3: Advanced Programming

3.1: Practical Programming

Summary: In the previous chapter, we introduced a simplified version of Python in order to communicate the fundamental ideas of programming. In this chapter, we'll explore more features of Python that allow us to complete the tasks that we had in chapter 2 more simply.

Definitions: readability, maintainability

In our journey down the tower of abstractions, we are currently discussing high-level programming languages, the tools with which large-scale systems are built. The goal of the previous chapter was to introduce the basic concepts of programming--things like syntax, semantics, the rigidity of a programming language, and how to think within a prescribed set of rules to accomplish basic tasks. For this purpose, we introduced a basic subset of Python.

As our goal is to drill down into the lower layers of abstraction rather than to give a full introduction to actual programming (which is easily an entire book all on its own), we'll only spend this brief chapter talking about more advanced features of Python, and then just enough to convince ourselves that Python can actually solve the real-world probems we posed in a sensible way. For example, think back to the code that simulated search-engine behavior, searching a file for a user-supplied string:

```
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)
```

There are many feature of this that would count as less-than-delightful:

- We aren't actually reading a file. We're pretending we got the words from a file into separate variables.
- We aren't actually getting user input from a prompt, but instead require users to change the program when they want to search for a different word.
- If we want to add another word to the file, we have to add a whole new variable and a whole other if block to test that word.
- We're repeating a lot of similar code. So if we decided, say, we wanted to print something else along with the name of the file that contained the query word, we'd have to change every single if-block. Quite the chore.
- If we had accidentally typoed one of the variable names, we may skip a word from the file and not realise it.

Several of these issues deal with the unrealistic nature of the example, which we'll certainly rectify in this chapter. However, some of them get at further concerns beyond realism and even beyond those of efficiency and scalability that we discussed in the previous chapter. Two such concerns are readability of code, and maintainability of code. These new concerns start to matter more when your code gets very large and has many users, but are worth at least thinking about for any code you write.

Readability refers to how easy it is to look at the code and both understand what it is meant to do and verify that it actually does that. Readability does not necessarily mean concision. Doing a complex task in one really clever line of code may obscure what task that line is actually performing. Breaking it into two or three more perspicuous lines may actually improve readability.

Maintainability refers to how easy it is to maintain the code. This includes things like:

- Identifying and fixing a bug.
- Changing the behavior of the code.
- Adding a new feature.

Code is said to be maintainable if it is written in such a way that makes these three tasks easy.

Our search engine code from the previous chapter fails miserably on both of these measures: As far as readability, it is relatively easy to figure out what the code means to do, but in order to ensure that it actually does what it is supposed to without error, we have to (among other things) read through every single if block to ensure there are no typos in variable names. Further, as far as maintainability, adding another word requires adding three lines of code. Adding another file with even just 100 words (the length of a short poem) requires 300 new lines to be added! While we did point out that short code is not necessarily more readable, code that is performing a relatively simple task shouldn't be hundreds of lines. And thinking about changing the behavior of the code, well, imagine changing it to have it print out its output in the format:

```
"Found a result: [filename]"
```

The internal groaning you just experienced speaks for itself.

This is not the fault of our methodology, but of our tools: Actual Python has more features than we introduced in the previous chapter, and can in fact accomplish this same task in maybe five lines of code regardless of how many files you want to search.

Before we embark on this chapter's brief journey into real-world programming, we indulge in one further remark on the principles of programming. The four high-level concerns that we've discussed (efficiency, scalability, readability, and maintainability), are not independent, and they sometimes war against each other. Sometimes the most efficient way to accomplish something is a highly obscure and clever algorithm that, when realized in code, obfuscates the purpose of that code entirely. It is also very common in large projects for reasonable amounts of efficiency to be sacrificed for the sake of having code that can easily be understood and modified in the future, say when a new programmer is hired and has to be introduced to the existing code that the other programmers have written.

Real-world programming almost always involves **trade-offs**, where we trade in, for example, a small amount of efficiency to make the code much more maintainable.

3.2: Running actual Python code

The same procedure for running code in the previous chapter carries over to this one.

3.3: Arrays

Summary: Arrays are a way of storing multiple values in a single variable. The individual values stored in an array can be accessed by subscript notation.

Definitions: subscript

Probably the most notionally offensive example from the previous chapter was our program for searching through a file. We weren't even really searching through entire files, but only through the first lines of the files, and that was already incredibly cumbersome. Imagine how massive a proper search program would be if it had to be written this way!

The method we used in that example was to store all the words in all the documents each in its own separate variable. What would be nice is to be able to store all the words for a single document in a single variable, but then to be able to get each word that is stored individually. There is in fact a way of doing this:

```
file1_words = ["twas","on","a","lofty","vases","side"]
```

Such a variable is called an array. All the variables we met in the previous chapter were storing one value under one name, so if we wanted to store multiple values, we needed to make several different variables, all with different names. But when dealing with a huge amount of data, as we expect to when making a search engine, this becomes inconvenient. So we have arrays, which let us store multiple values in one variable. The syntax for creating an array is

```
variable_name = [expression, expression, ...]
```

So, for instance,

```
x = 4
an_array = ["hello", 2, 9*x, x, "goodbye"]
```

declares an array with values

```
"hello"
2
36
4
"goodbye"
```

Once we have this set up, we can access the individual values stored in the array with the following **subscript** notation:

```
an_array[4]
```

This accesses entry number 4 in the array. So if we want to print this entry, we'd do

```
x = 4
an_array = ["hello", 2, 9*x, x, "goodbye"]
print(an_array[4])
```

As a result, we see printed:

```
goodbye
```

Great! Or is it? If you were paying attention, you'd notive that "goodbye" is the 5th entry in the array. What gives? In Python (as in most programming languages that have arrays) the number you put in the subscript is the offset from the first entry of the array. So

```
an_array[4]
```

means "start at the first value in the array and go 4 to the right". So, if instead you want the first word, you would start at the first word and go 0 to the right, i.e.:

```
an_array[0]
```

An array element such as `an_array[0]` is a variable like any other, so you can use them in expressions or assignments just like any other variable:

```
some_nums = [4,6,99,-1]
some_nums[0] = 5
print(some_nums)
print(some_nums[1])
some_nums[2] = some_nums[1] * some_nums[2]
print(some_nums)
if(some_nums[0] % 2 == 0):
   print("some_nums starts with an even number!")
else:
   print("some_nums starts with an odd number!")
```

So we see that we can access the various values stored in the array using this subscript notation, and any particular value in an array can be used in an expression or assignment just like any other variable. In particular, we know how to modify existing elements of an array, but what if we want to add elements to the array? Python allows you to concatenate arrays with the + operator:

```
array1 = [1,2,3]
array2 = [0,-1,-2]
big_array = array1 + array2
print(big_array)
```

prints

```
[1,2,3,0,-1,-2]
```

So what if we just want to put the value 4 on the end of the array?

```
array1 = [1,2,3]
```

We cannot do

```
array1[3] = 4
```

since there is no array1[3]. array1 has only three elements, and so trying to assign to its fourth will give a logic error. Instead, we need to assign array1 to be array1 plus a list containing just the element 4:

```
array1 = [1,2,3]
array1 = array1 + [4]
print(array1)
```

prints:

```
[1,2,3,4]
```

3.3.1: Strings as read-only arrays

We've actually seen some examples of arrays already, namely strings. If you think about all the things we just learned how to do with arrays--inspect and edit individual elements and concatenate arrays--these are operations we would equally want to be able to do to strings:

```
blah = "Hello World!"
print(blah[6])
print(blah[11])
```

prints

```
W
!
```

So, for instance, if the user has inputted 5 words, and we want to find all those words they've inputted that start with "a", we can now test this, since we know how to access the individual letters of a word. For example:

```
word = "betrothed"
print(word[0])
```

prints the first letter, "b". Now we don't want to print the letter, but to use it in a test. Recall that a test is something of the form

```
expression comparison expression
```

where the comparison we want is clearly == to test if the first letter of the word is equal to "a". Recall an expression is either a number, string, variable, or combination thereof. Here, we want to use the first value in the array called word.

Thus our test will be:

```
word[0] == "a"
```

and we get:

```
word = "betrothed"
if(word[0] == "a"):
   print(word[0])
```

which will not print anything, since word[0] is in fact "b". Now, putting this together with 5 random words (which will eventually be replaced with 5 user inputs):

```
word1 = "blithe"
word2 = "alphabetic"
word3 = "crocodiles"
word4 = "ate"
word5 = "asparagus"
if(word1[0] == "a"):
   print(word1)
if(word2[0] == "a"):
   print(word2)
if(word3[0] == "a"):
   print(word3)
if(word4[0] == "a"):
   print(word4)
if(word5[0] == "a"):
   print(word5)
```

This is somewhat reminiscent of our stupid search program, and indeed we make a slight initial improvement by storing all the words in an array:

```
words = ["blithe","alphabetic","crocodiles","ate","asparagus"]
```

But now careful--how do we get, say, the third word in this array? It is the first value in the words array, so it can be accessed like

```
words[2]
```

Indeed:

```
words = ["blithe","alphabetic","crocodiles","ate","asparagus"]
print(words[2])
```

prints

```
crocodiles
```

But now how do we get the first letter of the third word? The third word is words[2], and this is a string. How do we get the first letter of a string? By appending [0] to the name of the string. This string is called words[2], so indeed we can do:

```
words = ["blithe","alphabetic","crocodiles","ate","asparagus"]
print(words[2][0])
```

which will print the letter c. Thus we can modify our code:

```
```

```
words = ["blithe","alphabetic","crocodiles","ate","asparagus"]
w = words[0]
if(w[0] == "a"):
   print(w)
w = words[1]
if(w[0] == "a"):
   print(w)
w = words[2]
if(w[0] == "a"):
   print(w)
w = words[3]
if(w[0] == "a"):
   print(w)
w = words[4]
if(w[0] == "a"):
   print(w)
```

As with the search program, we'll learn how to improve this later.

