Basic Functional Programming

Introduction to Python Programming

# Overview

In this module, we introduce functional programming and the related topics of lazy evaluation, lambda functions, and list comprehensions. Before we begin, keep in mind that functional programming is an intermediate/advanced topic, and it is not something that can be learned in one week. There are entire courses devoted to functional programming. Therefore, our goal in this module is to only introduce the topic of functional programming and cover a few specific functions that can be used in Python code. We will also become familiar with some specific Python skills that can be used in all of our programming (not only with functional programming). For example, we will learn about the *map* function, which many Python programmers use, even when they are using a procedural (or object-oriented) programming style and not a functional programming style.

# Types of Programming

Thus far in this course, we have used imperative and procedural programming. Before we take a closer look at functional programming, let’s consider some characteristics of imperative and procedural programming.

## Imperative Programming

*Imperative programming* is a style of programming that dictates what the computer program should do, step-by-step. Think of a recipe for baking a cake. Each step is outlined, and the entire recipe is one script that you follow from beginning to end. Similarly, you can write a program that is just one large set of instructions, i.e., one large ‘script,’ that does not import any other libraries or modules. You can do a lot with imperative programming. However, you should not use imperative programming for anything more than a simple script because the code can become very difficult to maintain and develop.

## Procedural Programming

*Procedural programming* builds upon imperative programming. Procedural programming encapsulates portions of the code into procedures, or functions, that allow you to create much more modular code. A great example of this is the example we used in the previous module when introducing functions. We saw how we could take the code that we used to determine a package’s shipping cost, put it inside of a function, and then reuse it throughout our code. We can even use these functions in completely different programs. An example of this is using functions from the Python standard library. These functions are encapsulated procedures that can be used everywhere. Procedural programming results in code that is much easier to write, maintain, and organize.

Now, consider the recipe example from the paragraph on imperative programming above. Let’s say a recipe book has recipes for several cakes with different flavored frostings, but the steps for mixing the cake flour are the same for each cake. Instead of rewriting the same steps for the cake flour in each recipe (this would be an imperative style), the recipe could simply refer to a page in the book that has the instructions for mixing the cake flour. This would be a procedural approach— each recipe referring to the same shared procedure for mixing the cake flour.

## Functional Programming

*Functional programming* is a style, or paradigm, of writing code and structuring programs that can be implemented in many different programming languages, including Python. There is no need to use a specific language or software when using functional programming. It is simply a way to write code using a fundamentally different approach from that of procedural or object-oriented programming.

Functional programming, as the name suggests, only uses functions. Code written in functional programming treats everything like a math equation (i.e., a function) instead of as a set of step-by-step instructions. Another way to say this is that purely functional programming uses only ‘expressions’ and does not use ‘statements.’ Therefore, functional programming is quite different from imperative and procedural programming.

# How Functional Programming Works

Let’s say we need to write code that will take a list of numbers and produce a new list that consists of the numbers in the original list multiplied by 2. That is, the code will take the list [1, 2, 3, 4] and produce the list [2, 4, 6, 8]. The following is an example of this code written in the imperative style:

|  |
| --- |
| my\_list = [1, 2, 3, 4]  my\_new\_list = [] **for** x **in** my\_list:  new\_val = x \* 2  my\_new\_list.append(new\_val) |

Note that this code loops through the list [1, 2, 3, 4] and then creates a new list, ‘my\_new\_list’, which contains the values [2, 4, 6, 8]. This code simply multiplies each value in the original list by 2 and then puts those values in a new list.

Also, notice how this code is like a set of step-by-step instructions. It is an algorithm of multiple steps and not just one equation. This code is imperative, just like the set of step-by-step instructions to bake a cake. Also, note how this code has to maintain the my\_new\_list variable as it adds items to it to produce the final result.

With functional programming, we approach this differently. We try to write a one-line expression that fulfills the desired functionality, and we can do this with the following code:

|  |
| --- |
| my\_result = map(**lambda** x: x\*2, [1, 2, 3, 4]) |

To explain this code, we will need to introduce the *map* function and the *lambda* expression, but before we do this, let’s appreciate how this code is only one expression and not a set of multiple steps that consists of both expressions and statements. An *expression* is a line of code that computes a value like a mathematical equation, while a *statement* is a line of code that ‘does something,’ like an *if* or *for* statement. Also note, in the example above, that there is no data that we need to keep track of besides the initial input data.

Another important point is that the above expression is a composition of functions. That is, the ‘map’ function receives the ‘lambda’ function as input. In other words, it is a function of a function. This will make much more sense when we introduce the map function below.

In fact, if we were to use a purely functional programming style, our entire program would be one large composition of functions without any intermediate mutable data to keep track of. The inputs would produce the same output every time. Having said that, you do not have to write in a purely functional programming style to use the map and lambda functions. You can write procedural code, or object-oriented code, and still use the map and lambda functions. This is what many Python programmers do. They use a procedural programming style that includes the use of the map function and lambda functions (as well as the *filter* function and *list comprehensions*, which we will learn about in this module).

If functional programming seems strange or doesn’t make sense at this point, don’t worry. As previously mentioned, our goal is to only introduce functional programming on a broad level. Now, we will transition to explain some specific functions that can be used in Python programming. The reason these functions are introduced along with functional programming is that these functions are heavily used in the functional programming paradigm.

