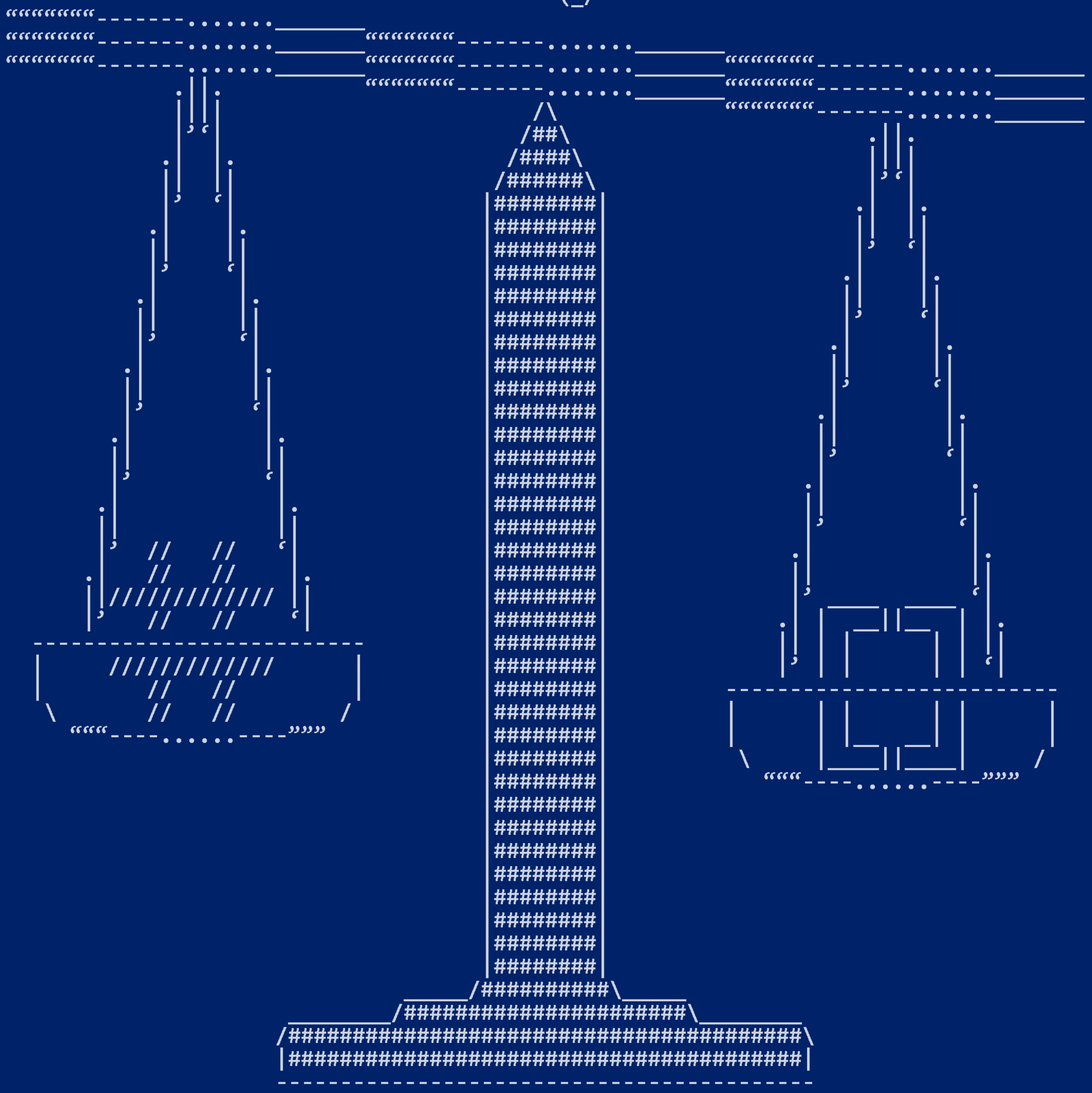
Computer Programming for Lawyers



Jonathan Frankle   Paul Ohm

Chapter 2

What is Programming?

2.1 As Easy as Baking

The notion of computer programming might seem intimidating at first. It can conjure images of hackers in dark rooms staring at neon green text flying across black screens, or mathematicians writing equations filled with Greek letters on chalkboards. Rest assured that programming is far more straightforward (and far less like a Hollywood thriller) than these impressions might suggest. Programming involves simply writing a set of directions for the computer to follow. Anyone who has ever read a recipe is exceedingly well prepared to program.