Aside:

As a remark, we said that arrays can store any list of values, and that those values can be anything that is allowed in a variable. But an array is allowed in a variable! So can we store an array with arrays as its values? The answer is "yes", and this is in fact what we just did above. But we can be more direct about it:

```
a_big_array = [[1,2,3],[99,-5],[1,2,-1,0,0]]
```

Then to access the first value of the second array, we can do: a_big_array[1][0]. Recall that a_big_array[1] is the second value in a_big_array, which in this case is [99,-5]. So a_big_array[1] is itself an array, and to get its first value, we append [0].

This is called a 2-dimensional array. So, for example, if you wanted to store the state of a tic-tac-toe game, you could store:

```
board = [["X","O","_"],["_","X","O"],["_","_","_"]]
```

and you could print the board with:

```
print(board[0][0] + board[0][1] + board[0][2])
print(board[1][0] + board[1][1] + board[1][2])
print(board[2][0] + board[2][1] + board[2][2])
```

which prints

```
XO_
_XO
___
```

3.3.2: Search engine, revisited

Now that we have a basic mechanism for storing the words in our files in the search engine example, let us go through an example of searching one of them.

Recall we could store the words in an array like:

```
file1_words = ["twas","on","a","lofty","vases","side"]
```

Now we have an input word like

```
search_word = "the"
```

and we want to compare it to each word in the array. Of course, as before, we can do this separately like:

```
if(file1_words[0] == search_word):
   ...
if(file1_words[1] == search_word):
   ...
```

But that would be only marginally better than what we did before. But observe: All that changes between the various if lines is a number. So this is basically begging for a while loop with a counter, as we deployed in the previous chapter:

```
file1_name = "Ode on the death of a favorite cat"
file1_number_of_words = 6
file1_words = ["twas","on","a","lofty","vases","side"]
search_word = "the"
counter = 0
while(counter < file1_number_of_words):
   if(file1_words[counter] == search_word):
      print(file1_name)
```

```
    counter = counter + 1
```

In fact, we can use the "arrays of arrays" trick to search multiple arrays:

```
number_of_files = 2
file_names = ["Ode on the death of a favorite cat", "Elegy in a county churchyard"]
file_word_counts = [6,8]
files = [["twas","on","a","lofty","vases","side"], ["the","curfew","tolls","the","knell","of","parting","da
search_word = "the"
file_counter = 0
while(file_counter < number_of_files):
   word_counter = 0
   while(word_counter < file_word_counts[file_counter]):
      file = files[file_counter]
      if(file[word_counter] == search_word):
         print(file_names[file_counter])
      word_counter = word_counter + 1
   file_counter = file_counter + 1
```

3.4: Logical operators

Summary: Before, an if or while statement could only include one test at a time. In fact, there is a way to combine multiple tests using logical operators, specifying either that we want all of the tests to pass, or that we want any of the tests to pass.

Another feature of the previous chapter's examples that might have felt somewhat notionally offensive was the following: In order to check if player 1 won, we had to check if his score was bigger than player 2's. And then, in the event that it was, we then had to have an inside if block to check whether player 1's score was also higher than player 3's. And if that held true as well, then we could finally declare player 1 the winner:

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

One might hope that instead, we can just test two things at once using something like:

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

In fact, Python is pleasant enough that literally this actually works. More generally, we are allowed to combine the simple tests of chapter 2 using the logical operators and and or. So, for instance:

```
print("Give me a number 0-10, but no 8s--they taste weird")
x = 4
if(x > 0 and x < 10 and x != 8):
   print("good job!")
else:
   if(x == 8):
      print("Yuck!")
else:
   print("BETWEEN 0 AND 10!")
```

Or:

```
print("Give me a number, 0 or 1--your choice")
x = 4
if(x == 0 or x == 1):
   print("Thank you, kind human.")
else:
   print("CAN YOU REALLY NOT FOLLOW SIMPLE DIRECTIONS?")
```

Any combination of multiple tests with your choice of and and or is valid, but there are some pitfalls:

```
print("Give me a positive number, either odd or above 10: ")
x = 4
if(x > 0 and x > 10 or x%2 == 1):
   print("Thank you, kind human.")
else:
   print("CAN YOU REALLY NOT FOLLOW SIMPLE DIRECTIONS?")
```

This program is meant to accept any integer above 10, but also any odd number 1-9 as well. However, this program will actually even accept -1 as a valid input. The reason is that ands are grouped together before `or`s. Put another way, the test could equivalently have been written:

```
(x > 0 and x > 10) or x%2 == 1
```

So when confronted with x = -1, it goes:

```
(-1 > 0 and -1 > 10) or -1 % 2 == 1
   THIS IS FALSE        THIS IS TRUE
```

And so since one of the tests is true, the combined test has a value of true.

To fix it, we can clarify our meaning by using parentheses:

```
print("Give me a positive number, either odd or above 10: ")
x = 4
if(x > 0 and (x > 10 or x%2 == 1)):
   print("Thank you, kind human.")
else:
   print("CAN YOU REALLY NOT FOLLOW SIMPLE DIRECTIONS?")
```

3.5: Functions

Summary: Functions are a way of storing actual code so that it can be easily re-used in multiple places in a program.

Definitions: input (function), body (function), return value (function)

In the previous chapter, we often set up our code to have certain variables acting as the inputs to that code. For instance, recall our factorial code:

```
input = 12
answer = 1
counter = 2
while(counter <= input):
   answer = answer * counter
   counter = counter + 1
print(answer)
```

The advantage of this was that if we wanted to compute instead 60!, we could just change the first line to

```
input = 60
```

and it would happen. But if in our code we want to compute first the factorial of 50 and then later that of 90, say, we'll have to have this same code appearing twice, like:

```
input = 50
answer = 1
counter = 2
while(counter <= input):
   answer = answer * counter
   counter = counter + 1
print(answer)
...
input = 90
answer = 1
counter = 2
while(counter <= input):
   answer = answer * counter
   counter = counter + 1
print(answer)
```

This is unpleasant for a lot of reasons. If in the future we discover a subtle bug in our factorial code, or even just a faster way to compute factorials, then we'll have to change our code in two places rather than one, and will have to be careful to make the same changes to both bits of code.

This is redolent of the situation we had with variables--if we have some value, we don't want to recompute it anew each time we want to use it. Instead, we compute it once, give it a name, and then can access it at any time in the future just by using that name.

The idea of a function is to do the same thing again, except for code. That is, we can give a bunch of code a name, and then whenever we want to use it, we can just run it by using its name.

The actual syntax for doing this is a little more complicated than for variables, however. For one thing, blocks of code like the above have certain variables that are acting as inputs, and others that are acting as outputs. And when we use the code's name to run it, we want to be able to tell it what values to use for inputs, and where to store the outputs.

If we've managed to name the above code factorial, then to compute 50!, we should be able to say

```
factorial(12)
```

if we want to store this in some variable called x, then we can do

```
x = factorial(12)
```

In order to do this, we set up the above code in the following way:

```
def function_name(variable_name):
   [lines of code that comprise the function]
   return expression
```

The variable named in the parentheses after the function name is called the **input** to the function. Whenever the function is run, this variable can be given a new value, which can be used inside the function. The indented lines after the function definition are called the function's **body**. The expression in the final line will be the output of this function--called its **return value**.

A function name may be anything that a variable name could be.

To cause the functionto run, then, we simply use the function's name like:

```
function_name(expression)
```

The function's input variable will be set to the value of the expression, and then the code in the function will be run. When it reaches the a return line, the return value will be used as the value of this function call. That is, this function call is actually a new kind of expression, whose value is the return value of the function.

That was rather a mouthful, so let's go through it slowly with an example:

```
def mogrify(x):
   y = x+1
   return 5*y+9*x*x
z = mogrify(2)
print(z)
```

So the function is named mogrify, and its input is named x. Then when we run the line

```
z = mogrify(2)
```

which does as described above:

1. It sets the input variable to the expression specified (in this case, 2). So since the input variable was called `x`, it sets x to be 2.
2. Then it runs the code in the function body. In this case, it runs the line `y = x + 1`. Since x was 2, this sets y to be 3.
3. Finally, when it reaches the a return line, the return value will be used as the value of this function call. The line z = mogrify(2) is an assignment. So the return value of mogrify(2) is getting assigned to the variable z. Which value is it? According to step 2, it is the return value of the function call--in this case, `5*y+9*x*x`, or just 51.

