie is resolved with a tie-breaker, so that in fact, by the time the algorithm above is run, there are no remaining ties.

This is one perfectly valid way of dealing with corner-cases: Just add some further steps that ensure that they can honestly never appear. This decision comes from asking the higher-level question of how we in fact want the game to work. If we really don't want there to be ties in the end (e.g., we always make it so there are tiebreaker rounds until all ties are resolved), then we may do as we suggested. But if we actually think ties should be a valid result in our game, then we would have to expand our score-comparison algorithm to deal with this and detect them.

2.2.2: Efficiency

Summary: Simply having an algorithm that solves the given problem isn't the end of the story. We can ask whether there is a more "efficient" algorithm, where "efficient" may mean anything from "takes less time to finish in every situation" to "requires more storage space to finish in some cases, but requires way less storage in the few cases we actually expect to run into".

Definitions: efficiency

There are two things about this algorithm's inefficiency that might offend your sensibilities: We're comparing three numbers, so it seems like the number of actual "compare these two numbers" operations required shouldn't be too many. But indeed, if score3 happens to be the largest number, then the above algorithm will perform this operation 6 times:

We first compared score1 to score2 and then to score3. Since score1 wasn't the largest, we then compared score2 to score1 and to score3. Since score2 was also not the largest, we finally compared score3 to score1 and to score2, and saw that score3 was indeed larger than the other two. This intuitively seems a lot more work than your brain is likely doing when you simply look at three numbers and spot the largest by eye, so it might be worth looking for an improved algorithm that will involve less work from the computer.

By contrast, consider the following algorithm:

```
If score1 > score2, then do the following: If score1 > score3,
Display "Player 1 wins", or else display "Player 3 wins!"

Otherwise, do the following: If score2 > score3, Display "Player 2
wins", or else display "Player 3 wins!"
```

A slightly more efficient algorithm for comparing three scores

This algorithm has three comparisons written into it, but it will only ever perform two of them regardless of what the scores are, making it three times as fast as the original algorithm, at least in the event that player 3 wins.

(As an aside: It is worth revisiting the question of edge-cases with this improved algorithm, and noting that while our previous algorithm printed nothing in case of a tie, this one may actively print the wrong thing--try it and see!)

This is an example of analyzing an algorithm's **efficiency**--that is, analyzing how much of a given resource an algorithm may require under various circumstances. That definition is intentionally quite vague: "Resources" may refer to time, or memory, or communication bandwidth, or money, or any other thing required by the algorithm that we may wish to study. "Various circumstances" may refer to asking about how the algorithm behaves in the most common case, or in the average case, or in the worst possible case.

Which notion of efficiency you consider in your algorithm design depends on circumstances usually outside the original problem statement--do you have lots of money and no time? An algorithm that is highly inefficient in its use of memory even in the common case, but very efficient in its use of time, even in the worst case, may be what you want.

The conclusion that we arrived at in our comparison of the two "who won?" algorithms, stated precisely, would be that they require the same amount of memory (both have to store exactly three values), but that in the worst case, the first algorithm would require six comparisons to be performed whereas the second would require only two.

Warning:

Even our mention of "time" as a resource was a little imprecise. Remember that when a computer runs a program, what it is actually doing is running ISA operations. Generally, each ISA operation takes a fixed number of microseconds, so to study how long a program takes, we technically have to understand what ISA operations it corresponds to and ask how many of these will be executed.

Counting the number of steps or arithmetic operations used by the algorithm is often a useful proxy for this more precise notion, but as you get more involved with high-level languages, the more disassociated "number of steps in your program" becomes from "number of ISA operations involved". If we wish to make our programs more efficient, then, it can be crucial to understand them at the ISA level. We will see examples of this in chapter 5.

2.2.3: Scalability

Summary: If we take our algorithm that solves a problem and become very successful, sometimes we end up facing a much bigger version of our original problem. Maybe our game only had three players to begin with, but what if it becomes popular and now has a million? This is another measure of an algorithm--how well it behaves when faced with larger and larger versions of the problem it was meant to solve. In other words, how well it "scales."

Definitions: scalability

There is a related but more subtle concern about the original algorithm, which is: Suppose instead we had 10 players? Or 100? What would it look like then?

The structure of the algorithm was:

```
First do all the comparisons to see if player 1 won.

If player 1 didn't win, then do all the comparisons to see if player 2
won.

etc.
```

So in the case of 10 players, it looks like:

```
If score1 > score2 and score1 > score3 and ... and score1 >
score10, then display "Player 1 wins!"

Otherwise, if score2 > score1 and score2 > score3 and ...

..............
```

Before, if player 3 won, we had to do 6 whole comparisons to find this out. Now there are 10 players who might have won, and testing each requires 9 comparisons, so we get a total of 90 comparisons that might have to happen before we can determine a winner! In general, for N players, we have N*(N-1) comparisons--so for 100 players, that's 9900 comparisons!

These numbers might seem notionally problematic, but it is possible to assess the actual cost in time and energy to say something objective: An average computer might say it is "clocked at a 1 GHz". That means it can do a one billion basic operations every second. If you're running a massively multiplayer online game with 10000 players, then using this algorithm, each winner calculation will take at least 99990000, or roughly one hundred million operations. With a 1 GHz computer, we can thus only compute a winner around 10 times per second. And if we want to constantly be displaying the current leader on the screen as the game progresses, this means the game can only update the display 10 times per second at most (i.e., run the game at 10 "frames per second"). But the illusion of smoothness in gameplay comes from updating the picture on the screen at the very least 30 times per second, so limiting a game to 10 frames per second would result in noticeable stuttering and could easily render the game unplayable.

This is an analysis of **scalability**--that is, of questions about how well the algorithm performs when we apply it to scaled-up versions of the same problem, and about how big the problem can become before we are forced to modify our algorithm to solve it effectively.

Whether scalability is a concern for your particular application is again more a question of your philosophy, ambitions, and external constraints. If you're really honestly only ever going to have three players, then reducing 6 operations to 2 on a computer that can perform billions every second, and putting a lot of effort in to make your program works for thousands of people at a time when that scale is very far in your project's future may or may not be the best use of your time right at the start of such a project.

2.3: Python: Expressing algorithms using a programming language

Now we come to the second step in using a computer to solve a problem. We have the algorithm describing the steps the computer can take to solve it. Now we need to write those steps in a language the computer understands--a **programming language**. In this section, we'll be introducing the rules governing a specific programming language--Python.

In this section, we'll provide a basic introduction to the language and get you running your first program. In the subsequent sections, we'll introduce the **syntax**--that is, what are the rules that determine whether the instructions in a program are valid, and the **semantics**-- that is, what do the various instructions actually do.

In this chapter alone, we will discuss only a subset of Python, which will be sufficient to accomplish many goals, at the price of being less powerful than the whole Python language actually us. In chapter 3 we shall see some of Python's advanced features and how they simplify some of the tasks that will be slightly cumbersome in this chapter.

Finally, as with most computer-related things, you can find more information on any specific topic with a studied use of your favorite search engine. If you find yourself confused about while loops in Python (whatever those are), punching 'python while loop' or even 'python while loop tutorial' into Google will often have an enlightening effect.

2.3.1: Running Python

Summary: In this book, some examples will be runnable in a simulator, which will be included in-line. Press run to start running the program. Press "step" to advance the program by one step. Change the number in the box next to "step" to make the "step" button advance the program by more steps. The link on the left sidebar will let you run a more complete version of the simulator.

Python is a real-world programming language with many uses in the wild. In this chapter, however, since we're focusing on a very small part of the Python language, we will not run any code on a real computer. Rather, we will use the web browser to simulate the code being run. This way, we will get to watch what happens, as if in slow-motion, as the code is executed, rather than it all happening at once and trying to work out or imagine what happened afterward.

We'll explain the use of the simulator interface and what you can do with it throughout this section, but if you're the sort of person who would rather play the game without first reading the rules, a few example programs and challenges are available by clicking the "Python" link on the sidebar, or by clicking here.

To use the simulator, you write your program in the text box on the right. When ready, click the "run" button above it, whereupon the simulation will be prepared. At this point, no changes to the code can be made until you press the "reset" button. To execute a single line of code, click the "step" button. If you want to run more than one step at a time, simply enter how many steps you want the step button to advance into the input box to the right of it.

As you step through the code, the line that will be executed the next time you step is highlighted in the program text input by a red arrow that will appear next to that line in the left-hand margin.

While you can write and test any program in this simulator, some programs right here in the main text will be embedded in a smaller version of the simulator so you can step through the examples and see how they work and edit them as you please to see how various changes affect the behavior. You can always restore the original example program by pressing the "original" button.

2.3.2: A first Python example

Summary: We introduce Python by describing a classical mathematics problem, writing an algorithm, and then writing the corresponding Python code for that algorithm.

Definitions: modulus

There is a famous mathematical problem--known as the 3x+1 conjecture--that goes as follows:

1. Start with any positive integer.
2. If it is even, divide it by 2. If odd, triple it and add one.
3. If the result is 1, stop. Otherwise, go back to the previous step.

For example, we start with 5:

Current value	What do we do next?
5	5 is odd, so triple and add 1 to get...
16	16 is even, so divide by 2
8	8 is even, so divide by 2
4	4 is even, so divide by 2
2	2 is even, so divide by 2
1	

The conjecture is that we always get down to 1, regardless of our start input. We just verified the conjecture starting with x = 5, but what about other values of x? We can start doing some, and it's

pretty easy: For x = 25, for example, it goes:

```
25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40,
20, 10, 5, 16, 8, 4, 2, 1
```

It takes a little while, but we can sort of do it by hand. However, for x = 27, if we start, we get the sequence:

```
27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161,
484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466,
233, 700, 350, 175, 526, 263, 790, 395, 1186, ......
```

Which looks like it might possibly continue with its fitful growth forever. If only we could get a computer to do the tedious work of running the algorithm for us...