A computer program is, at its heart, like a recipe—a set of steps that, followed carefully in the correct order and supplied with the right ingredients, produce a result. Recipes are composed of a list of instructions, which you—the cook—read from top to bottom. Some instructions include simple tasks that must be completed exactly once: "place a bowl on the counter" or "add flour, sugar, eggs, butter, and chocolate chips." Others require repetition: "keep stirring until ingredients are well-mixed" or "scoop each teaspoonful of dough onto the cookie sheet." Some steps should be performed conditionally: "remove from the oven when golden brown." When put together in the proper order, they form a recipe that will lead to a batch of cookies.

```
Place a bowl on the counter.
Add flour, sugar, eggs, butter, and chocolate chips.
Until ingredients are well mixed:
    Stir ingredients.
While there is still dough left:
    Scoop a teaspoonful of dough.
    Place scoop of dough on baking sheet.
Place baking sheet into oven.
Remove from the oven when golden brown.
```

Unlike bakers, computers are exact; given the same ingredients and recipe, they will always produce identical cookies. When your program is correct, this means that you can rest assured that your computer will always execute it properly and arrive at the right answer. Unfortunately, this also means that a computer will faithfully obey your instructions even when they are wrong. Unlike a chef, a computer cannot infer your intent and will happily compute the wrong result or do something nonsensical if it is so commanded. In the recipe above, for example, a computer working on line 7 will be unconcerned if the scoops of dough are placed in one big pile rather than with the appropriate amount of space between them. Worse, a computer could take line 9 to mean that it should wait until the oven is golden brown; in the meantime, your house is likely to burn down.

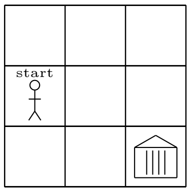
As you gain experience as a programmer, you will get more comfortable with detecting when the computer has interpreted your instructions differently from the way you intended. These *bugs*, as they are known colloquially in computer science, are a very common occurrence. Even expert programmers rarely write perfect code the first time they try to accomplish a task, and a large portion of programming time is actually spent *debugging*—gradually refining your initial attempt into something correct.

Our cookie recipe offers one final lesson about programming: natural human language is far too imprecise to guide the elaborate number-crunching machines that are modern day computers. In later chapters, we will explore vocabulary and syntax specially designed to unambiguously communicate programmer intent to computers—*programming languages*. In the meantime, however, we will first focus on the high-level computational thinking—the metaphorical art of recipe-writing—that you will employ every time you code.

2.2 Getting Started

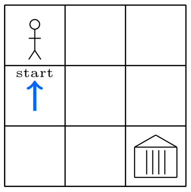
Now that you've begun to dip your toes into the water, we'll wade in a little deeper. Over the next few pages, we will introduce the act of programming more concretely and give a sense for what a program looks like. To do so, we will consider the plight of Larry, a hypothetical lawyer with a shockingly poor sense of direction. He appears in court often but struggles to find his way from his parking space to the courthouse. As his good friend, you have the honor of answering his phone calls when he gets lost and, thereby, of helping him find his way to court.

The world that Larry inhabits takes the form of a grid (see below). Today, Larry needs your help getting from his position on the far left of the grid to the courthouse in the lower right. You must give him directions to do so.



The only instructions you can give him are to move up, down, left or right. Metaphorically, we can think of Larry as our computer; he will follow any directions we give him to the letter. The directions we give him are programs —a set of instructions to be followed in a particular order to accomplish a task (getting Larry to a courthouse) based on a particular input (Larry's initial position). The *language* in which we write these programs has exactly four commands—the four directions we can tell Larry to move.

For example, if you told Larry to move up, he would take one step upwards.



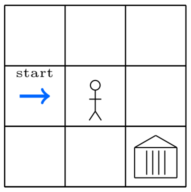
We can capture this instruction as a very simple program with exactly one command.

```
move up
```

Clearly, moving upwards isn't particularly helpful given his starting position. Instead, it would be more productive to tell him to move right.

```
move right
```

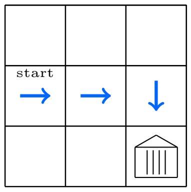
Doing so leaves him two steps away from the courthouse.



Unfortunately, no one instruction will get Larry all the way to the courthouse from his parking spot. We'll have to give him more than one command.