So the program above will print 51.

Some bits of code require more than one value as input. To turn these into functions, we simply specify more input variables, separated by commas. For instance, if we want to specify two points on a line and want to compute the distance between them, we can have easily just subtract them. However, we have to be careful about what order we do this in. For instance, if the first point is 3 and the second is 5, then the distance is 2, which is the second point minus the first. But if the second were instead 2, we would have to subtract in the other order to get the correct answer of 1. So we use an if statement as follows (in this example, say the points are -1 and 6):

```
point1 = -1
point2 = 6
if(point1 < point2):
   answer = point2-point1
else:
   answer = point1-point2
print(answer)
```

But if we're going to be doing this often, maybe we want it in a function. Once again, the inputs are relatively clear--point1 and point2--and the output is once again perspicuously called `answer`.

```
def line_distance(point1,point2):
   if(point1 < point2):
      answer = point2-point1
   else:
      answer = point1-point2
   return answer
d = line_distance(3,5)
```

So when we call

```
d = line_distance(3,5)
```

we will start executing the first line of the function with point1 set to 3 and point2 set to 5. Then, because point1 is less than point2, the if block will be executed, setting answer to point2-point1, i.e., 2, and the else block will be ignored. Finally the return value of the function will be set to answer, i.e., to 2. And so when we say `d = line_distance(3,5)`, as we stated earlier, the function call itself--i.e., the line_distance(3,5) bit--is equal to the return value. Thus d is assigned the value 2.

3.5.1: Composition

Summary: Since a function call is an expression, and the input to a function call must be an expression, we can use a function call as an input to another function call. This is technically nothing new, but it is one of the confusing points of functions and so bears some discussion all on its own.

As we've described above, when a function appears in an expression, it is treated as being whatever value the function returns. So, for instance,

```
def f(x):
   y = x*x
   return x + y*y
print("I am going to compute something!")
print(2*f(3))
```

Let us consider that last line and how it makes sense: it is a print line. If we look at our chart, we see that inside a print() is allowed only an expression. So the question then is: is 2*f(3) a valid expression and, if so, what is its value?

Well, an expression is supposed to be a number, string, variable, function call (our new addition to the expression family), or combination thereof using operators +, *, -, /, %. This indeed is a combination of a number and a function call using the operator *, so it is a valid expression.

Now what is this expression's value? As we've seen in the previous section, the way to determine this is to break off from the flow of the program into a sort of sub-flow, where we only consider the function call and step through the function with its own private variables:

Current line	Why we went to this line	Variables after current line runs	Output so far
`def f(x):`	It was the first line of the program	(none)	(none)
`print("I'm going to compute something!")`	The previous lines just defined the function. Now that the function has been defined, we advance to the first line after this.	(none)	`"I'm going to compute something!"`
`print(2*f(3))`	It was the next line	`x = 3`	`"I'm going to compute something!"`
`y = x*x`	We just called the function f, so we go to its first line	`x = 3, y = 9`	`"I'm going to compute something!"`
`return x + y*y`	It was the next line	`x = 3, y = 9`	`"I'm going to compute something!"`
`print(2*f(3))`	Now that we know the value of `f(3)`, we return to the line that called it, fresh with the knowledge that (in this case), `f(3) = 84`	(none)	`"I'm going to compute something!" 84`

In this example the single-stepping was especially simple, and perhaps we could have understood by eye what would happen, but this mechanical breakdown of how function calls work becomes especially important when we have multiple functions being called on the same line, sometimes inside each other.

For instance, it would not be unheard of in a mathematics course (in particular, the following is not Python code) to see functions like:

```
f(x) = x^2+1
g(x) = x/13-1
h(x) = 77+x
```

and then to have to compute

```
f(2*g(h(0)+1))
```

You may already know how to do this intuitively (or possibly not--this is one of the things that trips some people up), but if we're going to do similar things with a computer, we're going to have to understand somewhat mechanically how these things can be done. So let's take this example.

It is clearly f of something, but before we can start plugging anything into f, we have to figure out: it is f of what? Well, we're plugging in 2*g(h(0)+1). But to compute this, we need to know g(h(0)+1).

OK, what is g(h(0)+1)? It's g of something, but before we can plug into g, we have to compute what we're plugging in--in this case, h(0)+1. And to compute that, we need to know h(0).

So what is h(0)? It is 77+0 = 77. Good.

So now we can start to go backward:

We needed previously to know g(h(0)+1). And now that we know h(0), we know this is g(77+1) = g(78). And g(78) we can compute by plugging in and we get 78/13-1 = 6 - 1 = 5.

Before that, we needed to know f(2*g(h(0)+1)). Now that we know g(h(0)+1) is just 5, we know that this is f(2*5) = f(10), which we can again compute by plugging in: f(10) = 10^2+1 = 101. So the answer is 101.

This example was, as problems from math class tend to be, contrived. But the principle we saw when solving it is one that we'll need now, namely that the first actual function we evaluated was the one that was furthest inside--h(0). Then, once we had that, we could compute what we were plugging into g, and so we computed g(78). Then, once we had that, we finally knew what we were plugging into f, and so we computed f(10).

This composition of functions happens all the time in programming and allows you to express certain things very concisely, provided you can keep your head on straight well enough to do it correctly.

You can watch the above steps happen in Python:

```
def f(x):
    return x*x+1
def g(x):
    return x/13-1
def h(x):
    return 77+x
print(f(2*g(h(0)+1)))
```

3.5.2: Scope

Summary: Variables defined in a function are destroyed (or revert to their previous values) once the function returns.

There is some slight danger here regarding variables that were created inside functions:

```
def f(x):
   y = 1
   return x + y
print(f(2))
print(y)
```

We might expect that when we call f(2) on line 4, we then go into the function which, among other things, creates the variable y with value 1. Then, when we go to print y on line 5, we should merrily see the value 1 as we expect. Instead, what we see is a logic error that the variable `y` was not defined. The issue, most basically, is that variables that were created inside functions don't continue to exist once the function finishes.

So our expectation was:

Current line	Why we went to this line	Variables after current line runs	Output so far
`print(f(2))`	It was the first line (after the definition of the function)	`x = 2` (x is the input variable of the function f, and so when we call f(2), this sets the input variable to 2)	
`y = 1`	We just called the function f, so we go to its first line	`x = 2` `y = 1`	
`return x + y`	It was the next line	`x = 2` `y = 1`	
`print(f(2))`	We've finished running f, so now we come back here	`x = 2` `y = 1`	`3`
`print(y)`	It was the next line	`x = 2` `y = 1`	`3` `1`

Whereas what actually happens is:

Current line	Why we went to this line	Variables after current line runs	Output so far
`print(f(2))`	It was the first line (after the definition of the function)	`x = 2` (x is the input variable of the function f, and so when we call f(2), this sets the input variable to 2)	
`y = 1`	We just called the function f, so we go to its first line	`x = 2` `y = 1`	
`return x + y`	It was the next line	(Variables from inside the function body no longer exist)	
`print(f(2))`	We've finished running f, so now we come back here	(none)	`3`
`print(y)`	It was the next line	(none)	`3` `[error message]`

But what if the variable existed before the function was called?

```
def f(x):
   y = 1
   return x + y
y = 2
print(f(3))
print(y)
```

It turns out that this prints

```
4
2
```

But surely what should have happened is that y got created on line 4. Then we called the function, which, among other things, modifies y to have value 1. Then later when we print y, it should have value 1, yes?

So now we get to what is actually happening with variables in functions, which is that every time a function is called, it gets its own environment, or set of variables that is independent from and doesn't affect the variables either outside the function or in future calls to the function. So we can in fact more accurately picture what's happening thus:

			Output

Current line	Why we went to this line	Variables after current line runs	so far
y = 2	It was the first line (after the definition of the function)	y = 2	
`print(f(3))`	It was the next line	x = 3 y = 2	
Call to the function f with argument value 3			
`y = 1`	We just called the function f, so we go to its first line	x = 3 y = 1	
`return x + y`	It was the next line	x = 3 y = 1	
f ends now with a return value of 4			
`print(f(3))`	We now return to the line that called the function with the return value of the function	y = 2 (The function is over, so any changes it made to any variables are discarded and any variables that existed before the function call retain their previous values	4
`print(y)`	It was the next line	y = 2	4 2

This behaviour also stacks when we call functions from within other functions. Consider, for example:

```
def f(x):
   y = 1
   return x - y
def g(x):
   y = x*x
   z = f(x+y)
   print(y)
   return z
y = 2
print(g(3))
print(y)
```

Here, `y` is initially set to 2. Then we call the function `g`, which sets `y` (in this case to 9) and in turn calls `f`, which sets `y` yet again, this time to 1. But when `f` returns and we go back to `g`, then the value of `y` that `g` set is restored. And finally, when `g` returns, the value of `y` that was set before the call to `g` is again restored.

3.6: Library functions

Summary: Now that we know about functions, we can use some functions that come pre-packaged with Python (or otherwise). These are called library functions.

Definitions: library function, module

Functions are useful to us when writing our programs because they help us organise our code more nicely. But one of the great advantages of Python is that it comes with many, many functions already written that perform various difficult tasks. From displaying graphics to a screen, to playing sounds, to getting mouse click information, there is a pre-written function for almost everything. These pre-written functions are called **library functions**.