As described before, the first step in writing a program is to come up with an algorithm that expresses the procedure in terms of the kinds of operations a computer can do.

```
1. Store the value 27 in a slot called x (Storage)
2. See if x is equal to 1, and as long as it is not, execute the
   following steps (3-5) and come back to this step:  (Arithmetic and branching)
   3. If x is even, compute x/2 and store that in x (Arithmetic, storage, and branching)
   4. Otherwise, compute 3*x+1 and store that in x (Arithmetic, storage, and branching)
   5. Print out the current value of x (I/O)
6. Print out the text "Done" (I/O)
```

We call step 3 ("If x is even...") partly arithmetic because we compute x/2, and partly storage because we have to store that somewhere, and partly branching because it's choosing to either execute a computation or to skip it depending on a certain condition. However, testing that condition--whether x is even--doesn't seem like an arithmetic operation in the sense of addition, subtraction, multiplication, and division.

The key is that division is actually three separate operations: Just as in grade school, before you learned to think of 13 divided by 4 as 3.25, you may have instead learned it as "13 divided by 4 is 3 with a remainder of 1". That is, 4 goes into 13 three times, but then there's one left over.

In Python, the "/" operation represents the decimal division--so 13 / 4 is 3.25. The "%" operation performs integer division and gives back not the quotient, but the remainder. So 13 % 4 is 1, because 4 goes 3 times into 13, with a remainder of 1. This operator is known as the **modulus** operator.

> **Aside:**
>
> In case you're wondering where this word "modulus" comes from--it actually comes from an English word "modulo", meaning roughly "supposing we ignore". For example, "Bob and Sam basically get along quite well modulo matters pertaining to cheese". The reason for its use here is that 13 = 1 + 4 + 4 + 4, so modulo 4s--i.e. if we ignore the 4s, 13 is the same as 1. I.e. "13 modulo 4 is 1", written in computer lingo as 13 % 4 = 1.

The modulus operator gives us a way to test whether a number is even: "Even" means that when you divide by 2, there is a remainder of 0. So to test whether x is even, we'll compute x % 2. If this is 0, then x is even. Otherwise, x is odd. For simplicity, we'll break out this computation into a separate step in the algorithm:

```
Store the value 27 in a slot called x (Storage)
See if x is equal to 1, and as long as it is not, execute the
following steps and come back to this step:  (Arithmetic and branching)
   Compute x % 2 and store it in a slot called y (Arithmetic and storage)
   If y is 0, do the following steps: (Branching)
      Compute x/2 and store that in x (Arithmetic and storage)
   Otherwise, do the following steps: (Branching)
      Compute 3*x+1 and store that in x (Arithmetic and storage)
   Print out the current value of x (I/O)
Print out the text "Done" (I/O)
```

Now, we said that programming is taking an algorithm and expressing it in a computer's terms. To this end, we shall take the above algorithm and write a Python program that runs it (don't worry if you don't understand it yet--all will be explained in time):

```
x = 27
while(x != 1):
   y = x % 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print(x)
```

```
print("Done")
```

Before we jump into studying this program, let's run it to see what happens: To start the program, press the "run" button. Then to advance to the next step in the program, one step at a time, press the "step" button. It will of course take a long time of doing this to run the program through to completion, so we can use the box next to the step button to say how many steps we want the step button to advance us. If you want to see the end result, put something large like 1000 in and wait and behold the output.

Now, if you compare this listing to the English language algorithm, you notice many similarities, but the Python syntax is more rigid: The computer runs this program by the code one line at a time. After one line is run, it moves on to the next line, unless some instruction tells it to do otherwise.

This is a fundamental rule, and should be internalized so you can get used to understanding programs mechanically: Programs are run one line at a time. In particular, understanding a program entails not so much looking at it as a whole, but understanding what each line does individually. Generally, once one line's instruction is completed, the next instruction to be run is the one on the following line, except in certain situations in which an instruction can tell the program to jump to a different line. For example, the block:

```
x = 27
while(x != 1):
   y = x % 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print(x)
print("Done!")
```

means that all the indented instructions should be repeated as long as x is not 1 (which is written in Python as x != 1). In particular, when we get to the line

```
x = 27
while(x != 1):
   y = x % 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print(x)
print("Done!")
```

if x still isn't 1, then we don't go to the next line (which is the end of the program!), but instead jump back to the beginning of the indented "while" block--specifically, to the line

```
x = 27
while(x != 1):
   y = x % 2
   if(y == 0):
      x = x / 2
   else:
      x = 3*x+1
   print(x)
print("Done!")
```

There is a second difference between the English description of the algorithm and its programmatic realization, namely that whereas when describing the algorithm, we had a notion of "the current value of the number", which in the example took on the values "5, 16, 8, 4, 2, 1". In the program, we have to give the value a name. In this program, we have chosen the name "x". Such a "named value" is called a variable. Variables are used whenever we want to store the value of a computation for later use.

In this program, we have two variables, called "x" and "y", whose stored values get updated as the program progresses.

Using the rule that a program is run one line at a time, we can follow along with the program, one line at a time, to see:

1. Which lines get executed
2. What values the variables are storing

Current line	Why we went to this line	Variables after current line runs	Output so far
`x = 27`	It was the next line	`x = 27`	
`while(x != 1):`	It was the next line	`x = 27`	
`   y = x % 2`	x was not 1	`x = 27, y = 1`	

`if(y == 0):`	It was the next line	`x = 27, y = 1`	
`x = 3*x+1`	y was not 0	`x = 82, y = 1`	
`print(x)`	It was the next line	`x = 82, y = 1`	82
`y = x % 2`	We reached the end of the while block, but x was not 1, so we go back to the start	`x = 82, y = 0`	82
`if(y == 0):`	It was the next line	`x = 82, y = 0`	82
`x = x / 2`	y was 0	`x = 41, y = 0`	82
`print(x)`	We skip the else block because the if condition was true	`x = 41, y = 0`	82 41
`y = x % 2`	We reached the end of the while block, but x was not 1, so we go back to the start	`x = 41, y = 1`	82 41
`if(y == 0):`	It was the next line	`x = 41, y = 1`	82 41
`x = 3*x+1`	y was not 0	`x = 124, y = 1`	82 41
`print(x)`	It was the next line	`x = 124, y = 1`	82 41 124

2.3.3: Python language summary

We now present a brief summary of all of Python as we'll use it in this chapter. This includes a specification of the syntax--the rules that define what is allowed as Python code, and the semantics--what the various valid constructs do.

The way this specification works is recursive: For example, we will first specify that Python code is run one line at a time. So to do something in Python, we have to write a sequence of lines. So then the question is: What is a line? In the table below, we see that a line is either an 'assignment line', 'print line', or a 'flow-control' line.

We haven't said what any of these are yet, but behold, they show up later in the table! The table tells us that assignment lines are used for computation and for setting variables, print lines are for printing things, and flow-control lines are for deciding which line to execute next.

So if we now decide that what we want is to store some number in a variable, then we look at the specification for an assignment line, since that seems the thing for the job. It says that an assignment line looks like:

```
variable name = expression
```

We haven't said what variable names or expressions are, but they are also described in the table!

Looking further down to variable names, we see that a variable name can be anything with letters, numbers, and underscores, but not starting with a number, and not a Python keyword. So we can decide to call our variable, say, `H3LL0_th3re_`.

But what about the expression bit? Expressions are defined in the table as any valid combination of numbers, strings, and variables using the basic operations and parentheses for grouping. So if we want to store the value 89*90, we can do this like:

```
H3LL0_th3re_ = 89*90
```

And the fact that we have followed the table tells us that this is a valid assignment line which will store the value of the expression 89*90--i.e., the number 8010--into a variable called H3LL0_th3re_.

2.4: A detailed explanation of Python

Summary: The above table represents nearly all of the features of Python that we shall use in this chapter. The table does not explain how to think about or when to use these in practice. This section will get more into that, as well as a small number of odds and ends that don't come up often and so were left out of the reference table.

We now give a more detailed explanation of the Python language elements we introduced above. For example, in addition to providing greater detail about what they are, we provide examples and give examples of when to use them.

2.4.1: Python syntax

Summary: We summarize the syntax again, but this time in a form that is amenable to mechanically walking through a program and confirming that it follows all the rules: The syntax tree.

Definitions: syntax tree

Before we dive too deep into explaining Python, we reiterate that while the most interesting part of any programming task is always deciding how to match the code to the problem, your code must first always conform to proper Python syntax or it will not do anything at all. Happily, ensuring proper syntax in your Python is very much the easy part, and doesn't even require you to think about or even know what your code does. Much as you can probably tell that the phrase "Begone you ancephelous lummox!" is valid English, whereas "Accismus very kerfuffle the, bromnopnea in" is not, even without knowing the meaning of many of the words involved, you do not need to understand Python semantics in order to tackle its syntax.

To better understand the syntax, we will introduce another way of visualizing it, called a **syntax tree**. This is a visualization that highlights the relationships between the syntax elements. The most fundamental syntax element is a line of code: The program is entirely comprised of lines. But what is a line allowed to be? Well, there are three choices: Assignment lines, print lines, and flow-control lines.

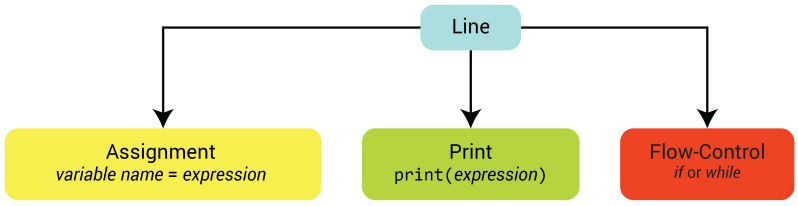

Figure 2.4.1.1: The start of the Python syntax tree

But the description of the assignment lines includes some further terminology that needs to be explained: What is a variable name? What is an expression? And for flow-control lines, what is a test? So to complete the syntax tree, we extend it with the descriptions of these things as well, taken from the table:

