Functional Programming Primer

Written by: Braun Brelin

Chapter 1. What is functional programming?

Functional Programming in a new type of programming paradigm.   Most of the popular programming languages today (Such as Java, C, C++, etc.) are called *imperative* languages.

For the purposes of this document, all coding examples are written in Python 3.

An imperative language is one in which a *series of statements* is used to reach a goal. For example consider the following program.

Example 1.

```
#!/usr/bin/python3

a = 10
b=5
c=a+b
print("c = ",c)
```

Each of the statements (a = 10; b = 5;, etc.) when executed changes the state of program in a defined way from assigning a value to a variable to calculating the value of c by dividing a and b and putting the resulting product into c.

Now let's see how this can be done using a functional programming paradigm.

Example 2.

```
#!/usr/bin/python3

def addnums(t):
   return(t[0]+t[1]);

print(addnums((10,5)))
```

Note that in this program, no variables were initialized or changed. All calculations were performed within the functions themselves and returned, including printing out the results.

What are the features of a functional programming language?

Let's take a look at the tools that allow us to write a functional program.

Functions are both first class objects and higher order objects.

What does it mean when we say that functions are first class objects? It means that the language treats functions as *first class citizens*.

This means is that functions are treated as ordinary variables with a return type. Because of this property, instead of needing specific variables to hold and change state, we can now substitute these variables with function calls..

Additionally, Python treats functions as *higher order* objects, which means that we can pass functions into other functions as parameters and insert functions as return values we see that in example 2, we can pass our addnum() function into the print() function as a parameter.

Let's consider another example. Here we want to sum up a list of randomly generated

numbers. Let's see how we would do this in an imperative paradigm.

Example 3.

```
#!/usr/bin/python3

import random

totalsum = 0
dataList = []
for i in range(0,9):
   dataList.append(random.randint(1,10))


for i in dataList:
   totalsum += i

print (totalsum)
```

Again, we see that we instantiate three variables, an integer called totalsum, a list(array) called dataList and an index variable i. In each case, we modify the variable itself and change the state of the running program.

Now let's look at the functional programming way of doing the same thing.

Example 4.

```
#!/usr/bin/python3

import random

def SumListOfRandomNumbers():
  return (sum([random.randint(1,10) for x in range(0,9)]))

print (SumListOfRandomNumbers())
```

Here we declare a function called SumListOfRandomNumbers(). Note that this function creates a list of randomly generated numbers from 1 to 10 by using a Python concept called *list comprehension*. This allows us to create a list in the same way a mathematician might do so. Again we see that no specific variables are created. The results of the list comprehension are directly sent to the sum() function as a parameter. The SumListOfRandomNumbers() function. returns the output of the sum() function.

The SumListOfRandomNumbers() is called a *pure function*. That is to say, it makes no changes to any variable outside of its inputs and outputs. For example, the following would illustrate a *non-pure* function.

Example 5.

```
#!/usr/bin/python3
dataList = []

def myNonPureFunction():
   dataList.append("a")
   dataList.append("foo")
   dataList.append("bar")

myNonPureFunction()
print(dataList)
```

MyNonPureFunction() is considered non-pure because it changes the state of a variable that is outside of its scope.

Note that it isn't feasible to create a program thatl has only pure functions. The functional programming paradigm considers any p with the outside world, for example Input/Output functions, to be changing the state of the program, therefore, the print() built in function is inherently inpure.

What are some of the advantages of functional programming?

Note that in example 4, we see that the code is significantly smaller and simpler than in the imperative version. Also, we can note that no variables are created, which means that there is no possibility of *side effects*. Side effects simply mean that it is possible that a change in the variable may affect the code execution at some point in the future.

Functional programs implement both closure and currying

Languages that enable functional programming generally have the concept of *closures*. What are closures? Closure are a way of defining and using a function inside another function. For example.

Example 6.

```
#!/usr/bin/python3
import math

def getQuadraticFormula(a,b):
        def CalculateEquation(c):
            print("a = ",a," b = " ,b," c = ",c)
            return ((-b  + math.sqrt(b * b - (4.0 * a * c)))/(2.0 * a))
        return CalculateEquation;


f1 = getQuadraticFormula(1.0,9.0)

print(f1(3.0))
```

Example 6 shows an example, both of *closure* and of *currying*. Note that the function *getQuadraticFormula*() has defined within it another function *CalculateEquation*(). Also note that the outer function returns a reference to the inner function. Thus, when f1 is assigned the return value from getQuadraticFormula() it is given a reference to the CalculateEquation() function.

The CalculateEquation() function also has access to the local variables in the outer function, in this case, a and b. We can now call the f1 function and pass it a single argument, in this case the value of 'c'. Currying simply means that as a programmer one can take a function that has multiple arguments and *break it down* into functions that have fewer parameters. In this case, Given that f1 already has the arguments for a and b, we can simply pass the c parameter and get the calculated result.

Functional Programming implements anonymous functions.

Functional programming languages implement anonymous functions, known as *lambdas*.