As a note, one of the things we have conspicuously omitted from our explanation of Python to date was input. This is because input is performed mainly using library functions. So we are now ready to introduce various kinds of input functions.

In Python, library functions are contained in **modules**--separate pieces of code that can be loaded into our programs and that contain functions that can be used from those programs. In Python, the way to load a module is with an `import` line:

```
import module_name
```

This will make it possible to then call functions from that module, which will be named like this:

```
module_name.function_name
```

For example, the module that deals with input is called `input`. So if we import it, get notice that this adds two new functions: `input.get_num(prompt)` and `input.get_string(prompt)`. These functions will display a window asking the user to input a number or a string (respectively) and prompting them with whatever string was supplied in the argument. They return the value that was inputted. For example:

```
import input

name = input.get_string("Enter your name")
age = input.get_num("Enter your age")
print("Hello "+name + "!")
if(age % 2 == 0):
   print("Your age is even!")
else:
   print("Your age is odd!")
```

`input`

Function	Arguments	Return value

`input.get_num(prompt)`	`prompt` is a string that will be displayed in the prompt where the user may enter a number	The number that the user entered will be returned by the function
`input.get_string(prompt)`	`prompt` is a string that will be displayed in the prompt where the user may enter a string	The string that the user entered will be returned by the function

Part of the power of Python is that it has many modules with functions that perform many complex tasks, meaning programmers can rely on modules written by others for common functionality, and can therefore concentrate on designing the code specific to their application.

This is the greatest power of a high-level programming language like Python, and the feature that enables it to be most effectively used for practical tasks like programming search engines and video games and so on. We'll learn more library functions as we go through some examples.

3.7: More simple examples

3.7.1: Square Roots

One of the early examples of an algorithm in this text was one for computing the square root of a number. Since this is a common operation that comes up in distance calculations (among other things), we want a function that will do this for us. Something like:

```
def square_root(x):
   # ... something
```

Now we just have to fill in the something.

Let us recap our algorithm for computing the square root of x:

1. Start by changing s to be s = x/2.
2. Take whatever s is currently and add the number x/s to it, making that sum the new value of s.
3. Divide s by 2, making the result the new value of s.
4. If s*s is not yet as close as we want it to be to x, return to step 2 and continue from there.

We can phrase this now in terms of variables to start getting it to look a little more like a Python program:

1. `s = x/2`
2. `s = s + x/s`
3. `s = s/2`
4. If s*s is not yet as close as we want it to be to x, return to step 2 and continue from there.

So most of that translates literally into Python, but the last step is indicating some kind of a loop that contains inside it steps 2 and 3. So maybe it should be (approximately):

```
def square_root(x):
   s = x/s
   while(s*s is not yet close to x):
      s = s + x/s
      s = s/2
   return s
```

Great! So how do we tell if s*s is close to x? First, we have to specify how close we want them to be. Let's say we want the first 5 digits after the decimal to be correct. This means that we need s*s and x to be within 0.000001 of each other.

But how, in turn, do we test this? A good first guess would be `s*s-x < 0.000001`. The problem is that if somehow s*s is 4 and x is 8, then `s*s - x` will be -4, which is definitely less than 0.000001. Of course, what we want is not just for `s*s-x` to be small (negative numbers count as very small indeed), but for the *absolute value* of this quantity to be small.

But we cannot simply write `|s*s-x|` to get the absolute value in a Python program, as this is not valid syntax (check the syntax tree and behold the absence of any "|" operator).

However, if we had a function, say `abs_val(x)`, that computed the absolute value, then we could instead write `abs_val(s*s-x)` for this absolute value, and then our function would become:

```
def square_root(x):
   s = x/s
   while(abs_val(s*s-x) < 0.000001):
      s = s + x/s
      s = s/2
   return s
```

Of course, we don't have any `abs_val` function, but that's OK--now that we've solved the problem at hand supposing we had one, now we can go ahead and try to write that as a sub-problem.

The `abs_val(x)` function: The absolute value function should receive an argument x and return just x unchanged if x is positive. But if x is negative, we want to return "x, but made positive". And how do we make a negative number positive? Negate it! So it should return `x` if `x` is positive, and `-x` if `x` is negative. To wit:

```
def abs_val(x):
   if(x < 0):
      return -x
```

```
    return x
```

There is kind of a corner-case to think about here: Not all numbers are positive or negative. Specifically, 0 is neither, so we should make sure our function behaves properly when the argument is 0. But in fact it does return 0 in this case, so all is well.

Now that we have our `abs_val` function and our `square_root` function, we can put them all together into a square root calculating program:

```
import input
def abs_val(x):
   if(x < 0):
      return -x
   return x

def square_root(x):
   s = x/2
   while(abs_val(s*s-x) > 0.000001):
      s = s+x/s
      s = s/2
   return s

x = input.get_num("Enter x")
print(square_root(x))
```

Debrief: Take note of the strategy we used here: We wrote down the algorithm as usual, turned as much into code as we could, and then had a look at each of the pieces that we couldn't as easily turn into code.

Once we zeroed in on the problem of "How do we check if s*s is close to x?", we were able to write a description of what we wanted: "Absolute value of s*s-x needs to be less than 0.000001". At this point, we didn't know how to compute absolute values in Python, but we realised we could pretend we had a function that would do it for us, and could therefore defer the problem to later when we would have to write this function.

This is a very powerful way to use the abstraction mindset even when just working on the high-level programming layer:

- Take your problem, figure out a set of functions that, if you wrote them, would make the problem solvable
- Solve the problem using these functions
- Then go and write those functions that you needed.

This scores high marks for readability, as we haven't inserted a bunch of absolute value computation code into the middle of the square root function, so the square root function still looks like the square root algorithm as we wrote it originally. It also scores high marks for maintainability: If we want more accuracy in our computations, it is relatively easy to see what to change. If we think of a better way of computing absolute values, we can adjust that function independently of the square root code and the square root function will work just the same.

All told, this is nice, clean code. However, it can be even cleaner.

Square root, Mark II: ur absolute value function was pretty simple and surely someone else both needed it and wrote it before us, right? Indeed it is so: Python has a `math` library that includes many common (and some less common) mathematics-related functions ready to use. The one that computes absolute values is called `math.abs(x)`. So instead we could have written our square root program using it:

```
import input
import math

def square_root(x):
   s = x/2
   while(math.abs(s*s-x) > 0.000001):
      s = s+x/s
      s = s/2
   return s

x = input.get_num("Enter x")
print(square_root(x))
```

Square root, Mark III: But wait! If you ran the above program you might have noticed that there is a `math.sqrt` function. It turns out that "sqrt" is the common abbreviation for "square root". So in fact the `math` module already came with a function for computing square roots, and we could simply have done:

```
import input
import math

print(math.sqrt(input.get_num("Enter x")))
```

Debrief: This underscores an important feature of real-world programming and a very attractive feature of high-level programming languages, and especially of popular ones like Python: If you have a problem that is at all common, then because the Python is so widely used, someone has probably already written a function that solves it and packaged it into a module that's ready to use.

In this case, we had the math module, which we document in full here:

Function	Arguments	Return value
`math.abs(x)`	`x` is a number whose absolute value we want to	Returns the absolute value of the argument `x`

	compute.	
`math.sqrt(x)`	`x` is a number whose square root we want to compute.	Returns the square root of the argument `x`
`math.floor(x)`	`x` is a number whose floor we want to compute.	Returns the largest integer not greater than the argument `x`
`math.distance(x1,y1,x2,y2)`	`x1` and `y1` are the coordinates of one point in 2D space, and `x2` and `y2` are the coordinates of a second point.	Returns the distance in 2D space between the points (x1,y1) and (x2,y2).
`math.range(start,end)`	`start` and `end` are numbers indicating the bounds of the array to be returned.	Returns an array of numbers, counting up by 1, between `start` and `end`. By convention, this array excludes `end` so that `range(0,N)` is an array with `N` elements.
`math.solve_quadratic(a,b,c)`	The three arguments are treated as the coefficients of a quadratic polynomial `a*x^2 + b*x + c`	Returns an array containing all the solutions of the equation `a*x^2 + b*x + c = 0`

3.7.2: Factorial

OK, so we have a bunch of mathematics functions already written for us. This is convenient, but we notice that it doesn't include a factorial function. So at least that will not be wasted work. We already wrote some factorial code in the previous chapter, so we can wrap it up in a nice function here:

```
import input
def factorial(x):
   answer = 1
   counter = 1
   while(counter < x):
      counter = counter + 1
      answer = answer * counter
   return answer

print(factorial(input.get_num("Enter a number")))
```

That was relatively straightforward. In addition to this, we want to mention a few alternatives that convey a few new ideas.

Factorial, Mark II: If we step back and think for a moment about what the factorial is really doing, it is multiplying a bunch of numbers together. This is a rather general operation: We can imagine wanting to be able to take any list of numbers and multiply all of them--not just the list from `1` to `N`, which multiply to give `N!`.