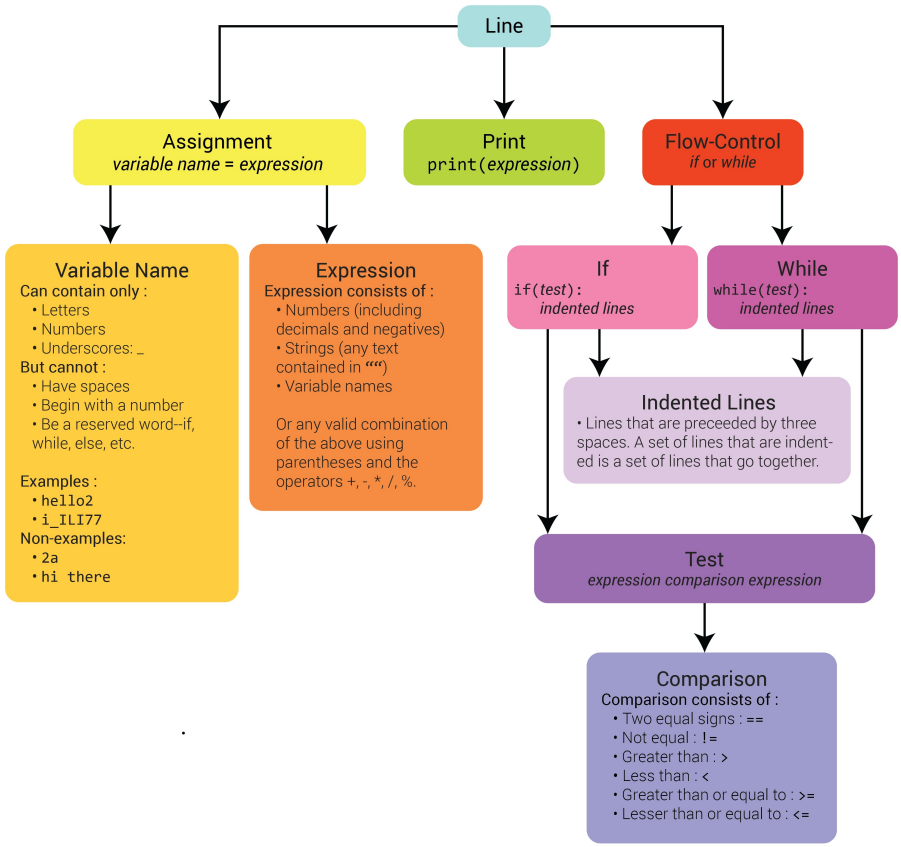

Figure 2.4.1.2: The full syntax tree for Python

To see how this works, let us take an example program:

```
x = 2
if(x < 2):
   print("Hi")
if(x >= 2):
   print("Bye")
x = 5 + x
```

First of all, of course this consists of lines. But further, each line is one of the three types of line:

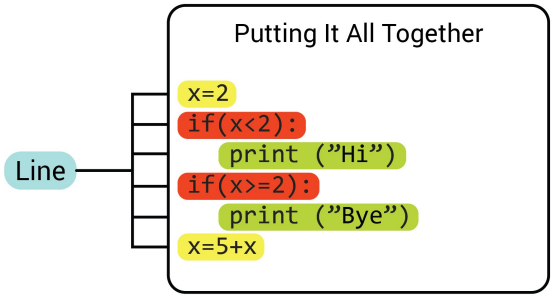

Figure 2.4.1.3: Lines of the program colored by type

And further, for a line like x = 5 + x to be a valid assignment, the first thing must be a valid variable name. "x" indeed satisfies all the rules, so this is OK. Then there must be an "=", which there is. Finally, there must appear a valid expression. "5+x" is a combination of a variable and a number using the "+" operator, so indeed qualifies as a valid expression. We can perform this analysis with the rest of the program, colorizing each syntax element according to its type, and we get:

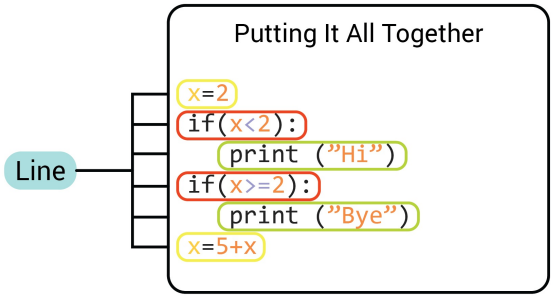

Figure 2.4.1.4: Lines of the program, with sub-elements colorized by type.

We can also play this game with our example program from the previous section:

```
x = 27
while(x != 1):
   y = x % 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print(x)
print("Done!")
```

and check line by line that it is a valid program.

```
x = 27
```

This first line looks like it must be an assignment. That means it has to look like

```
variable name = expression
```

Indeed, looking at the rules for variable names, "x" is a valid variable name. Further, expressions are defined as numbers, strings, variables, or combinations thereof. "27" is a number, and hence a valid expression. So indeed, this is a valid assignment line.

```
while(x != 1):
```

This is a while line, so it must look like

```
while(test):
```

So we need to check that

```
x != 1
```

is a valid test. A test has to look like:

```
expression comparison expression
```

and since x is a valid expression (it is a variable name), != is a valid comparison (as we can see from our list of comparisons), and 1 is a valid expression (it is a number), and since this line is followed by some indented lines, it is a valid while flow-control line.

```
   y = x % 2
```

This line is indented as it is part of the while loop. It looks like an assignment, and indeed, y is a variable name, and x % 2, being a combination of a variable name (x) and a number (2) using one of the allowed operators (%), is an expression.

```
   if(y == 0):
```

Much like we checked the while line, this is a valid "if" flow-control line. In fact, since it has an "else" later, these form a valid if/else flow-control structure:

```
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
```

Note that the line following the "if" line is indented by two steps. The if line is indented once because it is inside the while loop. Then remember that lines under the if statement have to be indented three spaces more than the if line they are following. The effect is that the line

```
      x = x / 2
```

gets indented using six spaces since it is inside the "if", and the "if" is already indented by three spaces since it is inside the "while".

```
   print(x)
```

This is a print line, which looks in general like

```
print(expression)
```

and is indeed valid, since x, being a variable name, is a valid expression.

```
print("Done!")
```

This is another valid print line, since "Done!" is a valid string, and strings are valid expressions.

So we've checked that this program is syntactically valid. One final thing to point out about this syntax-checking procedure was its mechanical, recursive nature: Once we identify that, say, the line

```
x = 3 * x + 1
```

is an assignment, that means it has to be constructed from a variable name and an expression. So we identify the pieces of the line that should correspond to each of these: x is the variable name, and 3 * x + 1 should be the expression. Then we check the definitions of each of these: A variable name is any combination of letters, numbers, and underscores, with certain exceptions. x is thus a valid variable name. Now we check 3 * x + 1 is a valid expression similarly just by going back to the definition of "expression" in the syntax.

Aside:

The fact that our specification of the syntax leads naturally to a mechanical checking procedure is important. Since ultimately, there has to be a program that understands the programming language in order to turn it into ISA operations, it is natural that this program be able to take such a specification and follow an algorithm that generalizes our checking procedure. In your future computer science studies, you may take a course in designing your own programming language, where you will make all this precise and effectual.

2.4.2: Assignments

Summary: Assignments are *variable_name=expression*. The assignment line will evaluate the expression and store the result in the variable named on the left side of the "=".

Recall that assignments are written

```
variable name = expression
```

These are how we store and modify the values in variables. For instance, the line

```
x = 2
```

stores the value 2 in a variable called x. It is worth noting that here "=" is functioning as an action: "set x to have value 2", unlike its use in mathematics, where it serves to make statements of fact ("x is equal to 2").

If later in the program we want to use this stored value, we can refer to it by its name, x. For instance,

```
y = 2 * x
```

will store the value 4 into another slot called y. We've seen a couple of slightly trickier variants of this already. For example, we can update variables using themselves:

```
x = 2
x = 7 * x
x = 3 * x + x
```

In the first line, x is set to 2. Then in the second line, x is set to 7*x. To figure out what this stores in x, Python looks up what was already stored in x. It finds 2 was stored there, so it multiplies this by 7 as requested, and stores the new value of 14 into x. The last line then sets x to 56 (convince yourself, if you are unsure of why.)

Assignment statements allow us to store the results of a computation. So suppose we wish to multiply x by 2 using the multiplication operator *. We might naively write the code:

```
2 * x
```

This is a valid expression, but we note from our syntax tree that an expression is not one of the three valid kinds of lines of code! We do have valid lines like print lines which can print the value of an expression, or assignment lines, which can store the value of an expression. So if you want to multiply x by 2, you have to store the result somewhere, and so you would use an assignment line. Either you would create a new variable, maybe called y, and store the value there:

```
y = 2 * x
```

or you might overwrite the old value of x with this new value:

```
x = 2 * x
```

or you might print this quantity without storing it anywhere:

```
print(2 * x)
```

2.4.2.1: Variable names

Summary: Variable names satisfy the usual rules, but they also cannot be any names reserved by Python, like "if" or "while". They are also case-sensitive, and in any complicated code at all, they should be given sensible names that indicate their purpose.

Definitions: camel casing

We've already specified in the syntax discussion what valid variable names are, but there are a few further points to make.

Remember we said variable names cannot be words that Python treats as special? While you're not terribly likely to use them by accident, it is good to have a complete list handy of what these all are. So here you go:

```
and           if
as        import
assert       in
break       is
class       lambda
continue      not
def        or
del        pass
elif        print
else        raise
except        return
exec        try
finally        while
for        with
from        yield
global
```

The reason for this is relatively straightforward: If you named a variable "if", then when the Python compiler gets to a line like

```
if = 2
```

The first thing it sees is "if". So it might decide that this is in fact a flow-control line of the "if" sort. But then none of the rest of the line makes sense. In this example it could recover and unambiguously realize that this must be an assignment, but other examples using more complicated language features will not permit disambiguation, and so these names are simply forbidden.

Another possible gotcha is that variable names are case-sensitive:

```
banana = 2
Banana = 3
```

will create two completely different variables that can be used independently. Likewise,

```
x = 3
y = 3*X
```

will cause an error, since the second assignment tries to use the variable called X. We just created a variable called x, but that is unrelated to the completely different variable called X.

Finally, even though x, y, and z are valid variable names, and using one-letter names will save on typing, there is a balance between brevity and perspicuity to be struck: Imagine you're reading someone else's code and you come upon a line like

```
a = mass * radius * radius * roll_rate * roll_rate / 5 + mass * velocity * velocity / 2
```

You'd say--"Great, you're doing some physics. You've got some object with mass, radius, velocity, rolling, and you've stored these quantities in sensibly named variables called mass, radius, roll_date, and velocity, and you're doing some massive computation involving all of these, and you're storing it in some variable called a. That's all nice, but what the heck is this thing you're computing supposed to mean?" How much more obvious is this:

```
rolling_sphere_kinetic_energy = mass * radius * radius * roll_rate * roll_rate / 5 + mass * velocity * velocity / 2
```

Obviously that could end up being a lot of typing to use this variable repeatedly in the future, but it's worth thinking about. Even KE would be a better name here than just a.

