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 further features of Python that allow us to complete the same tasks a 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 tool 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 its 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 give a full introduction to actual programming (which is easily a class 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 un-sensible:

- 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 concerns 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. 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 and maintainability.

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

For the moment, the only things described in this chapter that are supported by the in-browser simulator are the library functions we'll describe towards the end. This means that if you want to see the code here in action, you have to use an actual Python compiler from a command-line. Because this chapter is in some ways an excursion from our main journey, actually doing this is currently outside the scope of the text.

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"
9
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

```
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                                  # Now some_nums is [5,6,99,-1]
print(some_nums)                                  # Prints [5,6,99,-1]
print(some_nums[1])                               # Prints 6
some_nums[2] = some_nums[1] * some_nums[2]        # Now some_nums is [5,6,594,-1]
print(some_nums)                                  # Prints [5,6,594,-1]
if(some_nums[0] % 2 == 0):                        # Not true
   print("some_nums starts with an even number!") # Skipped
else:
   print("some_nums starts with an odd number!")  # Prints the string
```

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 add 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

Summary:

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--namely, 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"]
if(words[0][0] == "a"):
   print(words[0])
if(words[1][0] == "a"):
   print(words[1])
if(words[2][0] == "a"):
   print(words[2])
if(words[3][0] == "a"):
   print(words[3])
if(words[4][0] == "a"):
   print(words[4])
```

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]):
      if(files[file_counter][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 = 50
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(50)
```

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

```
x = factorial(50)
```

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)
```

This 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. So:

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

Then if later 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, a function call, when it appears in an expression, is treated as being whatever value the function returns. So, for instance,

```
def f(x):
   y = x*x
   return x + y*y
print("I'm 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 = 5 - 1 = 4.

Before that, we needed to know f(2*g(h(0)+1)). Now that we know g(h(0)+1) is just 4, we know that this is f(2*4) = f(8), which we can again compute by plugging in: f(8) = 8^2+1 = 65. So the answer is 65.

This example was, as problems from maths 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(8).

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.

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:

Current line	Why we went to this line	Variables after current line runs	Output 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`

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

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**.

Now, 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. This is done in the **Answer:**
simulator walkthrough (also linked on the sidebar). Have fun!