To this end, we can write a more general function that takes in an array of numbers and returns the product of all its elements. This will look broadly similar to the factorial function: We maintain a `counter` and an `answer` variable, but instead of multiplying the answer by `counter` each time, we multiply it by `numbers[counter]`:

```
def multiply(numbers):
   answer = 1
   counter = 1
   while(counter < len(numbers)):
      counter = counter + 1
      answer = answer * numbers[counter]
   return answer

print(multiply([2,4,6]))
```

This has an off-by-one type of error. See if you can fix it! (Answer will be given later.)

This would allow us to very easily compute factorials if we could get an array of the numbers `1` to `N` easily. In fact, the `math` module has such a function: `math.range`. So we can easily do the following (this version has the off-by-one error fixed too):

```
import math
def multiply(numbers):
   answer = 1
   counter = 0
   while(counter < len(numbers)):
      answer = answer * numbers[counter]
      counter = counter + 1
   return answer

def factorial(x):
   return multiply(math.range(1,x+1))

print(factorial(5))
```

Debrief: This isn't really much better than the original factorial function, either in terms of maintainability or readability. It is somewhat more general in that we wrote a function that could do something that might be useful in some other context later.

Aside:

Factorial, Mark III: There is yet another way to write the factorial function. This method is inspired by the definition that is often given for the factorial in a mathematics textbook. To wit:

```
n! = n*(n-1)!, where we define 0! = 1
```

This is a somewhat odd definition, since to define the factoral of `n`, i.e., `n!`, it uses the factorial in the definition. This actually plays out just fine, however: Let's try to compute 4! just by using this definition and not any of our foregoing knowledge of what a factorial is:

- First of all, by the definition, 4! = 4*3!
- So we need to know 3!. But according to the definition, 3! = 3*2!
- But then we need to know 2!. In turn, by the definition, 2! = 2*1!
- Now we need to now 1!. Once again, 1! = 1*0!
- Finally, by the definition, 0! = 1
- So, now that we know 0!, we can find 1!: 1! = 1*0! = 1*1 = 1.
- Then we can find 2! = 2*1! = 2*1 = 2
- Then we can find 3! = 3*2! = 3*2 = 6
- Then we can find 4! = 4*3! = 4*6 = 24

So it all works out in the end.

This is a special kind of definition, where to define the function we use the function itself in the definition. Such a definition is called **recursive**. Obviously such definitions can fail to make sense. Suppose I try to define a new operator @ with a recursive definition like:

```
n@ = n * (n+1)@
```

Then if you try the same procedure as above to compute 4@, you will never finish it. (Try on your own, if you dare!)

But the definition of factorial does make sense, so we can mimic it with a function that, like the definition, actually calls itself:

```
import input
def factorial(x):
   if(x == 0):
      return 1
   else:
      return x*factorial(x-1)
print(factorial(input.get_num("Factorial of: ")))
```

We can track its behaviour by watching the "Outstanding calls" section of the simulator and observe that a call to factorial(4) elicits an outstanding call to factorial(3), and then to factorial(2), etc. just like our analysis of the definition above went through computing 4! via 3! and 2! and so on.

A function that calls itself in this way is called a **recursive function**. Recursive functions are relatively subtle beasts, and we won't concern ourselves any further with them in this text other than to have briefly mentioned their existence.

This particular example might look like a shining example of readability: It is the shortest implementation by far, and it reads almost identically to the official definition of factorial. However for someone who isn't already comfortable with recursive functions, figuring out its exact behaviour can be a challenge. A recursive function that is any more complicated than this can get very confusing indeed. So in fact, depending on the audience this could instead be a shining example of concision masquerading as readability.

3.7.3: List of prime numbers

In the previous chapter, we had an example of a function that tested whether a given number is prime. The question is now whether we can generate a list of all primes up to a given integer N.

One approach would be to start with a list of all integers 2 through N, test each one to see if it is prime, and delete the elements that are not prime. To do this, we first have to package up our code from the previous chapter into an `is_prime` function. We'll say it should return 0 if the number is non-prime, and 1 if it is prime.

```
import input

def is_prime(n):
   counter = 2
   is_prime = 1
   while(counter < n):
      if(n % counter == 0):
         return 0
      counter = counter + 1
   return 1

print(is_prime(input.get_num("Enter a number")))
```

Then we can use `math.range` to make a list of integers 2 through N and then use a while loop call out `is_prime` function on each element:

```
import input
```

```
import math

def is_prime(n):
   counter = 2
   is_prime = 1
   while(counter < n):
      if(n % counter == 0):
         return 0
      counter = counter + 1
   return 1

N = input.get_num("Enter a number")
nums = math.range(2,N+1)
counter = 0
while(counter < len(nums)):
   if(is_prime(nums[counter]) == 0):
      del nums[counter]
   counter = counter + 1
print(nums)
```

Debrief: Actually, this example also has a subtler off-by-one error resulting from what happens when you delete an element from an array. Go back and see if you can fix it. Show answer
Say the `nums` is `[2,3,5,7,8,9,10]` and `counter` is currently 4. Then we're testing `nums[counter]` for primality. Since `nums[counter] = 8` in this case, it isn't prime, so we delete it from the list, leaving `nums` as `[2,3,5,7,9,10]` and `counter` is still 4. Then we increment counter so it becomes 5. This meanqs `nums[counter]` is now 10, and we've skipped testing 9 entirely.This mean we will leave 9 in the list of numbers without ever testing it!

The fix is to only increment the counter if we didn't delete something:

```
import input
import math

def is_prime(n):
   counter = 2
   is_prime = 1
   while(counter < n):
      if(n % counter == 0):
         return 0
      counter = counter + 1
   return 1

N = input.get_num("Enter a number")
nums = math.range(2,N+1)
counter = 0
while(counter < len(nums)):
   if(is_prime(nums[counter]) == 0):
      del nums[counter]
   else:
      counter = counter + 1
print(nums)
```

Even once we get this working, however, there is a question of this program's efficiency. The algorithm runs by checking first if 2 is prime, then if 3 is prime, all the way up to `N`. For a given `x`, how many steps does it take to check whether `x` is prime? We run through all possible divisors of `x`, of which there are about `x` (precisely, there are `x-2`, but we can be approximate).

To count the total number of steps we take, we can count the approximate number of steps needed for each number between 2 and N:

- Check if 2 is prime: 2 steps
- Check if 3 is prime: 3 steps
- Check if 4 is prime: 4 steps
- Check if 5 is prime: 5 steps
- ...
- Check if N is prime: N steps

This gives us a total of `2+3+4+...+N` steps. We can simplify this to `(1/2)*N^2 + (1/2)*N - 1`. When N is large, N^2 is the only part of this expression that really matters (think: if N is around 1000, then (1/2)*N^2 is 500000, while (1/2)*N-1 is only 499). So we would estimate the number of steps as roughly (1/2)*N^2.

It bears a moment's reflection to think about what this means: If we want to compute all primes up to 1 million this way, we will end up needing to take about a half trillion steps. So how long will this take?

Remember first that the computer isn't actually a machine that reads and understands Python. Rather, the Python code is getting turned into more fundamenal ISA operations which are actually being fed into the computer. A reasonable computer will be able to run about 2 billion ISA operations per second. Each line of Python will in practice work out to a couple ISA operations on average--for now, let's optimistically approximate by saying each line is 2 operations.

With all these approximations in place, the computer will have a trillion operations to run, which should take about one trillion divided by two billion seconds, or about 500 seconds. That is perhaps tolerable, but relies on some pretty optimistic estimates. For instance, a computer can run 2 billion ISA operations per second, but a computer is usually running multiple programs at once and won't dedicate all of its power to running a single program's instructions all the time. If 20 serious programs are running, now maybe the calculation takes 20 times as long. The realistic conclusion to draw from all thisanalysis is that it will take on the order of several minutes, and possibly even up to an hour depending on exactly what situation it is running in.

Prime listing algorithm, Mark II: There is a classic algorithm for enumerating primes, which dates back to a Greek thinker called Eratosthenes of Cyrene (same guy who famously approximated the circumference of the Earth!).

The idea is: Start with a list of numbers 2-N. 2 is prime, but then all following multiples of 2 are not, so remove them from the list. The next number remaining in the list is 3, so it is prime, but then we can walk through and cross out all multiples of 3 from the list, as they are thus definitely not prime. The

next remaining item in the list is 5 (4 got removed when we removed multiples of 2), so it is prime, but we can cross out all multiples of 5.

To run through an example on 0-12:

- We start with list [0,1,2,3,4,5,6,7,8,9,10,11,12]
- 0 and 1 are not prime, so we "cross them out" by setting them to 0: [0,0,2,3,4,5,6,7,8,9,10,11,12]
- The first interesting number is 2. So we can cross out list items 4, 6, 8, etc., which we shall do by setting them to 0, leaving: [0,0,2,3,0,5,0,7,0,9,0,11,0]
- The next surviving number is 3. So we can cross out list items 6, 9, 12, etc., which we shall do by setting them to 0 (though some of them are already 0), leaving: [0,0,2,3,0,5,0,7,0,0,0,11,0]
- The next interesting number is 5, etc.
- When we're finished using this to cross numbers out, delete all the 0s, leaving: [2,3,5,7,11], a list of primes!