Beyond just making the names meaningful, there are differing conventions found in the wild regarding naming. In this book, if a variable name is supposed to be a sequence of words (like above, when we wanted to name a variable as 'rolling sphere kinetic energy'), we will have all the words be lower-case and separated by underscores (since spaces are not allowed in variable names). There is another convention that some people use, which is to have nothing separate the words but to capitalize the first letter of each word, as in `RollingSphereKineticEnergy`. This is called **camel casing**.

2.4.2.2: Expressions

Summary: Expressions are the usual things, but they have to make sense: No adding a string to a number or dividing a number by 0. When the program gets to an expression, it will evaluate it by plugging in the current values of all variable names used in the expression and performing the computation.

Definitions: value (expression), evaluation (expression), concatenation

The right-hand side of an assignment is required to be an expression, i.e. any variable name, number, string, or valid combination thereof using arithmetic operations.

Any expression has a **value** which is the result of retrieving what is stored in each variable mentioned in the expression and applying all the specified arithmetic operations. For example, the line `x = 2 * 3` stores 6 into x because the value of the expression `2 * 3` is 6. The process of determining the value of an expression is called **evaluation**, and happens as follows:

- First, find any variable names in the expression and substitute in place of those variable names the values currently stored in the corresponding variables. If any of the variables has no stored value, throw up a logic error.
- Then perform the operations specified in the expression, with the usual order of operations:
  - Anything in parentheses gets evaluated first.
  - Any multiplication or division or modulus operations get performed left-to-right.
  - Then finally any addition and subtraction happen last.

  If any operation has invalid inputs (e.g., division by zero or attempting to multiply things that are not numbers) throw up a logic error.

We can use this to determine what gets stored in x in the following case:

```
x = 4
y = x + 1
x = 2
x = 5*x + 2*(3 - y)
```

The first assignment sets `x` to 4. The second one sets y to whatever is the value of `x + 1`. To evaluate this expression, we plug in the current value of `x` (namely 4) in place of the name `x` in this expression, yielding 4 + 1, and then we perform the arithmetic to get a value of 5. So the second assignment assigns the value 5 to `y`. The third assignment stores the value 2 in `x`.

Finally, to find out what the last assignment stores in x, we need to evaluate the last expression

```
5*x + 2*(3 - y)
```

To do this, we again plug in the current values of x and y in place of their names: x is currently 2, and y is currently 5, so we get:

```
5*2 + 2*(3 - 5)
```

Then we're back to a normal grade-school problem, whose answer is 10 - 4 = 6.

In addition to numbers, expressions are also allowed to use strings. For instance:

```
"Hello world!"
```

is a valid expression, and hence

```
s = "Hello world!"
```

is a valid assignment. The only caveat is that arithmetic operations and strings rarely make sense. For example, adding a number to a string isn't OK, so

```
"Hello" + 2
```

is not a valid expression and will create a runtime error. Likewise,

```
"hello" * "aerodrome"
```

is not a valid expression and will similarly elicit an error.

We can, however, add two strings:

```
"hello" + "world"
```

is a valid expression, and its value is the **concatenation** (i.e., sticking the two strings together to make one long string) of the two strings: "helloworld"

So consider the following examples:

```
x = Hello world!
y = z+1
z = 3+"hello"
```

The variable names on the left sides are fine, but all three expressions are invalid. The first one looks like it is meant to be a string, but because it is not enclosed in quotation marks, Python doesn't realize this and instead tries to interpret it as variable names. But variable names cannot contain spaces or !s, so this falls flat too, and Python gives up an error.

The second looks valid--it is a combination of a variable name and a number using the operator +. But we never defined a value for z! So Python will try to store the value of this expression in y. To do this, it will look up what we stored in z and try to add 1 to it. And this will fail, because we never stored anything at all in z.

The third example contravenes the weasel-word in the syntax definition: We required only "valid" combinations of variables, numbers, and strings. Obviously adding two numbers is a "valid" combination, but exactly what else counts? Certainly almost any arithmetic operation should work on any numbers or variables that store numbers. For instance,

```
x = 1+1
y = 78/x
```

As mentioned, division (namely, / and %) when the right operand is 0 will cause an error, since as in mathematics generally, dividing by zero does not make sense to a computer:

```
x = 4
y = x/2-2
z = x % y
```

Further, if you try to operate on a number and a string, this is not going to go well:

```
x = "hello"
y = 89/x
```

2.4.2.3: Strings

Summary: Strings are anything enclosed in quotation marks. To include a quotation mark inside a string, escape it--i.e., precede it by a backslash. To insert a line break into a string, escape the letter n. To insert an actual backslash into a string, just escape a single backslash with a second one.

Definitions: escape, newline

Recall we defined a string as any sequence of characters contained inside quotation-marks. It is a type of expression whose value is the text inside the quotes. But what if we change that text to itself contain a quote? For instance,

```
print("hi")
```

prints:

```
hi
```

But what if we want to print the following:

```
"hi", he said
```

We might try

```
print(""hi", he said")
```

But Python understands this as three things:

The string "", which contains no text, followed by the variable called hi (since this is outside any quotes, so is not a part of the string), followed by the string ", he said". Disaster follows.

To include a quote, then, you need to **escape** it--that is, you precede it in the string with a backslash. In general, backslash is the string escape character, meaning that whatever character comes after it doesn't have its usual meaning, but has some special meaning. The usual meaning of the double-quote character is "this is the boundary of a string", but if we precede it with a backslash, then its meaning just becomes just "the double-quote character". So

```
print("\"hi\", he said")
```

accomplishes the desired result. Similarly, if we want to store a string that contains multiple lines, like the string

```
XXO
 O_
OX_
```

which might represent a tic-tac-toe board in a game we're programming.

It turns out that if we escape the character n, when escaped, has the special meaning of `the **newline** character'--that is, a character which represents going to another line. Thus

```
print("XXO\n_O_\nOX_")
```

prints the above.

2.4.3: Print

Summary: Print lines evaluate the provided expression and output the value to the screen. We can put print statements at various places through our program to get an idea of what it's doing.

Now that we know what expressions are, print is reasonably straightforward: it simply prints to the screen the value of the expression it is given:

```
print("hello")
x = 4
y = "blah"
print(x*2/5)
print(y+"zzzzz")
```

prints

```
hello
1.6
blahzzzzz
```

We can use it to print the value stored in a variable.

```
print("Hello")
x = 2
print(x-3)
print(x)
```

A common use-case for print beyond just printing a final answer is debugging: If we have a program that is misbehaving and we want to get some idea of what is happening, we can insert a print statement in the middle. For instance, suppose we accidentally wrote our 3x+1 program thus:

```
x = 27
while(x != 1):
   y = x / 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print(x)
print("Done!")
```

Then we would see printed

```
27
82
247
...
```

and more numbers that would keep growing forever.

If we wanted to know what was happening in more detail, we could think: y is supposed to be either 0 or 1, telling us whether x was even or odd. So let us confirm that this is happening by printing y during the loop as well:

```
x = 27
while(x != 1):
   y = x / 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print(y)
   print(x)
print("Done!")
```

And we see printed

```
13.5
82
41.0
247
123.5
742
371.0
2227
1113.5
6682
3341.0
20047
10023.5
60142
...
```

So something has gone wonky, since we shouldn't be seeing any decimals in the procedure run normally. To make it even more clear, we'll add in some more print statements to show us which numbers being printed were x and which were y:

```
x = 27
while(x != 1):
   y = x / 2
   if(y == 0):
      x = x / 2
   else:
      x = 3 * x + 1
   print("here is y: ")
   print(y)
   print("behold x: ")
   print(x)
print("Done!")
```

Giving:

```
here is y:
13.5
behold x:
82
here is y:
41.0
behold x:
247
here is y:
123.5
behold x:
742
here is y:
371.0
behold x:
2227
here is y:
1113.5
behold x:
6682
```

So we see that instead of being 0 or 1, y is just being set to some crazy, often decimal values. So which line could possibly be wrong except the line that assigns a value to y?

So we see that

```
   y = x / 2
```

should have been

```
   y = x % 2
```

and then all is well.

2.4.4: Flow control

Summary: Flow-control lines determine what line gets executed next. A flow-control line can either be an if/else pair or a while loop.

Flow-control lines are lines that control the order in which the lines following it get executed. The two kinds of flow-control we introduce here are if/else lines and while lines:

2.4.4.1: if/else

Summary: "if" lines look like if(*test*): and are used to choose between two choices of sets of lines to run. The lines following an "if" line, which should be run only if the test in the "if" line is true, must all be indented--that is, they should start with three more spaces than the "if" line does. "if" lines are sometimes paired with an "else" line, which is written simply as else: The lines following an "else" line, which will be run only if the test is false, should also be indented with three more spaces than the "else" line.

The if statement exists for the following use-case: Imagine we're writing a search engine, and we have a variable that stores the word the user is searching for. It might have come from one of two files: poems.txt or raps.txt, and we need to determine which. (It's a very simple search engine for now). To display the answer, we'll have to have both the line

```
print("poems.txt")
```

and the line

```
print("raps.txt")
```

somewhere in our code. But we definitely don't want both of them to run--we only ever want at most one to run, but which one we want depends on which file the word was found in. Choosing what code to run depending on some condition is exactly the job of the if statement.

If you recall, an if/else statement is:

```
if(test):
   indented lines
else:
   indented lines
```

and it runs the first set of indented lines if the test succeeds, and the second set if the test fails.

So if we can contrive a test that will tell us whether the searched-for word is in poems.txt or raps.txt, we'll be set: We can then put the first print statement

```
print("poems.txt")
```

in the first set of indented lines, and the second print statement

```
print("raps.txt")
```

in the second set.

But these are supposed to go in as *indented* lines, so we return to our syntax tree to recall what this means: They have to be preceded with three spaces more than the previous line. There were no spaces at the beginning of the if line, so to indent the print lines, we given them three spaces at the beginning, like so:

```
if(test):
   print("poems.txt")
```

```
else:
   print("raps.txt")
```

(We still don't know what the test is supposed to be, but we'll discuss that later.)