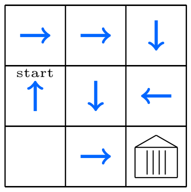
```
move right
move right
move down
```

Provided these instructions, he will follow them line by line until he runs out, at which point he will stop wherever he happens to be. By moving from his starting position right twice and then down once, he will successfully arrive at the courthouse.



This certainly isn't the only program that will get him to the proper destination. You might instead tell him to move up, right twice, down, left, down again, and then right to arrive at the courthouse.

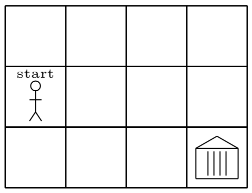
```
move up
move right
move right
move down
move left
move down
move right
```



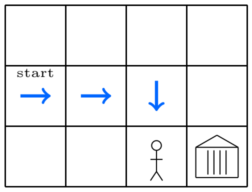
These instructions, while technically correct, are decidedly less direct than just moving right twice. You could even send Larry in circles or make him move back and forth several times before continuing on his way. Larry will be pleased to have made it to the courthouse, but he will be very frustrated that you unnecessarily wasted so much of his time in the process of doing so. This observation translates precisely to computer programs, where there are many ways to instruct a computer to solve any problem, but some methods are more efficient, easy to understand, or compact than others.

2.3 Loops

The next day, your ever directionally-challenged friend Larry finds himself lost after parking a little further away than yesterday. This time, he is four spaces too far from the courthouse.

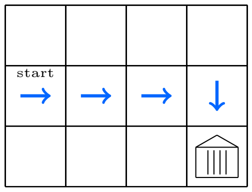


If he follows your instructions from yesterday, he will end up one space away from the courthouse:



Clearly, you will need to give him a new set of directions. You could give him the same sort of instructions you did yesterday, telling him to move right three times (instead of just two times like yesterday) and down once.

```
move right
move right
move right
move down
```

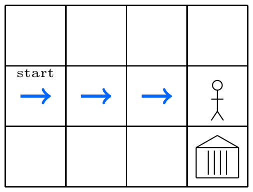


But what if he parks even further away tomorrow? Or what if he is lucky enough to park right in front of the courthouse? You will quickly get tired of giving directions that are so similar every day. Instead, it would be convenient if you could give Larry one set of instructions that will get him to the correct place every time he parks in the middle row on the grid. In programming, we would call this *generalization.* We should aim to write programs that apply as widely as possible. For example, instead of writing a program that downloads Securities and Exchange Commission (SEC) filings for Google and then writing separate programs each for Apple, Microsoft, and Facebook, it would be far better to write one program that asks you for a stock ticker and downloads the filings for whichever company the ticker corresponds to. Even if writing the more general version of this program takes a little extra time initially, the upfront time investment will more than pay for itself over the long run.

Let's apply that principle to Larry. Rather than spelling out each step he should take, you can tell him to keep moving to the right over and over again until he is one space above the courthouse, at which point he should move down exactly once.

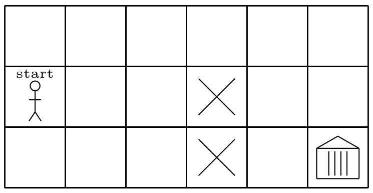
```
1 while not above the courthouse:
2     move right
3 move down
```

When he gets to the first line of these instructions, he will repeat whichever steps are indented immediately afterward so long as the condition "not above the courthouse" remains true. This indentation distinguishes the step(s) that should be repeated from those that come after the repetition is done, which return to the original level of indentation. In computer programs, we refer to this repetition as a *loop*. In this situation, he will move right, check that he is not above the courthouse, move right again, check again, and finally move right and check one last time.



At this point, he sees that he is above the courthouse, breaking the condition on line 1 and allowing him to proceed to the next direction after the loop, line 3. He will move down a single time and arrive at the courthouse.

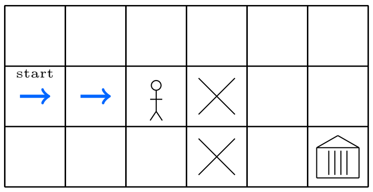
Unfortunately, you forgot that periodic construction in front of the courthouse can interfere with the instructions you have just given.



When Larry tries to follow your instructions, he will run into an error when he reaches the construction site. Specifically, he will call you on the phone to say:

```
Error on line 2: Tried to move right but was unable due to a
    construction site in the way.
```

You manage to find him here, to the immediate left of the construction site.

