Controlling the Flow of Your Code

Introduction to Python Programming

# Introduction

In this module, we’ll explore the concept of *control flow*. Simply stated, control flow is the order in which statements execute in code. Another way to think of this is how code can lead to different paths of execution. The examples below will help illustrate how control flow works.

## Example 1

Imagine a shipping company that needs a very simple piece of code written to help them determine what they should charge to ship a package. The code will take the package weight (one input), and it will determine the shipping cost (one output). In our illustration, there are only two potential shipping costs: $5 for light packages and $10 for heavy packages. If the package weighs less than 15 lbs, it is ‘light,’ and costs $5 to ship. If the package weighs more than 15 lbs, it is ‘heavy,’ and costs $10 to ship. Therefore, the code is very simple. It only needs to determine if the package is less or greater than 15 lbs and then output either $5 or $10. Notice in this example, there are two paths in the code. One path leads to an output of $5, and the other path leads to an output of $10. So, the question for us is, “How do we write code that can take multiple paths?” This is what we will learn in this module.

## Example 2

Our second example asks us to consider code that will be used to determine the length of a text file. This code should run for 100 files and then output the length of each file. To accomplish this, the same line of code could be written 100 times, or the line of code could be written and then an additional code could have the first line of code repeat 100 times. For this illustration, the latter option is far superior because it is much less writing and much less code to maintain.

Now, let’s consider how programmers specify control flow through the use of statements.

# Programming Statements

The main programming structures, or statements, that allow programmers to specify the control flow are *if/else* statements, *for* loops, and *while* loops. These structures are present in almost all programming languages and not only in Python. We will examine each of these main programming statements in greater detail below.

*Note*: Although there are also other structures that fall under the “control flow” category (e.g., switch statements), these are not native to Python, nor necessary, so we will not cover them in this course.

## ‘If/Else’ Statements

With *if/else* statements, blocks of code are written that will only execute when a certain condition is true or not true. A good example of this is the shipping example above. If the package weighs over 15 lbs, the shipping cost is set to $10, and if the weight is under 15 lbs, the shipping cost is $5. Thus, this code has two paths, and the path that is taken is determined by a *condition*. If/else statements allow us to program these conditions.

The following is the Python code for this example:

|  |
| --- |
| package\_weight = 16 **if** package\_weight < 15:  shipping cost = 5 **else**:  shipping cost = 10 |

Now, let’s analyze this code.

1. On the first line, we assign the value 16 to the variable package\_weight.
2. Then, we start the *if/else* statement. The statement starts with **if**, and then we write a condition that must be evaluated to be *true* or *false*. In this case, we write ‘**if** package\_weight < 15‘, which means we are testing if the package\_weight variable is greater than 15. This will either be true or false. We must then end the **if** line with a colon or ‘:’ symbol or we will get an error. This is very important!
3. On the next line, we write the code we would like to execute when the *if* condition is true. This line ‘shipping cost = 5’ must be indented as seen in the example. Indentation is also very important because this is how Python knows whether or not this line should run. Thus, when the *if* condition is true, the indented line will run; if not true, it won’t run. Then, we simply set the shipping cost variable to the value of 5 if the *if* condition is true. It’s also important to note that though there is only one line in this example, there can be as many indented lines as needed.
4. Then comes the **else** statement. The *else* statement runs anytime the *if* statement is not true. In this case, the *else* statement will run if the package weight is greater than or equal to 15. Note that the **else** statement must also end with the ‘:’ symbol (or ‘**else**:’). As with the indented line under the *if* statement, the line indented under the **else** statement will only run if the *else* condition is triggered.

In the above explanation, it’s again important to emphasize the need to correctly use the colon or ‘:’ symbol and to correctly indent the lines of code. This is Python syntax, meaning that this format is what the Python interpreter expects, and it’s the only way Python will understand what to do. Other programming languages have *if/else* statements, but the syntax will be different. This difference may be similar to grammatical differences in languages such as English or Japanese.

### ‘Elif’ Statements

Let’s look at one more example, so we can introduce the *elif* part of an *if/else* statement:

|  |
| --- |
| package\_weight = 5 **if** package\_weight < 10:  shipping cost = 5 **elif** package\_weight < 15:  shipping\_cost = 7 **else**:  shipping\_cost = 10 |

In the above example, the **elif** statement is introduced. *Elif* is just short for “else if,” and it allows programmers to introduce another condition. Thus, in the above example, we now have three tiers of shipping costs.

You may be thinking, “Won’t a package that weighs 5 lbs trigger both the *if* and *elif* statements because 5 is less than 10 and also less than 15?” That is a good question! The answer is that once one of the conditions in the *if/else* statement is evaluated as true, all the other portions of the *if/else* statement are skipped. Therefore, since a 5-lb package will trigger the first statement and the shipping cost will be set to $5, the rest of the *if/else* statement will be completely skipped since the first portion was evaluated to be true.

For some additional understanding and practice, try copying the above examples into a Jupyter notebook and run them. Change the value of package\_weight to see what happens.

We will also cover *if/else* statements again in course lectures, but let’s move on to discuss *for* loops.

## ‘For’ Loops

*For* loops allow us to repeat code a certain number of times. They can also allow us to repeat code one time for every item in a list.

Let’s look at an example:

|  |
| --- |
| my\_list = [1, 2, 3, 'a', 'b', 'c'] **for** item **in** my\_list:  print(item) |

In the above code, we define a list. We then write a *for* loop that ‘loops over’ every value in the list.

*Note*: The best way to become familiar with these new concepts is to run them in a Jupyter notebook, so as you’re reading these instructions, it is highly recommended that you copy this code into a Jupyter cell and run it. (As mentioned above, you can also get more practice by adjusting the values and seeing what happens.)