For if statements in general, the semantic meaning of the indentation makes it very important, and often a source of subtle bugs:

```
x = 2
if(x == 2):
   print("x is two")
else:
   print("x is not two")
print("potato")
```

outputs

```
x is two
potato
```

whereas

```
x = 2
if(x == 2):
   print("x is two")
else:
   print("x is not two")
   print("potato")
```

outputs only

```
x is two
```

because the line

```
print("potato")
```

is indented, which puts it in the block of lines that only get executed if the test fails. And since the test is testing whether x is equal to 2, the test does not fail.

2.4.4.2: tests

Summary: Now we need to know what kinds of tests we can use inside an "if" line. They are simply two expressions separated by a comparison operator, which can test whether the two expressions are equal, or whether, say, one is larger than the other.

Definitions: succeed (tests), fail (tests)

So what kinds of tests can we use in if lines? The syntax tree tells us that they have to look like

```
expression comparison expression
```

We already know what expressions are. To find out what a "comparison" is, we look at our syntax tree and find that comparisons are any of:

```
<, >, <=, >=, ==, !=
```

which have the following meanings:

Comparison operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

On either side of the comparison operator can be an expression. The test is said to **succeed** or to **evaluate to true** if the values of the two expressions do in fact compare in the way that the

comparison operator dictates. If they do not, then the test is said to **fail** or to **evaluate to false**.

So for example, we can test if 2*x is greater than 5 with the test:

```
2*x > 5
```

We can test if the variable x stores the string "hello" with

```
x == "hello"
```

If we have three variables x, y, and z, which all store numbers, and we want to print something if x + y is not the same as z, we can do:

```
if(x+y != z):
   print("something")
```

One final dumb example is

```
if(1 == 1): print("hello") else: print("something
is wrong with the universe")
```

This is indeed a valid test: Numbers are valid expressions, and == is a valid comparison, so 1 == 1 is a valid test. But it should always succeed (unless something goes wrong with the universe), so the above program should be just a very complicated way of writing the entirely equivalent code:

```
print("hello")
```

2.4.4.3: while

Summary: "while" lines look like while(*test*): and are used to get a set of lines to run repeatedly. The lines following the "while" line that you want to run over and over until the test in the "while" line is no longer true should all be indented three spaces further than the "while" line.

Definitions: infinite loop

Finally, we get to the last flow-control statement that we'll talk about here, namely while. While is how you get a block of lines repeated several times:

```
while(test):
   lines that you want to run for as long as the test holds true
   (the test is re-run every time we get to the end of the indented lines)
```

This comes in handy when, as an example, we want to print 20 smiley faces followed by a smirk. We could use 21 print statements:

```
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":)")
print(":/")
```

But if we wanted to print 100 smileys, our program would get rather long, and it may be harder to ensure we got exactly 100 print statements in there. To simplify this dramatically, we can instead use a while statement. This requires some thought, however: The while statement will repeat the indented block as long as the test is true. So clearly we want the print line to be in the indented block:

```
while(something):
   print(":)")
print(":/")
```

What should the something be? We want the indented block to repeat 20 times, so the something should be a test that is true the first 20 times it is run, but not the 21st time. One way to accomplish this is to have a counter variable that counts how many times the indented block has been run. When the program starts, we'll set the counter to 0:

```
counter = 0
while(something):
   print(":)")
print(":/")
```

Except now, each time the block is run, we want the counter to increase by 1. An assignment that can increase the counter by 1 is:

```
counter = counter + 1
```

so if we put this inside the indented block, it will be run each time the indented block is run:

```
counter = 0
while(something):
   print(":)")
   counter = counter + 1
print(":/")
```

Now finally, what should the something be? The first time the while block is run, counter will start as 0 and get updated to 1. The second time, it will start as 1 and be updated to 2. The third time, the counter will start as 2 and be updated to 3 during the execution of the block. So if the something is

```
counter < 3
```

then after the third run of the block, counter will be 3, so the test will fail, and the block will not be repeated a fourth time, and the line will proceed to the line

```
print(":/")
```

Thus if we wanted to print three smileys and then a smirk, we would do:

```
counter = 0
while(counter < 3):
   print(":)")
   counter = counter + 1
print(":/")
```

We can double-check our understanding of what happens by single-stepping this program:

Current line	Why we went to this line	Variables after current line runs	Output so far
counter = 0	It was the next line	counter = 0	
while(counter < 3):	It was the next line	counter = 0	
print(":)")	counter was < 3	counter = 0	:)
counter = counter + 1	It was the next line	counter = 1	:)
print(":)")	We reached the end of the while block, but counter was < 3, so we go back to the start	counter = 1	:) :)
counter = counter + 1	It was the next line	counter = 2	:) :)
print(":)")	We reached the end of the while block, but counter was < 3, so we go back to the start	counter = 2	:) :) :)
counter = counter + 1	It was the next line	counter = 3	:) :) :)

print(":/")	counter was not < 3	counter = 3	:) :) :) :/

Thus to accomplish the original goal, we can use the program:

```
counter = 0
while(counter < 20):
 print(":)")
 counter = counter + 1
print(":/")
```

This pattern of having a counter incrementing by 1 in a while loop is very common, and you should bear it closely in mind for the future, but it is not the only paradigm for while loops. For example, we can use a while loop to print all the powers of 2 less than a thousand:

```
x = 1
while(x < 1000):
   print(x)
   x = 2*x
print("Done")
```

If you're not comfortable with why this program accomplishes this goal, single-step it in the same manner as before to see.

Finally, in the spirit of the last dumb if/else example, we also have a dumb while loop example. Remember, 1 == 1 is still a valid test, so

```
while(1 == 1):
   print(":)")
```

is a valid while loop. Since the test is always true, and the while statement is supposed to repeat the indented lines for as long as the condition is true, it will print smileys forever. This is called an **infinite loop**. They can happen intentionally when you're just trying to be infinitely happy like in the above example, but they are generally a run-time error (i.e., part of a valid program that provides undesirable behavior when run), and are not always as obvious as that first example might lead you to believe. For instance:

```
x = 49
while(x*x % 4 != 3):
   x = x + 1
   print(x)
   print(x*x % 4)
```

Does this program infinite-loop or not?

2.4.5: Comments and whitespace

Summary: Python ignores any lines that are blank or otherwise only contain spaces and tabs. It also ignores anything after a # symbol, so you can use this to insert English comments into your code explaining what it does.

Definitions: whitespace, comment

As you may have gathered from previous examples that inadvertently included blank lines, python ignores any line that is blank, or any line that includes nothing but spaces or tabs. Such lines are called **whitespace** (presumably because the background colors for many text editors is white, and so the blank space is white).

Python will also ignore anything after a '#' character on any line. Text preceded by a '#' can therefore be used to annotate the program, explaining what it does inline. These pieces of explanatory text are called **comments**.

Both commenting and the judicious use of whitespace can help someone reading the code to better understand what's going on (or at least, what the programmer meant to happen). For an example, we comment our original 3x+1 program and give it a little whitespace to help separate the different steps

```
x = 27
# Repeat the procedure as long as x isn't 1
while(x != 1):

   # To test whether x is even or odd, we store x % 2 in the variable y
   y = x % 2

   if(y == 0):
      # If y is 0, then x was even; divide it by two!
      x = x / 2

   else:
      # If y wasn't 0, then x was odd; 3x+1 it!
```

```
        x = 3 * x + 1

    print(x)

# If we got here, then the while loop is over and we can declare victory!
print("Done")
```

2.5: Simple examples

Now we set about writing some simple programs. As discussed in chapter 1 and reflected in the structure of this chapter, this happens in two stages: First, turning the problem into a computer-friendly but still human-readable algorithm, and then second, writing that algorithm in a computer-friendly language, (in our case, Python).

That is the theory, at least. In practice, the first algorithm we think of won't always do the trick quite correctly, or even if it does, the first program we write may not implement that algorithm correctly. This is normal, and easily remains true of all computer programming regardless of level of experience. So in these examples, many of our first attempts may include some error, or may be inefficient or scale poorly. In these case, we will include further steps for debugging or optimizing, where we go back, identify the source of the problem, and try to fix it.

2.5.1: Factorial

Summary:

To write a program that computes factorials (10 factorial is 10*9*8*7*6*5*4*3*2*1, for example) we use the idea of a while loop with a counter, but we keep another variable that will eventually store the answer that we multiply by the counter each time we go through the loop.

There is some subtlety in getting exactly the right answer and not going through the loop one too many or one to few times.

Definitions: off-by-one error

Introduction:

The factorial is a mathematical operator that takes in a single positive integer (1, 2, 3, ...) and outputs the product of all the positive integers at or below the input. The factorial of 5 is written as "5!". So 5! = 5*4*3*2*1 = 120, and 10! = 10*9*8*7*6*5*4*3*2*1 = 3628800. In this example, we wish to write a program that will compute the actual value of 10!.

Algorithm:

The definition above is completely unambiguous, and explains perfectly well to a human how to compute factorials. On the other hand, we saw earlier a complete list of all the things we can tell a computer how to do, namely assignment, print, if, if/else, and while. You'll note that "multiply the integers 1 through 10" is not on that list, so we need to find a way of telling the computer how to compute 10! using only these simpler operations.

From one perspective, this is easy: Clearly we want to print the product 1*2*3*...*10. If we reference our chart above, we see that we are allowed to print any expression. But what was an expression? It was any number, string, variable, or combination thereof using operations +, -, *, /, %. Aha! So, for example, 1*2*3*4*5 is an expression (as it is a combination of numbers using the operation *). Therefore:

Program:

```
print(1*2*3*4*5*6*7*8*9*10)
```

Debrief:

That was all very good, except now say I want to know what is 27! Now I have to make non-trivial and tedious changes to the program--it is basically the same work as doing this on a calculator by hand! In short, we have a problem of scalability. This program cannot compute larger factorials without us doing a large amount of work also to change it. What we would like is a program that looks like:

```
input = 10
... some things
... some more things
print(answer)
```

That is, the program starts by storing the desired input value in a variable, and then whatever that value is, it does some computation and eventually ends up with the factorial of this initial value stored in the variable called answer.

This way, if we decide later that we instead wanted 27!, we can just change the first line to

```
input = 27
```

and leave the rest of the program untouched, and still get the correct answer.