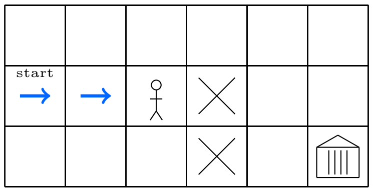


Tracing Larry's steps, he initially checked whether he was above the courthouse (line 1), found that he was not, and moved right (line 2). He returned to the beginning of the loop (line 1), checked again, found that he still wasn't above the courthouse, and moved right (line 2). Finally, he returned to line 1, again found that he wasn't above the courthouse, and tried to move right (line 2). However, his move right was impeded by the construction. Unable to follow your instruction, he simply gave up and stopped.

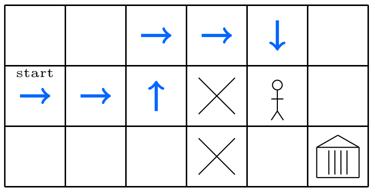
To fix the problem with the program you gave him so that it works in this new situation (to *debug* it), you will need to add some instructions to help him avoid the construction site. Your program will get a little more complicated, though. First, he needs to move right until he reaches the construction site, at which point he will need to go around it by moving up, to the right, and down. He can then continue moving right again just like before.

```
1  while not adjacent to construction:
2      move right
3  move up
4  move right
5  move right
6  move down
7  while not above the courthouse:
8      move right
9  move down
```

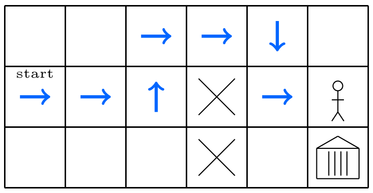
The first loop (on line 1) tells Larry to keep moving right until he is standing in a square that is adjacent to the construction site.



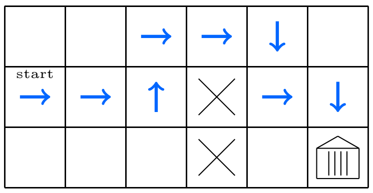
When he reaches the square next to the construction site, the condition "not adjacent to construction" will no longer be true. He will therefore exit the loop and proceed to line 3. Lines 3 through 6 navigate Larry around the construction site.



Finally, once he is on the other side, the instructions are the same as they were when there was no construction. Larry enters another loop (line 7) that instructs him to keep moving right so long as he is "not above the courthouse."



When he is above the courthouse, this condition no longer holds, so he exits the loop and continues to the final line, which tells him to move down and gets him to the courthouse.



Upon seeing these instructions, Larry will probably start to complain. There are a lot of steps, and it's difficult, at first glance, to understand why exactly each step is necessary. He has a point. If you ever need to go back and modify these instructions, it will probably take you a long time to figure out what you had in mind when you first wrote them. Anyone else who tries to read them will have an even harder time. When programming, presentation matters; code is a writing composition task, and future readers, yourself included, will be grateful if you make your programs as easy to understand as possible. This discipline, called *style*, will be a recurring theme throughout this book.

To meet this need, programmers have invented *comments*, which are lines of code that the computer (in this case, Larry) should ignore. They are written purely for the purpose of *documenting* the logic behind your code for yourself and other readers. To help Larry make sense of our instructions, we can add several comments, preceded by # signs, to the instructions. A # sign at the beginning of a line tells your computer (in this case, Larry) to ignore everything that comes afterward on that line. You also add a few empty lines to the instructions to space things out better.

```
# Move right until you reach the construction site.
while not adjacent to construction:
    move right

# Go around the construction site.
move up
move right
move right
move down

# Continue right.
```

```
while not above the courthouse:
    move right

# Arrive at the courthouse.
move down
```

As far as Larry is concerned, the program above provides the exact same directions as the program we wrote before. However, for you, the programmer, the version of the instructions with comments and empty lines is infinitely more readable. More importantly, a few weeks later, when Larry inevitably runs into trouble again and you have long forgotten the exact layout of the courthouse property, you will have an easy time getting back up to speed with the instructions you gave him before.

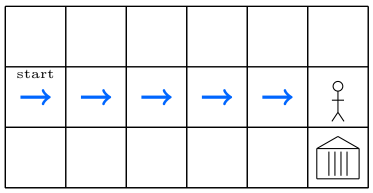
2.4 Conditionals

Thinking you have the problem solved, you leave Larry with the updated instructions. A couple of weeks later, he calls you, completely exhausted, hours after he was supposed to be at the courthouse that day. You ask where he is, only to find that he is halfway across the city. When you inquire as to why, he angrily responds that he followed your directions exactly, moving to the right until he was next to construction. Unfortunately, the construction crew had taken the day off, so, finding no construction near the courthouse, Larry kept moving right until he stumbled upon a construction site on the other side of town. You realize that your carefully crafted instructions were perfect so long as there was construction, but would fail on days when there wasn't.