We can turn this algorithm into a program as follows:

```
import math
import input
N = input.get_num("See all primes less than: ")
ns = math.range(0,N+1)
ns[1] = 0
c = 2
while(c <= N):
   i = c+c
   while(i <= N):
      ns[i] = 0
      i = i+c
   c = c+1
   while(ns[c] == 0):
      c = c+1
i = 0
while(i <= len(ns)):
   if(ns[i] == 0):
      del ns[i]
   else:
      i = i+1
print(ns)
```

Aside: For people with a familiarity with calculus, we can actually approximate the time efficiency of this algorithm. First, it eliminates all multiples of 2 that are below N. There are N/2 of these, so this takes N/2 steps. Then it eliminates all multiples of 3, of which there are N/3. Then it eliminates all multiples of 5, which is N/5 steps. And so on, so the total number of steps is:

```
N/2 + N/3 + N/5 + N/7 + N/11 + ... + N/N
```

This is certainly less than the same series where we include 1/x for every x and not just the prime x:

```
N/2 + N/3 + N/4 + N/5 + N/6 + ... + N/N
```

We can factor out an N:

```
N(1/2 + 1/3 + 1/4 + 1/5 + 1/6 + ... + 1/N)
```

But the sum

```
1/2 + 1/3 + 1/4 + 1/5 + 1/6 + ... + 1/N
```

Approximates the integral from 1 to N of 1/x. This integral evaluates to ln(N). So in fact the expression

```
N(1/2 + 1/3 + 1/4 + 1/5 + 1/6 + ... + 1/N)
```

is just approximately

```
N * ln(N)
```

Because the sum is a lower Riemann sum (and ln(N) is an overestimate), and since we overestimated the original answer to begin with by adding in all those extra terms to the sum, this estimate of N*ln(N) is actually still slightly larger than the real number of steps. A more careful analysis would show that a better estimate is N*ln(ln(N))

So how many steps should it take, approximately, to get the primes up to a million?

```
(1 million) * ln(1 million) = (1 million) * 13.82
```

So approximately 14 million. By our back-of-the-envelope calculation above, this should take about 7 seconds. A vast improvement on our optimistic estimate of 500 seconds for the previous algorithm.

For those who skipped the aside, the upshot was that the code should take only about 7 seconds to compute the primes up to 1 million. This presents another example of a trade-off: The code

implementing this algorithm is kind of crunchy and hard to read. However, it is far superior in speed to the previous algorithm. One way to help this would be to put the code into a module function--maybe `math.get_primes`--so that while the code is there and hard to read, it is clear what it does, and most of the time we can use it, getting the full advantage of its speed, without ever having to look at it.

3.8: Applications

3.8.1: Search Engine

Now that we know about libraries, we can start working toward a slightly more realistic search-engine application. Before we get into details of an algorithm, we will find it useful to write down a slightly higher-level design for this program, specifying how it stores the data that is searchable, how it gets queries from the user, and what it gives back to the user.

Search engine design: For a start, we might try something like the following:

- The files that the search engine allows the user to search through are stored somewhere
- When the search engine starts, it reads these files and constructs arrays of all the words in the files
- Then it is ready for user queries, and it asks the user to input a query word
- It then prints out the names of the files that contain that word and asks the user for another query word.
- If the user enters "EXIT" as their query, then the program quits.

At a high level, we know how to do some of these tasks: Asking the user for input is the job of the input module. However, there are a few pieces we haven't learned yet. For example, how do we read files?

In our version of Python, we have a `file` module, with all the functions we could ever ask for pertaining to files:

Function	Arguments	Return value
`file.list()`	No arguments	Returns an array containing a list of filenames for files that exist.
`file.readline(filename, line_num)`	`filename` is a string specifying which file you want to read. `line_num` is a number specifying which line you want to read from that file.	Returns a string which is the requested line from the specified file.
`file.length(filename)`	`filename` is a string specifying which file you want to ask about.	Returns a number which is the number of lines in the specified file. Returns -1 if the file does not exist.
`file.write(filename,text)`	`filename` is a string specifying which file you want to append to. `text` is the text you want to write	Writes `text` to the file with name equal to `filename`, overwriting any data previously stored there. Returns 1 if the write was successful and -1 otherwise (for example, if the file does not exist).
`file.append(filename,text)`	`filename` is a string specifying which file you want to append to. `text` is the text you want to append	Appends `text` to the file with name equal to `filename`. Returns 1 if the append was successful and -1 otherwise (for example, if the file does not exist)
`file.create(filename)`	`filename` is a string specifying which file you want make.	Creates a new file with name `filename` and returns 1. If a file with that name already exists, it does nothing and returns -1.
`file.delete(filename)`	`filename` is a string specifying which file you want to delete.	Deletes the file with name `filename` and returns 1. If no such file exists, does nothing and returns -1..

So, for example, we can write some code that reads all the files:

```
import file
filenames = file.list()
counter = 0
files = []
while(counter < len(filenames)):
   name = filenames[counter]
   file_length = file.length(name)
   line_counter = 0
   file_contents = ""
   files = files + [""]
   while(line_counter < file_length):
      files[counter] = files[counter]+file.readline(name,line_counter)+"\n"
      line_counter = line_counter + 1
   counter = counter + 1
print(files)
```

We could proceed exactly in this fashion, writing fragments of code all day that do a piece of the task, and then at the end of the day work to string them together into one big program that behaves as specified. However, now that the task is getting a little complex, we may want to get a little more organized about it. One strategy to writing the large program is to start by subdividing the task into smaller tasks, as we have done. Then, rather than just writing code that solves these tasks, we

imagine that we have functions that solve the tasks, and then write the program assuming we have these functions, and then proceed to actually write the functions.

For example, the first concrete task that we can get from our rough outline above is that we have to read all the files into an array. We will imagine a function that does this job called `read_lines`. Let's say we feed this function an argument containing a list of filenames to read and it returns an array of strings, each containing the contents of the corresponding file.

Naturally, this function will be somehow constructed using the above code, but for now we'll write a function that mimics this behaviour in a more obvious way: it will just return an array of what could be file contents.

```
def read_files(names):
   return ["file1 words","file2 words","file3 has more words"]
```

Eventually we will replace this function with something that really reads the files (probably using a method based on the earlier file reading code), but for now, this function returns something of the same shape as the actual thing will, so we can use it in our later code. This idea of creating a function with the correct kind of behaviour that simulates the actual function's behaviour in a much simpler way is called creating a **mock up** (or sometimes just **mock**) of the function in question.

Now that we have got the file contents, the next task is to construct arrays of the words in the files. If we're going to search through the files for matching words, we would like it if the we had an array of all the words in the file so we could take our query word and simply compare it to all the elements of that array, and thereby determine whether the file contains the query word.

This will be the job of another function, say called `file_to_array` that will take in a file's text as a string and will somehow return an array consisting of all the words in the file. A mock of this function, then, might look like:

```
def file_to_array(text):
   return ["words","from","one","of","the","files"]
```

The next apparent task will be to actually get a query word from the user. We could write a function to do this, but we actually have a pretty obvious library function to do this already: `input.get_string`. So rather than mocking this out, we'll just use this in constructing our actual code.

The final task that we might extract from our outline is the business of actually searching through one of these arrays of words for a given word. This could be accomplished by a function called `search_file` which takes two arguments: One containing the array of words to search through and the other containing the word to search for. We have to have some way of letting this function indicate whether the word is actually in the file. One way would be to have it return 1 if the word is present, and 0 if not. For one possible mock up, we could then write:

```
def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0
```

Search Engine Algortihm: With all that preliminary organization in place, we can start designing the actual search engine algorithm. From a somewhat high-level perspective, this would look something like:

1. Get a list of filenames from `file.list`.
2. Get a list of file contents from `read_files`.
3. Turn the list of file contents into lists of words using `file_to_array`.
4. Get a user query using `input.get_string`
5. Use `search_file` to search through each of the lists of words that we got in step 3.

There are details missing from here: Step 3 and step 5 are probably going to be while loops, but we'll get to those details when we come to those steps.

Search Engine Programming: Let's start with some code that includes all our mocks and steps 1 and 2:

```
import file

def read_files(names):
   return ["file1 words","file2 words","file3 has more words"]

def file_to_array(text):
   return ["words","from","one","of","the","files"]

def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)
```

For step 3, we're going to have to loop through each of the strings in `file_contents` and use file_to_array to turn each of those strings into a list of words. We can store those lists of words in one big list called `index`, which will be the repository of words that we'll search later.

This entire procedure is an essentially familiar while loop pattern. The algorithm would look like:

1. Create `index` as an empty array
2. Set a counter, (say for simplicity we shall call it `i`) to 0
3. As long as `i` is less than then length of the list of file contents, do the following:
    1. Call `file_to_array` on `file_contents[i]` and add the resulting list as another element of `index`.
    2. Increment `i`.