Algorithm, Mark II:

The above desired form of the program tells us the first and last steps of the algorithm:

```
Step 1: Store the number whose factorial we want in a variable called input.
...
Step last: print the number stored in the variable answer.
```

So clearly at some point we'll need to make a variable called answer. May as well make that step 2. So step 2 will be an assignment (a valid operation), setting answer equal to...what? Clearly we cannot just say `answer = input!` since ! is not one of the operations that Python understands. So let's just start small--`answer = 1`.

```
Step 1: Store the number whose factorial we want in a variable called input.
Step 2: Store the number 1 in the variable answer.
...
Step last: print the number stored in the variable answer.
```

Now, clearly what we need to do is to first multiply answer by 2, and then by 3, and then by 4, and so on until we get to 10. So let us store the thing we're multiplying by in yet another variable. This variable will be counting up from 2, and will be multiplied into answer at each step. So we'll call this variable counter, and since the first multiplication will be by 2, we'll start this variable at 2.

```
Step 1: Store the number whose factorial we want in a variable called input.
Step 2: Store the number 1 in the variable answer.
Step 3: Store the number 2 in the variable counter.
...
Step last: print the number stored in the variable answer.
```

Now we want to increment counter repeatedly until it is 10, or, more precisely, until it matches whatever is in the variable input. Intuitively this process looks like:

```
answer = answer*counter
counter = counter + 1
answer = answer * counter
counter = counter + 1
answer = answer * counter
counter = counter + 1
answer = answer * counter
counter = counter + 1
...
```

except we want this to stop when counter exceeds input.

So:

```
Step 1: Store the number whose factorial we want in a variable called input.
Step 2: Store the number 1 in the variable answer.
Step 3: Store the number 2 in the variable counter.
Step 4: Repeat the following until counter is greater than input:
 Substep 1: Replace answer with answer * counter
 Substep 2: Increment counter by 1
Step 5: print the number stored in the variable answer.
```

And recall that this 'repeat the following' sort of step is something that we can tell the computer to do--this is what the while instruction does! So we do:

Program, Mark II:

```
input = 10
answer = 1
counter = 2
while(counter < input):
   answer = answer * counter
   counter = counter + 1
print(answer)
```

Let us single-step this program with a slightly smaller input: 4. So the program should compute 4! = 4*3*2*1 = 24 and print this result. Let us see:

Current line	Why we went to this line	Variables after current line runs	Output so far
input = 4	It was the next line	input = 4	

answer = 1	It was the next line	input = 4 answer = 1	
counter = 2	It was the next line	input = 4 answer = 1 counter = 2	
while(counter < input):	It was the next line	input = 4 answer = 1 counter = 2	
answer = answer * counter	counter was less than input	input = 4 answer = 2 counter = 2	
counter = counter + 1	It was the next line	input = 4 answer = 2 counter = 3	
answer = answer * counter	We reached the end of the while block, but counter was still less than input, so we go back to the start	input = 4 answer = 6 counter = 3	
counter = counter + 1	It was the next line	input = 4 answer = 6 counter = 4	
print(answer)	We reached the end of the while block, and counter was not less than input, so we proceed to the next line	input = 4 answer = 6 counter = 4	6

So something went slightly wrong! That is, a runtime error has occurred!

We wanted the last step to be

```
answer = answer * 4
```

. But counter was set to 4 at the end of the loop body, and then because counter < input was false (4 is not less than 4), the next iteration of the loop, which would have done the multiplication by 4, did not run.

This is a particularly notorious kind of runtime error called an **off-by-one error**. There are a couple of possible fixes. The simplest is that we want the loop to run when counter is less than input or equal to it. To get this behavior, we replace < by <= to get:

```
input = 10
answer = 1
counter = 2
while(counter <= input):
answer = answer * counter
counter = counter + 1
print(answer)
```

Another possible fix, the analysis of which is left as an exercise, is the following:

```
input = 10
answer = 1
counter = 1
while(counter < input):
   counter = counter + 1
   answer = answer * counter
print(answer)
```

Debrief:

So finally we have a factorial program that we can easily use to compute larger factorials too:

```
input = 50
answer = 1
counter = 2
while(counter <= input):
   answer = answer * counter
   counter = counter + 1
print(answer)
```

Later we will learn how to make the program ask for and receive input from the person running the program (as opposed to now, when you have to actually change the program to affect the computation it runs). Having written the program not using any foreknowledge of the value of the input, it will be easy plug in the code for reading like:

```
[get user's input and store its value in the variable input]
answer = 1
counter = 2
while(counter <= input):
   answer = answer * counter
```

```
   counter = counter + 1
print(answer)
```

2.5.2: Primality testing

Summary: To test if a number is prime, we simply need to take all the numbers between that 1 and number (exclusive), and check if any of those numbers divides into our number evenly. To do this checking, we can use a while loop with a counter, and to check whether a number divides evenly into another, we can use the modulus operator. This algorithm is badly inefficient, and finding a more efficient one is a hard problem.

Introduction:

A prime number is a positive integer that is not evenly divisible by any positive integer other than itself and 1. For instance, 12 is evenly divisible by 4 and therefore is not prime. 13, on the other hand is prime. Prime numbers are of interest because every number can be built out of primes. For example, we just saw that 12 is evenly divisible by 4. More specifically, 12 = 4*3. 4, in turn, is also not prime--it is evenly divisible by 2. More specifically, 4 = 2*2. So when we wrote 12 = 4*3, we could break it up further as 12 = 2*2*3. 2 and 3 are primes, so we have shown how to make 12 using just prime numbers.

It is sometimes of great interest to determine whether a number is prime. In this section, we shall write a program capable of doing so. Specifically, the program shall start with a line like

```
n = 101231
```

and shall, at the end, print either "n is prime" or "n is not prime" depending on the result of its test.

Algorithm:

We are given a number, say stored in a variable called n. We want to test if n is prime. As before, we start with the definition: If it is prime, then no number between 2 and n-1 inclusive divides into n evenly. Thus we can step through these numbers in sequence, as we did with the counter variable in the previous example, and test each one in turn to see if it divides into n evenly.

```
Step 1: n = 10000001
Step 2: counter = 2
Step 3: Repeat the following as long as counter < n:
 Substep 1: Test if counter divides into n evenly
 Substep 2: Increment counter
```

This seems to cover all the things required by the definition of "prime", but it is still quite incomplete: What do we do if counter does divide evenly into n? And for that matter, what do we do at the end of the loop?

In the course of step 3, we'll discover whether or not n was prime. If we store this information somehow, then at the end of the loop we can print out the result based on this stored information. To this end, we'll also create a variable called is_prime. We'll start with it set to 1, and if we find a divisor, we'll set it to 0. Then, at the end, we'll test the value of is_prime and print as appropriate.

```
Step 1: n = 10000001
Step 2: counter = 2
Step 3: is_prime = 1
Step 4: Repeat the following as long as counter < n:
 Substep 1: Test if counter divides into n evenly.  If it does, set is_prime to 0
 Substep 2: Increment counter
Step 5: If is_prime is 1, print "n is prime"
Step 6: If is_prime is 0, print "n is not prime"
```

Program:

To turn this into an honest program is reasonably straightforward. The elements we have not yet used are the tests, which are accomplished with the if/else instructions. How to see if `counter` divides `n` evenly? One way to do it is to divide n by counter and see if the quotient is an integer. However we don't yet have any operation for testing "is this variable an integer", so that's no good for now. Another way would be to divide n by counter and see if the remainder is 0. Recall we can get the remainder of this division with n % counter. So we get:

```
n = 1000001
counter = 2
is_prime = 1
while(counter < n):
   if(n % counter == 0):
      is_prime = 0
   counter = counter + 1
if(is_prime == 1):
   print("n is prime")
if(is_prime == 0):
   print("n is not prime")
```

This may take quite a few steps. Remember that you can adjust how many steps are taken by the "step" button using the input box next to that button.

Debrief:

It is worth noting that the values we chose for the variable is_prime--0 to represent that the number is not prime, and 1 to represent that it is--were completely arbitrary. We could have started instead by setting is_prime to the string "durian", and then in the loop if we found a divisor of n, we could change it to the string "penguin goes shopping" and get the exact same behavior:

```
n = 1000001
counter = 2
is_prime = "durian"
while(counter < n):
   if(n % counter == 0):
      is_prime = "penguin goes shopping"
   counter = counter + 1
if(is_prime == "durian"):
   print("n is prime")
if(is_prime == "penguin goes shopping"):
   print("n is not prime")
```

The use of 1 and 0 is sort of standard, however, because in some sense (which we will discuss later), it takes the least amount of space possible to store a variable that is only ever 1 or 0, whereas if we have to store the strings "durian" and "penguin does shopping", this requires more space. In many contexts, therefore, when we have a variable whose value represents the truth of some statement, a value of 1 conventionally represents "true" and a value of 0 represents "false".

On the topic of potential improvements, one thing that might be nice is for this program to not just tell us that a number isn't prime, but also to tell us why the number isn't prime--that is, tell us a number that divides evenly into it.

This program is also far, far less efficient than it could be in many ways. You can probably think of improvements already--for instance, we don't have to test all the possible divisors less than `n`--we can certainly get away with testing only to n/2, since if n/2 evenly divides n, then 2 also evenly divides n, so we would have caught that already. Also, none of the numbers between n/2 and n can possibly divide evenly into n. In fact we can get away with fewer than that.

For a further inefficiency, you should see how long it takes to run in tell us that 222 isn't prime, and compare this with how many steps you think ought to be needed.

The general question of whether this problem had a truly efficient solution remained frustratingly unanswered for years until an undergraduate project by two students together with their adviser at the Indian Institute of Technology found an algorithm that they could prove would get the answer quickly (in some precise complexity-theory sense) in all cases.

2.5.3: Counting digits

Summary: To count the digits in a number, we repeatedly divide it by 10 until it is smaller than 1. If we keep a counter as we do this, then the number of times we divide by 10 will be the number of digits in the number (provided we do this carefully and avoid off-by-one errors).

Introduction:

Starting with some number, count how many digits are required to write the number down. For example, with the number 1045, the answer would be 4.