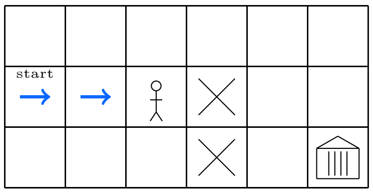
To solve this new problem, you need to give the hapless Larry instructions that will work no matter whether there is construction. First, you know that he needs to repeatedly move to the right. There are two conditions under which he should stop—if he is above the courthouse (on days when there is no construction) or if he is next to construction (on days when there is construction).

```
# First, move right as far as you can.
while not above courthouse and not next to construction:
    move right
```

This loop instructs Larry to move right repeatedly, but to stop as soon as either (1) he is above courthouse (he loops while he is "not above courthouse")



or (2) is next to construction (he loops while he is "not next to construction").



In other words, he should keep looping so long as neither of these conditions holds—so long as he is "not above courthouse *and* not next to construction."

Upon taking a second look at your instruction, you realize it is slightly ambiguous. On the one hand, Larry could interpret your instruction to mean exactly what is described above:

```
# First, move right as far as you can.
while (not above courthouse) and (not next to construction):
    move right
```

The parentheses indicate the logic of the instruction. Larry should keep moving right so long as he is both (1) not above the courthouse and (2) not next to construction. On the other hand, it might have been reasonable for him to think that you meant:

```
# First, move right as far as you can.
while not (above courthouse and not next to construction):
    move right
```

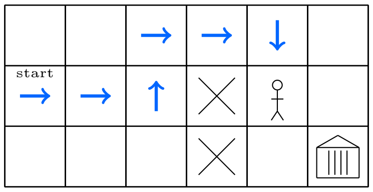
The logic of this interpretation takes a moment to parse. Copying literally from the instructions, Larry will keep moving to the right so long as he is neither (1) above the courthouse nor (2) not next to construction. Rearranging the logic to clarify the implications of this instruction, Larry will keep moving to the right so long as he is either (1) not above the courthouse or (2) next to construction. In other words, you just told Larry to wander into a construction site.

Since you clearly meant the first interpretation, you should add parentheses clarifying that intent. Although Larry might naturally have arrived at the same decision without the additional help, it is good style to ensure that your meaning is entirely unambiguous.

When Larry finally exits this loop, he could be in two possible situations. If he is next to construction, he needs to go around it and keep moving right; if there isn't any construction, however, he needs to stay exactly where he is. You can express this logic with a *conditional* command that will execute the indented instructions only if the condition holds.

```
# Go around the construction site if there is one.
if next to construction:
    move up
    move right
    move right
    move down
```

Unlike a loop, the indented code after a conditional executes only once. If the *condition* (in this case, “next to construction”) is true, Larry will follow the indented directions in order exactly once and then continue to line 10.



If the condition is not true (i.e., Larry is not next to construction), he will simply skip all of the indented directions and go straight to line 10. Since conditionals begin with the word “if,” they are often referred to as *if-statements*.

Finally, now that he has bypassed any construction that there might be, Larry can keep moving right until he is above the courthouse and then go down to finish his journey.

```
10  # Continue right.
11  while not above the courthouse:
12      move right
13
14  # Arrive at the courthouse.
15  move down
```

If he were already above the courthouse, the loop on line 11 will never execute, since the condition “not above the courthouse” does not hold; in that case, he would skip directly to line 15.

2.5 Summary

Although you would probably be quite frustrated with Larry by the end of this ordeal, he did nothing wrong. He carefully followed every direction that you gave him, sometimes so faithfully that your seemingly ironclad instructions were too imprecise.

Just like the directions you had to write for Larry, computer programs consist of lines of code, which a computer follows in order from start to finish. They can have loops to repeat operations until a particular exit condition is met. They can have conditionals to account for a wide range of possible inputs (the programmatic equivalent of construction sites) and change the behavior of the program accordingly. They also include comments to allow programmers to *document* their code. Python, specifically, requires each comment to be preceded with a `#` character and distinguishes the code inside loops and conditionals from the rest of the program using indentation.

In the next Part, we will begin writing real Python programs, which you should find to be a familiar and comfortable task after your experience with

Larry. First, however, you will need to install a few tools so that your computer can write and execute programs in Python.