After the my\_list variable is created, we start writing our *for* loop with the **for** keyword. The complete *for* statement is ‘**for** item **in** my\_list:’. What this means (in English) is “for each value in the list, run the following code. Use the variable name ‘item’ to refer to each value in the list as you loop through them.” Note that the *for* statement also ends in a colon or ‘:’ symbol. Again, this is very important.

In this example, we only have one line of code ‘inside’ the *for* loop. This is the indented line that is under the *for* statement. That line is ‘print(my\_value)’. So, the line ‘print(item)’ runs six times because there are six values in the list. Remember that the code runs once for each value in the list.

Now, we also have a variable name ‘item’ which we are using to refer to each value. So, on the first iteration of the loop, ‘item’ will have the value of ‘1’. On the second iteration of the loop, ‘item’ will have the value of ‘2’. On the third iteration of the loop, the value of ‘3’, etc. On the last iteration of the loop, ‘item’ will have the value of ‘c’.

We do not have to use the variable name ‘item’ and can use any name we want. For example, the following code is perfectly valid:

|  |
| --- |
| my\_list = [1, 2, 3, 'a', 'b', 'c'] **for** my\_value **in** my\_list:  print(my\_value) |

As mentioned, all the lines inside the *for* loop must be indented, and you can have as many lines as needed inside the *for* loop. For example, if we had three lines in the *for* loop, it would look like this:

|  |
| --- |
| my\_list = [1, 2, 3, 'a', 'b', 'c'] **for** my\_value **in** my\_list:  my\_value\_type = type(my\_value)  print(my\_value\_type)  print(my\_value) |

Sometimes, we may not want to loop code for every item in a list and may only want to loop code a certain number of times. For example, if we wanted to loop some code five times, we can do this in Python in the following way:

|  |
| --- |
| my\_list = [1, 2, 3, 4, 5] **for** num **in** my\_list:  print(num) |

This code will work, and it’s easy enough to type in ‘[1, 2, 3, 4, 5]’. But imagine if we wanted the code to loop 100 times. We don’t actually want to type out a list of numbers from 1 through 100. Thankfully, we can use the *range* function which generates an *iterator* (an iterator is like a list) that will iterate between any two numbers.

Try these following examples in a Jupyter notebook to see this in action:

|  |
| --- |
| **for** i **in** range(100):  print(num) |

|  |
| --- |
| **for** j **in** range(10, 20):  print(num) |

|  |
| --- |
| **for** k **in** range(56, 98):  print(num) |

## ‘While’ Loops

*While* loops are very similar to *for* loops except that they will repeat while a certain condition is true. They will stop when that condition is no longer true. Be careful, since they are not restricted to only running a certain number of times— they could run forever!

Let’s look at a simple example:

|  |
| --- |
| my\_num = 5 **while** my\_num < 10:  my\_num = my\_num + 1 |

In this example, we set the variable my\_num to the value of ‘5’. We then start our *while* loop. The *while* loop statement is ‘**while** my\_num < 10:’. This line can be translated as “while the value of my\_num is less than 10, run the following code.” Again, note that this line ends in a colon or ‘:’ symbol.

The code that is run by the *while* loop is the indented code that follows the *while* statement. In our example, this code is ‘my\_num = my\_num + 1’. This code simply increases the value of my\_num by 1. So, this line will repeat until my\_num is no longer greater than 10.

Again, it would be best to attempt to experiment yourself with this code in a Jupyter notebook. However, as mentioned, be careful not to accidentally code an infinite *while* loop as seen in the following example:

|  |
| --- |
| my\_num = 5 **while** my\_num < 10:  my\_num = my\_num - 1 |

Note that in the code above my\_num will always be less than 10. Thus, this loop will run forever. If this happens, you can stop the running loop by hitting the ‘stop’ button at the top of the notebook.

## Combining *If/Else*, *For* Loops, and *While* loops

Now that we have considered *if/else* statements, *for* loops, and *while* loops, let’s look at how we can combine them. By themselves, these structures are pretty simple, but combining them can lead to very rich code. It is relatively common that almost all the control flow in code is determined by *if/else* statements and *for* loops, with a few (rare) *while* loops sprinkled in. Let’s look at some examples of how these statements can be combined.

### Example 1

|  |
| --- |
| my\_list = [-4, -2, -1, 0, 1, 2, 3, 4] **for** num **in** my\_list:  **if** num > 0:  print('positive')  **elif** num < 0:  print('negative')  **else**:  print('zero') |

In the above code, we loop through a list of numbers. We have an *if/else* statement inside of the *for* loop, and on each loop the *if/else* statement determines if ‘num’ is positive, negative, or equal to zero.

### Example 2

|  |
| --- |
| package\_weights = [4, 10, 21, 16] package\_costs = [] **for** package **in** package\_weights:  **if** package < 10:  shipping cost = 5  **elif** package < 15:  shipping\_cost = 7  **else**:  shipping\_cost = 10   package\_costs.append(shipping\_cost) |

In this code, we are looping through a list of package weights and applying the *if/else* statement from the example in a previous section of this reading. On each loop, the code calculates the shipping cost and appends the value to the package\_costs list.

# Conclusion

In this module, we have explored how control flow works. We’ve considered several examples of how to use the main programming statements (*if/else* statements, *for* loops, and *while* loops) that allow programmers to specify the control flow. Since the use and combination of these statements are present in almost all Python programming, it is necessary to become proficient with using them. In upcoming modules, we will continue our discussion of these and other important features of Python.