Algorithm:

This is one of those problems that is very easy to do by hand: When you have a number, it is given to you as a list of digits, so you just count the digits on the paper--no sweat. But to a computer, 'look at how many digits are in this number' isn't one of the operations we are allowed to do to numbers. All we can do is add, subtract, multiply, and divide, and so to solve the problem with a computer, we need to think of how to solve it with only these operations available to us.

So, to get the number of digits in a number mathematically, we can repeatedly divide it by 10 until it gets below 1. The number of times we have to do this is the number of digits in the number. For instance, 9 / 10 = 0.9, so we only have to divide once, so the number 9 has one digit. 128 / 10 = 12.8, 12.8 / 10 = 1.28, 1.28 / 10 = 0.128. So it takes three divisions to get to below 1, and indeed, 128 has three digits.

To accomplish this programmatically, we will need two variables--one to store the number that we will repeatedly divide--say this is called `n`, and another to store how many times we've divided so far--say this one is called `digit_count`. Then we can write the algorithm more precisely as:

```
Step 1: n = 1238129 (or whatever)
Step 2: digit_count = 0
Step 3: Repeat the following as long as n > 1:
 Substep 1: Increment digit_count
 Substep 2: Divide n by 10
Step 4: Print digit_count
```

Program:

```
n = 1238129
digit_count = 0
while(n > 1):
   digit_count = digit_count + 1
   n = n/10
print(digit_count)
```

You would be right to worry about an off-by-one error--do we start digit_count at 0 or at 1? We can tell by single-stepping the program in a simple case, as we did before, or you can run the program with a few different inputs as examples. Convince yourself that we are OK.

There is also a tricky edge-case that may have slipped your notice, however: Consider what happens when we start with `n = 1`: First `digit_count` will be set to 0, and then we'll run the while loop as long as `n > 1`. But even at the start, this test fails, as `n = 1`, so `n` is not greater than 1! So the stuff inside the while loop gets skipped entirely, and the value of `digit_count`--namely 0--gets printed. And this is definitely the wrong answer--the number 1 has one digit!

In fact, this isn't the only issue--this problem shows up whenever `n` gets set to 1 during execution, for example if it started at 1000, we would get the wrong answer of 3.

To fix this, we simply need to realize that we wrote our algorithm correctly--we stop if n is below 1--but our program doesn't reflect this--it stops if n is below 1 *or if n is equal to 1*. We can rectify this by telling it to continue with the while loop even when n is 1:

```
n = 1238129
digit_count = 0
while(n >= 1):
   digit_count = digit_count + 1
   n = n/10
print(digit_count)
```

Debrief:

We should note that this program doesn't work for non-integers, nor for integers below 1.

If you played around with the factorial program, you probably got the sense that factorials were pretty huge. To get an idea of how huge, we can combine our digit-counting program with our factorial program to find out how many digits are in 99!:

```
input = 99
answer = 1
counter = 1
while(counter < input):
   counter = counter + 1
   answer = answer * counter
n = answer
digit_count = 0
while(n >= 1):
   digit_count = digit_count + 1
   n = n/10
print(digit_count)
```

2.6: Applications

In this section, we'll revisit our three example applications of search engine, game console, and cell phone to see how we might start to apply our limited knowledge of Python in the direction of creating these systems. The programs we'll write here will only be indicative of the ideas involved, since we don't yet know enough of the Python language to provide a realistic solution for any of them--for instance, we don't even have a way of getting user input without requiring the user to edit the program! But we'll bash on ahead anyway and see what we can do.

2.6.1: Search engine

As we discussed in chapter 1, a working search engine solves many different problems--storing lots of files on lots of computers, displaying a search box to millions of people per minute, and more. Here, we shall suppose we have the files split into words and each word is stored in its own variable, like so:

```
file1_name = "Ode on the death of a favorite cat"
file1_word1 = "twas"
file1_word2 = "on"
file1_word3 = "a"
file1_word4 = "lofty"
file1_word5 = "vases"
file1_word6 = "side"
file2_name = "Elegy written in a country churchyard"
file2_word1 = "the"
file2_word2 = "curfew"
```

```
file2_word3 = "tolls"
file2_word4 = "the"
file2_word5 = "knell"
file2_word6 = "of"
file2_word7 = "parting"
file2_word8 = "day"
```

We'll also suppose the word that the user searched for is already stored for us in a variable. Then the game is just to take each word from each file and compare it to the search word.

```
search_word = "the"
```

Then, to make do the actual searching is just a painful slog of if statements like:

```
if(search_word == file1_word1):
   print(file1_name)
```

So the full program looks like:

```
search_word = "the"
file1_name = "Ode on the death of a favorite cat"
file1_word1 = "twas"
file1_word2 = "on"
file1_word3 = "a"
file1_word4 = "lofty"
file1_word5 = "vases"
file1_word6 = "side"
file2_name = "Elegy written in a country churchyard"
file2_word1 = "the"
file2_word2 = "curfew"
file2_word3 = "tolls"
file2_word4 = "the"
file2_word5 = "knell"
file2_word6 = "of"
file2_word7 = "parting"
file2_word8 = "day"
if(search_word == file1_word1):
   print(file1_name)
if(search_word == file1_word2):
   print(file1_name)
if(search_word == file1_word3):
   print(file1_name)
if(search_word == file1_word4):
   print(file1_name)
if(search_word == file1_word5):
   print(file1_name)
if(search_word == file1_word6):
   print(file1_name)
if(search_word == file2_word1):
   print(file2_name)
if(search_word == file2_word2):
   print(file2_name)
if(search_word == file2_word3):
   print(file2_name)
if(search_word == file2_word4):
   print(file2_name)
if(search_word == file2_word5):
   print(file2_name)
if(search_word == file2_word6):
   print(file2_name)
if(search_word == file2_word7):
   print(file2_name)
if(search_word == file2_word8):
   print(file2_name)
```

2.6.2: Game console

Comparing scores:

We started this chapter with an example algorithm for comparing the scores of three players. We'll now implement this algorithm as a Python program, supposing we have the three scores stored in three variables, say like so:

```
player1_score = 20
player2_score = 21
player3_score = -1
```

Recall the algorithm:

```
If score1 is greater than score2 and score1 is also
greater than score3: Display the text: "PLAYER 1 WINS!"
```

```
If not, then if score2 is greater than score1 and score2 is also
greater than score3: Display the text: "PLAYER 2 WINS!"

If not, then if score3 is greater than score1 and score3 is also
greater than score2: Display the text: "PLAYER 3 WINS!"
```

So the main difficulty is in performing this combined test, where we test two things at once. We can perform one of the tests like:

```
if(player1_score > player2_score):
   ...
```

But then we need to put the second test of step 1, namely, player1_score > player3_score, somewhere. The point is that we can break up the algorithm to perform only one test at a time: Instead of

```
If score1 is greater than score2 and score1 is also
greater than score3: Display the text: "PLAYER 1 WINS!"
```

We can say

```
If score1 is greater than score2, then test if score1 is greater
than score3.  If this second test is true, print the text: "PLAYER 1 WINS!"
```

Thus we get the code:

```
if(player1_score > player2_score):
   if(player1_score > player3_score):
      print("PLAYER 1 WINS")
```

So for a full version of this program, we get:

```
player1_score = 20
player2_score = 21
player3_score = -1
if(player1_score > player2_score):
   if(player1_score > player3_score):
      print("PLAYER 1 WINS")
if(player2_score > player1_score):
   if(player2_score > player3_score):
      print("PLAYER 2 WINS")
if(player3_score > player1_score):
   if(player3_score > player2_score):
      print("PLAYER 3 WINS")
```

Computing distances:

As discussed at in chapter 1, one of the things that happens frequently in a game is that we need to compute distances between points. Let us say we are dealing with 2D points, whose coordinates are stored in variables like:

```
point1_x = 20
point1_y = 55
point2_x = 10
point2_y = 60
```

representing point 1 being (20,55) and point2 being (10,60).

And say we want to see which point is furthest away from the origin--(0,0). Then the algorithm is straightforward:

```
Compute the distance of point 1 to the origin and store it in a variable
Compute the distance of point 2 to the origin and store it in another variable
If the first variable is larger than the second, print that point 1 is closer to the origin
If the second variable is larger than the first, print that point 2 is closer to the origin
If the two variables are equal, print that they are the same distance to the origin
```

The one problem that presents is that the distance to the origin is given by

```
√(x1*x1 + y1*y1)
```

but we don't yet have an operation for square root. Fortunately, instead of comparing √(x1*x1 +

y1*y1) with √(x2*x2 + y2*y2), we can just compare, x1*x1 + y1*y1 with x2*x2 + y2*y2, since sqrt is an increasing function. So we'll modify our algorithm to not compute the distances, but rather the square distances, and get the program:

```
point1_x = 20
point1_y = 55
point2_x = 10
point2_y = 60
square_distance1 = point1_x*point1_x + point1_y*point1_y
square_distance2 = point2_x*point2_x + point2_y*point2_y
if(square_distance1 < square_distance2):
   print("Point 1 is closer to origin")
if(square_distance1 > square_distance2):
   print("Point 2 is closer to origin")
if(square_distance1 == square_distance2):
   print("Points are the same distance from the origin")
```

2.6.3: Mobile phone

Recall our earlier discussion of a cell phone with an accelerometer--a noisy sensor. We want to estimate our acceleration, but the readings may be affected--sometimes rather dramatically, by random noise. Since the noise is equally likely to cause the sensor to overestimate as to underestimate, we can take some number of readings from the sensor, and instead of taking any individual reading as the actual acceleration value, we can take the average of all of them.

In this example, we suppose we've somehow got 5 readings from the sensor stored in some variables for us. (For the moment, we'll just store the values rather than actually read them from a sensor, since we're studying how to do the averaging, and not yet how to talk to sensors):

```
reading_1 = 20
reading_2 = 21
reading_3 = 15
reading_4 = 22
reading_5 = 20
summation = reading_1 + reading_2 + reading_3 + reading_4 + reading_5
average = summation/5
print("The average is: ")
print(average)
```

2.6.4: Debrief

So after all of that, where are we on our journey? At this point, recall, we were working our way down the tower of abstractions. We're still in the top half of the tower--those layers that deal with programming a computer (as opposed to building a computer), and in particular we're currently at the programming language level, just above the ISA. That is, we are not programming the computer directly, but we are writing code that we trust can be turned into ISA code and actually be run by a computer.