Example 7.

```
#!/usr/bin/python3

y =lambda z:z**z + 1
print(y(2))
```

The above example is a trivial use of anonymous functions known as lambdas. In this case, the lambda function is taking one argument, z, squaring it and adding one to the result. At least for Python, care should be taken not to make lambda functions too large and complex, lest it render the code unreadable.

We will see later how lambdas can be combined with other built in functions for increased flexibility and power when doing functional programming.

Functional Programming uses recursion

Recursion is a major component of FP. Recursion allow programmers to do things like iterate through a loop without having to have an index variable (which would mean needing to change the state of the program).

Here is an example of using recursion to find the factorial of an integer.

Example 8.

```
#!/usr/bin/python3

def factorial(x):
    if x == 1:
        return(1)
    else:
        return x * factorial(x-1)


print (factorial(5))
```

Note that we could use iteration to calculate the factorial, but by doing it via recursion, no new variables need be instantiated and no state is changed. Also, the program above is a particular type of recursion called *tail recursion* which means that the recursive function call is the last line of code in the function. Making a function tail recursive means that, internally, the compiler may be able to replace the recursion with an iterative (i.e. looping) function. This removes one of the dangers of recursion, namely, exhausting the stack memory on the computer. With tail call recursion, the compiler no longer has to save a new return address on the stack. It simply goes back to the address of the calling function, which in this case, is itself.

Functional programming implements and uses functions such as map,filter and reduce.

The map function explained

A *map* in a functional programming context is a way of using a declarative syntax to apply methods to collections such as lists. What do we mean by this? Let's look at a simple example.

Example 9.

```
#!/usr/bin/python3

upperlist=[]
lowerlist = ["foo","bar","baz"]

for elem in lowerlist:
   upperlist.append(elem.upper())

for elem in upperlist:
   print(elem)
```

This is the standard imperative way of creating a new list consisting of the transformed elements of the old list. In this case, we change the case of each element in the first list.

Now let's see how we'd do it as a functional program using the map() function.

Example 10.

```
#!/usr/bin/python3
def mapToUpperCase(elem):
   return(elem.upper())

lowerlist = ["foo","bar","baz"]

upperlist = map(mapToUpperCase,lowerlist)

for elem in upperlist:
   print(elem)
```

In this example, instead of manually using a for loop to iterate through the list, we call the map() function. The map function takes two arguments, a function and a collection (usually a list). It then calls the function for each element of the list, placing the return value as an element of a new list.

Addendum: In Python 3, which is what all of the examples use, the map() function returns a *map object*, basically, an iterator, whereas in Python 2, map() returned a list. This means that you can't use the [] method on a map object in Python 3. If you want an actual list object, you have to coerce it into a list like so:

upperlist = list(map(mapToUpperCase,lowerlist))

Note that with a simple function like the one shown above, we could have also written the map function like this:

Example 11.

```
#!/usr/bin/python3

lowerlist = ["foo","bar","baz"]

upperlist = map(lambda elem : elem.upper(),lowerlist)

for elem in upperlist:
    print(elem)
```

Here we embed the function directly into the map argument parameter list by using lambda anonymous functions. Note that in example 11, the code is even shorter and clearer than in the two previous examples.

The filter function explained

The *filter*() function is similar to the map() function, however, instead of doing a transformation, the filter function returns all elements of the original collection that match an expression supplied via a function. For example.

Example 12.

```
#!/usr/bin/python3

inputList = ["foo","whoo","doo","bar","baz","noo"]

def ooInElem(elem):
    return("oo" in elem)

print (list(filter(ooInElem,inputList)))
```

In example 12, we want a new list that contains all members of the old list which in turn, contains the string “oo”. We use the filter() function to call a user supplied function ooInElem() which in turn will return a boolean that indicates whether a given element of the inputList contains the “oo” string. If so, the filter function will place that element into the new function object (As with map(), the filter function in Python version 3 returns an iterable object rather than a collection. We use list() to coerce this object into a list).

The reduce function explained

The *reduce*() function takes two arguments and passes it to a function. The function will then take the first argument, the final sequence, and apply the function to the second argument, which is the input to the function. It will then return the result into the first argument. For example

Example 13.

```
#!/usr/bin/python3

from functools import reduce

print (dict(list(reduce(lambda output, current: output + [(current, ord(current))], 'abc', []))))
```

This code, while somewhat difficult to read, shows an example of the reduce() function. In this example, we take the input,'abc' and return as output to the empty list a tuple consisting of the input value (first a, then b and finally c) and it's ASCII value. Therefore, the output of the lambda function is:

[('a',97),('b',98),('c',99)]

Remember that the reduce function like the others we've seen, returns an iterable reduce object rather than a collection, so we then coerce this into a list. Finally, we use the dict() function to turn this from a list of tuples into key value pairs, where the first element of the tuple is the key and the second element is the value. So, the final output would look like this:

{'b':98,'a':97.'c':99}.

(Note that in a dictionary, like similar data structures in other languages, does not guarantee any type of sorted order by default).