This algorithm for step 3 translates pretty neatly into code:

```
import file

def read_files(names):
   return ["file1 words","file2 words","file3 has more words"]

def file_to_array(text):
   return ["words","from","one","of","the","files"]

def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)

# Step 3:
index = []
i = 0
while(i < len(file_contents)):
   index = index + [file_to_array(file_contents[i])]
   i = i+1
```

From here, step 4 is a pretty clear application of the input module:

```
import file
import input

def read_files(names):
   return ["file1 words","file2 words","file3 has more words"]

def file_to_array(text):
   return ["words","from","one","of","the","files"]

def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)

# Step 3:
index = []
i = 0
while(i < len(file_contents)):
   index = index + [file_to_array(file_contents[i])]
   i = i+1

# Step 4:
query = input.get_string("Enter your query: ")
```

Finally, step 5 will be another while loop, which we outline in algorithm form here:

1. Set a counter, (again just call it `i`) to 0
2. As long as `i` is less than then length of the index, do the following:
    1. Call `search_file` with arguments being the query word and the list of words in the `i`th file, i.e., `index[i]`.
    2. If the result is 1, print the filename: `filenames[i]`.
    3. Increment `i`.

Thus we get:

```
import file
import input

def read_files(names):
   return ["file1 words","file2 words","file3 has more words"]

def file_to_array(text):
   return ["words","from","one","of","the","files"]

def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)
```

```
# Step 3:
index = []
i = 0
while(i < len(file_contents)):
   index = index + [file_to_array(file_contents[i])]
   i = i+1

# Step 4:
query = input.get_string("Enter your query: ")

# Step 5:
i = 0
while(i < len(index)):
   print("Search for " + query)
   if(search_file(query, index[i]) == 1):
      print(filenames[i])
   i = i+1
```

We did also want to loop back and ask for another query word, but we'll add that feature later. For now, we have the more pressing task of implementing our mocked functions for real.

`read_files`: The `read_files` function we can implement by transplanting with only very minor changes the code from earlier that used the `file` module:

```
import file
import input

def read_files(names):
   counter = 0
   files = []
   while(counter < len(names)):
      name = names[counter]
      file_length = file.length(name)
      line_counter = 0
      files = files + [""]
      while(line_counter < file_length):
         files[counter] = files[counter]+file.readline(name,line_counter)+"\\n"
         line_counter = line_counter + 1
      counter = counter + 1
   return files

def file_to_array(text):
   return ["words","from","one","of","the","files"]

def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)

# Step 3:
index = []
i = 0
while(i < len(file_contents)):
   index = index + [file_to_array(file_contents[i])]
   i = i+1

# Step 4:
query = input.get_string("Enter your query: ")

# Step 5:
i = 0
while(i < len(index)):
   print("Search for " + query)
   if(search_file(query, index[i]) == 1):
      print(filenames[i])
   i = i+1
```

`file_to_array`: The `file_to_array` function is a little tricker. It will require taking these large strings consisting of the whole contents of a file and splitting them into words. We haven't talked much about complex operations on strings yet, but of course there is a module for this. Real-life Python (as well as just about every other programming language) has a very extensive library of string functions. In our version of Python, these are found in the `string` module, two of whose functions we will find very useful:

Function	Arguments	Return value
`string.simplify(text)`	`text` is the string to be simplified.	Returns a new string with all characters that aren't letters or spaces removed, all space characters (tabs, newlines, multiple spaces) turned into a single space character " ", and all letters converted to lower case.
`string.split(text,char)`	`text` is the string to be split, and `char` is the string to use as the separator.	Returns an array of all strings taken from `text` separated by instances of the string `char`. For example, `string.split("babacathe","a")` returns the array `["b","b","c","the"]`.

So in fact we can implement the `file_to_array` function as simply a call to the string module's functions:

```
string.split(string.simplify(file_text)," ")
```

Thus:

```
import file
import input

def read_files(names):
   counter = 0
   files = []
   while(counter < len(names)):
      name = names[counter]
      file_length = file.length(name)
      line_counter = 0
      files = files + [""]
      while(line_counter < file_length):
         files[counter] = files[counter]+file.readline(name,line_counter)+"\\n"
         line_counter = line_counter + 1
      counter = counter + 1
   return files

def file_to_array(text):
   return string.split(string.simplify(text)," ")

def search_file(word,word_list):
   if(word == "words" or word == "from" or word == "one"):
      return 1
   else:
      return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)

# Step 3:
index = []
i = 0
while(i < len(file_contents)):
   index = index + [file_to_array(file_contents[i])]
   i = i+1

# Step 4:
query = input.get_string("Enter your query: ")

# Step 5:
i = 0
while(i < len(index)):
   print("Search for " + query)
   if(search_file(query, index[i]) == 1):
      print(filenames[i])
   i = i+1
```

As a note, if we were unsure about whether `file_to_array` worked as intended, we could always pull it out and test it separately with a small code snippet like:

```
import string
def file_to_array(text):
   return string.split(string.simplify(text)," ")

print(file_to_array("Hello!  I am Bob, and this is my pet kangaroo."))
```

and observe that what happens matches our expectations. This is an element of maintainability called **testability**: The ability to test that your code works as intended not just by testing it all at once, but by taking out pieces and testing them individually.

`search_file:` Finally, the `search_file` function is a simple while loop remiscient of our many other examples of this to date. We also add in a while loop to keep asking for a new query until we get the "EXIT" query in this version of the code, making it our first complete search engine example:

```
import file
import string
import input

def read_files(names):
   counter = 0
   files = []
   while(counter < len(names)):
      name = names[counter]
      file_length = file.length(name)
      line_counter = 0
      files = files + [""]
      while(line_counter < file_length):
         files[counter] = files[counter]+file.readline(name,line_counter)+"\\n"
         line_counter = line_counter + 1
      counter = counter + 1
   return files

def file_to_array(file_text):
   return string.split(string.simplify(file_text)," ")

def search_file(word,word_list):
   i = 0
   while(i < len(word_list)):
      if(word_list[i] == word):
         return 1
      i = i+1
   return 0

# Step 1:
filenames = file.list()

# Step 2:
file_contents = read_files(filenames)

# Step 3:
```

```
index = []
i = 0
while(i < len(file_contents)):
   index = index + [file_to_array(file_contents[i])]
   i = i+1

query = ""
while(query != "EXIT"):
   i = 0

   # Step 4:
   query = input.get_string("Enter a word to search for, or EXIT to exit the program")
   print("Searching for: "+query)

   # Step 5:
   while(i < len(files)):
      if(search_file(query,files[i]) == 1):
         print(filenames[i])
      i = i+1
```

3.8.2: Game Console

Suppose now we want to write a game using Python. Now, there are plenty of games we could write with only text input and output--"guess a number", "hangman", "nethack", and "colossal cave adventure" are some classic examples. Colossal cave adventure in particular presents some very interesting programming challenges.

Modern games, however, have access to advanced graphics technology and don't have to rely quite as much on the user's imagination. These advanced technologies require correspondingly complex programming to take full advantage of, which is well outside the scope of this text.

For our game example, then, we'll meet somewhere in the middle with a very limited graphics display: a 10x10 grid of squares that can change color, display text, and are clickable. We call this arrangement our "lightboard", and it can be controlled by functions from the `lb` module:

Function	Arguments	Return value
`lb.set_color(x,y,color)`	`x` is the x-coordinate of the button to set, `y` is the y-coordinate of the button to set, and `color` is a string describing the color to set for in the button.	Sets the button at coordinates x,y to have color described in the `color` argument. Returns 1 if the setting was successful, and -1 if it failed (e.g. if x or y was too large).
`lb.set_text(x,y,text)`	`x` is the x-coordinate of the button to set, `y` is the y-coordinate of the button to set, and `text` is the text to put in the button.	Sets the button at coordinates x,y to have text equal to the first character of the string `text`. Returns 1 if the setting was successful, and -1 if it failed (e.g. if x or y was too large).
`lb.get_click()`	No arguments	Waits for the user to click the lightboard. When they do, this function returns an array containing, in order, the x coordinate of the click, the y coordinate of the click, the text of the button clicked, and the color of the button clicked.

For example, we can easily write a program that allows the user to turn a box red by clicking on it:

```
import lb
a = lb.get_click()
x = a[0]
y = a[1]
lb.set_color(x,y,"red")
```

Or a program that allows them to turn as many boxes as they want red:

```
import lb
while(1 == 1):
   a = lb.get_click()
   x = a[0]
   y = a[1]
   lb.set_color(x,y,"red")
```

(To get the full effect, change the number of steps to 1000 before pressing "step".) This is kind of like a really simple paint program. It is interesting, actually: previously we described infinite loops as a likely logic error, but now here we are writing one and it seems to behave sensibly. We could break the loop, however, by having a winning condition or a button we can press to exit or similar. For example, if we want the top right button to be the "quit" button, we can, instead of looping forever, loop as long as a variable (say we call it `done`) is set to 0. Then, when the right button is clicked, we'll set it to 1, which will cause the loop to stop looping:

```
import lb
done = 0
while(done == 0):
   a = lb.get_click()
   x = a[0]
   y = a[1]
   if(x == 9 and y == 0):
      done = 1
   else:
      lb.set_color(x,y,"red")
```

Interesting to note about this is that the top-left button has x coordinate 9 and y coordinate 0. This is a common feature of graphics programs, where the top left is the origin in the coordinate system and the x-coordinate describes how many steps you take to the right, and the y-coordinate describes how many steps you take down.