## The Map Function

The *map* function takes two inputs: an iterable data structure and a function. Think of ‘an iterable data structure’ as a list, which is a data structure that contains items that can be iterated through. So, in simple terms, the map function takes two inputs: a list and a function. The map function then applies the function to each item in the list, and the output is a “list-like” output (but not exactly a list) that consists of the output of the function. Let’s look at an example:

|  |
| --- |
| **def** **multiply\_by\_2**(x):  **return** x\*2  my\_result = map(multiply\_by\_2, [1, 2, 3, 4])  my\_new\_list = list(my\_result)  print(my\_new\_list) |

In this example, we first define a function called ‘**multiply\_by\_2**’. This function is very simple in that it takes an input and then returns that input multiplied by 2. For example, if we input 5 into the function, it will return 10.

We then use the map function to apply this function, **multiply\_by\_2**, to every item in the list ‘[1, 2, 3, 4]‘. So, the output will contain values 2, 4, 6, and 8. We assign this output to the variable ‘my\_result’. This output is not a list, so we need to convert it to a list before we print it. That is what the line ‘my\_new\_list = list(my\_result)‘ does.

Let’s look at another example. Let’s say we want to convert a list of words to lowercase. We can do so like this:

|  |
| --- |
| **def** **to\_lower**(x):  x.lower()  my\_result = map(to\_lower, ['APPLE','BANANA']) my\_new\_list = list(my\_result) print(my\_new\_list) |

In this code, we define a function ‘**to\_lower**’. This function accepts a string and then converts it to lower case using the ‘lower’ method. We use the map function to apply the **to\_lower** function to the list ‘['APPLE','BANANA']’. The result will be [‘apple’, ‘banana’].

## Lazy Evaluation

In the examples above, we noted that the output from the map function is not a list, just similar to a list. So, what is it? The output is called a *map object*, and it does not actually contain the results. It just contains the ‘instructions’ for producing the results.

Consider the following code:

|  |
| --- |
| **def** **multiply\_by\_2**(x):  **return** x\*2  my\_map\_result = map(multiply\_by\_2, [1, 2, 3, 4]) |

The code above does not compute the results [2, 4, 6, 8]. Rather, it just produces a map object that contains the instructions on how to produce the results. But no computation has taken place, yet.

Now consider this example:

|  |
| --- |
| **def** **multiply\_by\_2**(x):  **return** x\*2  my\_map\_result = map(multiply\_by\_2, [1, 2, 3, 4]) **for** item **in** my\_map\_result:  print(item) |

Here, you will see that each item prints as expected in the *for* loop. That is, you will see ‘2’, ‘4’, ‘6’, and ‘8’ print. However, we may ask, “If my\_map\_result does not contain these results, why do the correct results print?” The reason is that the results are calculated “just-in-time.” That is, as soon as we request one of the items from the my\_map\_result variable, it is computed. This is called *lazy evaluation*, and it means that results are not calculated until they are needed. Also, once the map object has produced its results, it cannot produce them again. That is, you cannot loop through the same map object twice. Once you loop through it once and calculate all the results, the map object is ’empty.’ Using lazy evaluation can lead to more efficient code in certain situations, but for now, just be aware of what is happening. You don’t need to worry about restructuring your code so you can use the map function whenever possible— simply be aware that this method is just one tool available to you for code efficiency.

## Lambda Functions

In the first section of this reading, we introduced the following as an example of functional programming:

|  |
| --- |
| my\_result = map(**lambda** x: x\*2, [1, 2, 3, 4]) |

So far, we have explained the map function, but we still have yet to explain *lambda*. So, what is lambda? Well, let’s consider the following code example that we have seen a few times now:

|  |
| --- |
| **def** **multiply\_by\_2**(x):  **return** x\*2  my\_map\_result = map(multiply\_by\_2, [1, 2, 3, 4]) |

As we already know, this code applies the **multiply\_by\_2** function to every item in the list [1, 2, 3, 4]. This code is fairly concise but we still have more than one line of code. This is because we have to define the **multiply\_by\_2** function. But what if we could define this function in the same line of code that we use to call the map function? For example, what if we could do something like the following?

|  |
| --- |
| my\_map\_result = map({definition of the function that we want to input to map goes here}, [1, 2, 3, 4]) |

In this line of code, we want to not only pass the ‘name’ of the function to map, but the actual ‘definition’ of the function. This is what lambda functions allow us to do. Lambda functions allow us to define a function inline. They are often used when passing a function as an argument to another function, just as we do with *map*.

A lambda function is defined by writing the lambda keyword, the arguments that it takes, and then the line of code that is executed when the function is executed, as seen in the following example:

|  |
| --- |
| my\_result = map(**lambda** x: x\*2, [1, 2, 3, 4]) |

In this line, we have defined a lambda function. The lambda function takes one argument, ‘x’, and the equation that is executed when the function is called is ‘x\*2’. Lambda functions are sometimes called ‘anonymous’ functions because they are never given a name. Lambda functions are not appropriate for functions that you intend to reuse because you cannot reuse lambda functions. They need to be rewritten everywhere you want to use them. If you were to adopt a purely functional programming style, you would use lambda functions a lot. However, even procedural and object-oriented programmers can use lambda functions frequently. This is true for many programmers. They use lambda functions often even though they use procedural and object-oriented programming much more frequently than functional programming. As someone who is just learning Python, you only need to be aware of what lambda functions are and be able to use simple lambda functions.

In the lectures for this module, we will see other examples of map and lambda and also have an introduction to the filter function and list comprehensions.

# Conclusion

In this reading, we’ve considered some of the basics of functional programming. We’ve compared it to imperative and procedural programming styles and have looked at how some specific functions can be used in Python code. The map and lambda functions are useful examples of how functions can be used to simplify our programming.