This chapter hopefully introduced us to the concepts and mindset involved with programming at this level, as well as some experience with actual programming. However, this experience may have at times felt somewhat limited or unwieldy. This was indeed the case--we intentionally restricted attention to a small subset of the Python language focus on the concepts and thought process with the minimal useful complexity that we could get away with.

As a result, none of the preceding "real world" examples contain realistic production-quality code. They do, however, contain the nuggets of basic ideas that are used in those worlds in a more full, unrestricted programming-language environment. In the next chapter, we'll learn some more advanced constructs in Python that will allow far less cumbersome implementations.

For example, some unsatisfying features that will be rectified in the next chapter include:

- We didn't allow user input
- Our method of output was only console printing
- The only way of storing a massive collection of values was to use a massive number of variables

Stay tuned!

2.7: Exercises

2.1: What does programming express to the computer?

1. A solution to a real world problem in terms that the computer can understand.
2. A high-level, more human-friendly way to tell the computer what operations to do.
3. A sequence of steps for it to perform (also known as the program), specified in a way that the computer understands (also known as a programming language).
4. Both b and c

5. All of the above
6. None of the above

Answer:
(5), programming: solves real world problems, is high-level and human-friendly, and expresses algorithms to the computer.

2.2: What acts as an intermediary between a high-level programming language and a low-level language?**Answer:**
A compiler

2.3: In the classic parable mentioned in the chapter, what exactly caused the trouble?

What does a computer programming language and algorithm need to ensure to avoid such a situation?

Answer:
Ambiguity in the instructions caused the trouble. To prevent this issue, a precise and unambiguous algorithm is required.

2.4: According to the text, what specific method to solve problems do computers need that humans may not necessarily need?

2.5: What are edge cases (also known as "corner cases")?**Answer:**
Edge-cases are situations within a program that are in some way exceptional, such as a variable having a value on the boundary between desirable values and undesirable values, or a resource that the program wants to access being unavailable.

2.6: Define efficiency and scalability. How are they related?

Under what circumstances will you not bother making your program more efficient?

Answer:

Efficiency is how many (or how few) resources are used to execute a program. These resources can include time, memory, CPU, money, etc.. Scalability is how well a program would operate if it were required to handle big data (how efficiently it operates when scaled up). For example, a program that can handle 3 players playing a game may not be very scalable, and could not handle 50 people playing that same game at once.

An inefficient program that is scaled up will use significantly more resources than an efficient program. Thus, the impact of efficiency becomes more obvious when we scale up the program

A programmer may not bother making their program more efficient if they know it will never be used on a large scale. For example, a video game developer may create a game that runs well for 5 people playing at once, but would work inefficiently with 20 people playing at once. The developer doesn't need to improve his/her program's efficiency if it is not meant to be scaled beyond 5 players.

2.7: What three line types are there in Python?**Answer:**
Assignment lines, print lines, and flow-control lines.

2.8: How does indentation play a role in flow-control lines?**Answer:**
The indented lines following a flow-control line are executed if the test on the flow-control line evaluates to true, and are skipped otherwise.

2.9:

Label each line of the example python program from section 2.3.2 as an 'assignment line', 'print line', or a 'flow-control' line.

```
x = 27
while(x != 1):
   y = x % 2
   if(y == 0):
      x = x / 2
   else:
      x = 3*x+1
   print(x)
print("Done!")
```

Answer:

```
x = 27           assignment
while(x != 1):   flow-control
   y = x % 2     assignment
   if(y == 0):   flow-control
      x = x / 2  assignment
   else:         flow-control
      x = 3*x+1  assignment
   print(x)      print
print("Done!")   print
```

2.10:

If a computer executed the instruction "a=2" on line number 21, what would be the line number of the next instruction executed?

What type of instruction would have to be on line 21 to move to an instruction that is not on the following line?

Answer:
Line 22 would be the next line unless line 21 contained a flow-control line with test that evaluated to false

2.11: Using the definition of an assignment, assignments assign a(n) ___ to a(n) ___?

1. variable name, expression
2. expression, test
3. expression, comparison
4. variable name, comparison

Answer:
(1)

2.12: What is camel-casing?

2.13: What is the arithmetic operation permitted on Strings and what is it specifically called?

Give an example of this principle.

2.14: How does the backslash character (\) play a role in Python syntax?

2.15: How do you comment in Python?

2.16: Which of the following are valid Python variable names?:

1. `_num_items`
2. `item_#6`
3. `numItems`
4. `num_items6`
5. `2015sales`
6. `_2015_sales`
7. `item 6`
8. `sixth-item`
9. `global`

2.17: In this example program from section 2.4.2.2,

```
x = 4
y = x + 1
x = 2
x = 5*x + 2*(3 - y)
```

we know that 6 is the value stored in x after the code has finished executing. What is the final value stored in y? Explain.

2.18: Copy the following program into the python simulator and run it. Why does this program yield an incorrect result?

What should be changed in order for this program to work as desired?

```
# you have $30 and the taxi will cost $20. Do you have enough money to cover the ride?
cashAmount = 30
if (CashAmount > 20):
   print("you have enough cash for the taxi")
else
   print("you do NOT have enough cash for the taxi")
```

2.19: Consider the following Python program:

```
name = "John Smith"
age = "twenty"
```

```
print("Name:")
print(name)
print("Age:")
print(age)
```

the output for this program looks like:

```
Name:
John Smith
Age:
twenty
```

How can you condense the code to use only one print statement without changing how the output looks?
(Hint: how can you concatenate each of the strings to create a single print statement)

2.20:

Part A: Translate the following python program into an english sentence

```
if (days_until_birthday == 0)
   print("happy birthday!")
else
   print(days_until_birthday + " more days until your birthday!")
```

Part B: Translate the following english sentence into a python program:

"If the person's birthday is less than or equal to 7 days away, print "your birthday is this week", otherwise, print "your birthday is ___ weeks away" (note: divide days_until_birthday by 7 to get weeks_until_birthday)

2.21:

Part A: Write an algorithm (i.e. steps written in human-friendly language) to find the volume of a sphere (A = 4/3?? r^3), given the diameter. Round ?? up to 3.1416 and use only the mathematical operations we have learned thus far (+, -, *, /, %).

Remember that your algorithm must abide by Order of Operations rules (otherwise known as PEMDAS).

Part B: Now turn your algorithm into a program in Python. End your program by printing, "the volume of the sphere is ____". Where the "____" is the value of your volume variable.

#Assume the value of the sphere's diameter has already been put into a #variable called "diameter" and use that variable for your calculations

Part C: Test your program in the python simulator. In order to do this, you must precede your program with a line that creates the "diameter" variable and sets it equal to some value of your choice. You can check that your volume algorithm is correct by searching "volume of a sphere with diameter [your diameter value]" at http://www.wolframalpha.com.
(Note: Answers may be slightly off since we rounded ??)

2.22: Give 5 words in Python which cannot be used as variable names.

2.23: What is camel casing? Give examples.

2.24: Why is commenting important while programming?

2.25:

Assuming there is no code before these lines, which lines of code would produce errors? Explain.

1. `"word" + 2`
2. `"word" + "word"`
3. `x = word`
4. `"word"/"word"`
5.
```
x = 4
y = x
z = x*x + y*x +z
```
6.
```
x = "word"
y = x
```
7.
```
x = "wprd"
y = x*2
```

2.26: What is the value stored in x after each of the following sequence of statements are executed. In cases you expect a syntax error, mention the same, and the reason for it.

1.
```
x = 5 + 6
```
2.
```
x = '5' + '6'
```
3.
```
x = '5' + 6
```
4.
```
b = 20
if = 10
x = if + b
```
5.
```
a = 10
2b = 20
x = a + 2b
```
6.
```
a = 10
_b = 20
x = a + _b
```
7.
```
b = 20
if_ = 10
x = if_ % b
```
8.
```
a = 10
_2b = 20
x =  _2b / a
```

2.27: How many times will the "hello" be printed by each of the following sequence of statements? What is the value of i at the end in each case?

1.
```
i = 0
while (i < 10):
    print("hello")
i=i+1
```
2.
```
i = 0
while (i <= 10):
    print("hello")
i=i+1
```
3.
```
i = 2
while (i <= 20):
    print("hello")
i=i+1
```
4.
```
i = 0
    while (i < 10):
        print("hello")
    i=i+1
```

2.28: What would be the output of following Python programs:

1.
```
a = 10
b = 20
if (a == b):
   b=a
else:
   if(a
      a = b
   else:
      b = a
print(a)
print(b)
```
2.
```
#sneaky indentations
a = 10
b = 20
if (a == b):
   b=a
else:
   if(a>b):
      a = b
else:
   b = a
print(a)
print(b)
```
3.
```
n = 5
i=1
# if(n == 5):
# print "hi5"
while(n >= 0):
   print(n*i)
   print(i)
   n = n - 1
```

```
    i = i * -1
print(n)
print(i)
```

2.29:

Write a program to print the following series:

1. 1 3 5 7 9 ... till n terms. (Initialize n as 10)
2. 1 2 4 8 16 32 ... till n terms. (Initialize n as 10)
3. 1 4 13 40... just before a term exceeds 10,000. (The series is actually 1 1+3 1+3+9 1+3+9+27..)

2.30:

Write a program to print the first n Fibonacci numbers. (Initialize n as 15). Start with 0 and 1 as the first two numbers. The next number is created by adding the previous two numbers. Thus, the series would go like this: 0 1 1 2 3 5 8 ...

2.31: Write a program to swap(exchange) the values of two variables a and b.

2.32: Write the output for the following:

1. ```
   print("\n\n\n\\")
   ```

2. ```
   x = 10
   y = 3
   while(y>0):
      while(x>0):
         x = x -1
         print("*")
      print("#")
      y = y-1
   ```

3. ```
   y  = "write this"
   x = 3
   z = 3*3*x-8
   print(z+"cs 252")
   print(y)
   ```

4. ```
   d =  75
   n = 75
   while(d=!n):
      print(d)
   print(d/n)
   ```

5. ```
   x = 2
   y = 4
   while(x>=0):
      while(y>0):
      print("^")
      if(y%2 == 0):
      print("$")
      y = y-1
      x = x -1
      print(x)
   ```

2.33: What is an off-by-one error and give an example from the text.
   ```