Simple game, version 1: We could go down the route of improving our paint program by allowing the user to select a color or something, but suppose that instead we want to make a game with a character that we can move around the screen. In this case, gray will represent the background and red will represent the player's location. Let us say the goal is to get to the bottom right corner of the screen.

Algorithm: To start programming this, we need to think about what data describes the current situation in the game. Since right now all that happens is a single player moving around with no obstacles, the situation can be summarised with two pieces of information: Where is the player, and have they made it to the destination corner yet?

The location of the player can in turn be described by their x- and y-coordiantes, so the situation of the game at any point in time can be stored in three variables, which we shall call `user_x`, `user_y` and `done`. These variables comprise what is called the game's **model**--that is, the information that describes competely the game's current setup (positions of all objects, scores, whatever), independently of how this data is displayed to the user. For example, if our game had a "save" feature, the variables that are part of the model would be the only values we needed to store, since we can recreate the entire setup from that information.

Now, just like before, we'll have a loop that continues until `done` changes, where we get the user's clicks. But this time, rather than blithely setting wherever they clicked to be red, now we'll test whether they clicked next to the player's current position or not. If they did, then we'll move the player to the spot they clicked. Specifically:

- Change the player's current position to gray.
- Update the player's position (`user_x` and `user_y`) to the location of the click.
- Change the player's new position to red.
- If the player has reached the bottom right corner, set `done` to be 1.

So how do we check whether the click was next to the player? We can use the `math` library, which includes a function for measuring distance: `math.distance`. If the distance between the click point and the player's position is exactly 1, then the user clicked one of the four adjacent spots.

```
import lb
import math
user_x = 0
user_y = 0
done = 0
lb.set_color(user_x,user_y,"red")
while(done == 0):
   a = lb.get_click()
   x = a[0]
   y = a[1]
   if(math.distance(x,y,user_x,user_y) == 1):
      lb.set_color(user_x,user_y,"gray")
      user_x = x
      user_y = y
      lb.set_color(user_x,user_y,"red")
   if(user_x == 9 and user_y == 9):
      done = 1
```

Note that if we want the player to be able to move diagonally, then those diagonal clicks will be a distance of sqrt(2) (or about 1.4) away, so we could enable this behavior by changing the test:

```
import lb
import math
user_x = 0
user_y = 0
done = 0
lb.set_color(user_x,user_y,"red")
while(done == 0):
   a = lb.get_click()
   x = a[0]
   y = a[1]
   if(math.distance(x,y,user_x,user_y) <= 1.5):
      lb.set_color(user_x,user_y,"gray")
      user_x = x
      user_y = y
      lb.set_color(user_x,user_y,"red")
   if(user_x == 9 and user_y == 9):
      done = 1
```

At this point, we can start adding things to the model: we can store an array of walls that the player cannot step through, of enemies that if the player touches they lose, of teleport locations, of horses the player can mount and get to move faster, of whatever we can think of! We'll include here a simple example with some walls the player has to navigate around:

```
import lb
import math

# Initialize model:
user_x = 0
user_y = 0
wall_xs = [0,0,0,1,1,2,2,2,3,3,3,3,3,4,4,4,5,4,6,6,6,7,8,8,8,8,8,8,8,8,9,2,3,4,5,4,1,6,6]
wall_ys = [3,8,9,0,5,0,1,2,1,2,3,4,5,1,1,4,1,5,7,8,3,4,7,4,3,2,6,8,6,8,7,8,8,8,6,8,6,5]
done = 0

# Initial lightboard setup:
lb.set_color(user_x,user_y,"red")
lb.set_color(9,9,"green")
i = 0
while(i < len(wall_xs)):
   lb.set_color(wall_xs[i],wall_ys[i],"blue")
```

```
    i = i+1

# Play loop:
while(done == 0):
    a = lb.get_click()
    x = a[0]
    y = a[1]
    color = a[3]
    if(math.distance(x,y,user_x,user_y) == 1 and color != "blue"):
        lb.set_color(user_x,user_y,"gray")
        user_x = x
        user_y = y
        lb.set_color(user_x,user_y,"red")
    if(user_x == 9 and user_y == 9):
        done = 1
```

3.8.3: Mobile Phone

For our final example of using Python to construct a realistically useful program, we turn to a common function performed by many mobile phones today: Giving directions. Specifically, we want to write a program which will ask the user for a starting street address and a destination address and will provide a list of directions between the two locations.

This is a rather daunting problem. Even if we had the raw map data for where all buildings and roads are located, it would still be an enormous challenge to locate the start and end addresses within this massive amount of data, let alone search all roads and paths and compare combinations of them to find the optimal routes. The algorithms that do this quickly are interesting and surprisingly accessible and may well appear in your later computer science study. For now, though, we will take advantage of the fact that Google has actually solved this problem already with Google maps.

When a person or a company creates an application such as Google maps, they of course also have to provide a way to use it. So you can take advantage of all their mapping work by going to google.com/maps. But maybe you're walking and just want to be able to speak your start and endpoint addresses into the phone, rather than go to a website and type it in. This would be using another program, different from the website, to access the same data. If our goal is to write that program, we have to have a way of accessing Google's map data from within a program, rather than by going to the website. Such a mechanism that allows not just people using a web browser, but other programs to access data and functionality contained in a given program (such as Google maps, in this case) is called an **application programming interface**, or **API**.

So Google maps has an API that we can use from our program to get directions from one address to another. In fact, it is better than that, as there already exists a Python module that performs the task of using this API: the `googlemaps` module.

Function	Arguments	Return value
`googlemaps.init()`	No arguments	This function does the work needed to set up communication with the Google Maps server. It must be called only once in the program, and must be before any other googlemaps functions are called.
`googlemaps.directions(start, end)`	`start` is a string with the address from which navigation should start, and `end` is a string with the destination address	The function returns an array whose entries are the steps in navigating from the starting address to the ending address.

For a simple example of using this, we call the `googlemaps.init` function first (as we always must). Then we call the `googlemaps.directions` function with whatever two arguments we please. This returns an array, which we loop through to print out each element, thereby supplying the directions:

```
import googlemaps
googlemaps.init()
dirs = googlemaps.directions("North/Clyburn, Chicago, IL","Centrailia, PA")
i = 0
while(i < len(dirs)):
    print(dirs[i])
    i = i+1
```

If you run this program, it should output the directions from The North and Clyburn Red Line station in Chicago to Centrailia, Pennsylvania.

We note in the above example that the `init` function didn't appear to really do anything. It didn't set any variables and it didn't output anything. However, as per the documentation, it is actually necessary--if we call the `directions` function without first calling `init`, we will get a runtime error. What this function does is actually I/O, but not keyboard input or screen output. Rather, it outputs the appropriate thing on the computer's network interface (wireless card or whatever) to set up a connection with Google's map service so that we can communicate with it with future calls. If we just call the `directions` function immediately without first calling `init`, then this will send our request to Google who will promptly respond with the computer equivalent of "Uh...who are you again?"

It is now a rather simple exercise to combine this with the input library to let the user provide the start and end addresses:

```
import googlemaps
import input
googlemaps.init()
start = input.get_string("Enter the starting address")
end = input.get_string("Enter the ending address")
dirs = googlemaps.directions(start, end)
```

```
i = 0
while(i < len(dirs)):
    print(dirs[i])
    i = i+1
```

3.9: Real-life Python versus Simulator Python

We've provided the inline simulator in this text for the ease of readers who are first getting into programming and/or have no prior programming experience. It is worth mentioning, then, that some intentional simplifications have been made to the Python language in the browser-simulated version compared to Python as it exists in the wild.

There are many small examples--we have disallowed multiplying integers and strings, whereas Python is very happy to evaluate `2+"hello"` to mean `"hellohello"`. Python also includes a `//` operator for performing integer division.

There are also corner-cases where the simulated Python will behave slightly differently than real-world Python regarding scopes.

One of the largest features of actual Python that we omitted is an organizational tool called "object-oriented programming". We talked some about the advantages of splitting code into smaller functions earlier, and object-oriented programming is one paradigm that takes this modularization of code to the next level, with all the advantages that it can bring.

The largest actual difference (as opposed to ommision) between our simplified version and Python as it is used in the real world, however, is the modules we've provided. Analogues of these modules do exist in real-world Python, but they do not all behave exactly as the ones we have included here. Some of them, such as the module for accessing Google maps, have many more features that make them more useful (for example, if you want to get walking directions instead of just driving directions), but also correspondingly more complex, so we have created and presented simplified versions here to avoid this complexity (as well as avoiding the sometimes arcane procedures required to install some of these modules).

Otherwise, however, the language you have learned reflects the basics of the actual Python language pretty closely, and the programming concepts and techniques all very much apply to all kinds of real-world programming, Python and otherwise. So go forth and write efficient, scalable, maintainable, readable programs, making trade-offs where necessary and commenting your code!

In the meanwhile, now that we've seen more convincing examples of complex systems being built using Python, we are ready to peer into the next levels of abstraction: How are these complex commands we are writing actually being run by a physical machine? And what is happening inside these mysterious modules? We might believe that math.sqrt is just another Python function we could have written, but what about file.read? What's actually happening there? These questions and more will be answered as we proceed onward down to the ISA level in our tower of abstractions.