Functional Programming

We have already seen OOPs in Python. Before we move to Data Science, let's talk about Functional programming.

Why this paradigm is needed?

```
x = 5
x = x * 2
x += 1
print(x) # the value of x keeps on changing here
```

Let's look at the other way of doing it.

```
x = 5
x2 = x * 2
x3 = x2 + 1
print(x3) # the value of every variable remains same here
```

Although, these variables are mutable, we are kinda treating them as immutable. Why is it called functional programming is because we can use functions to perform these operations.

```
def twice(x):
    return 2 * x

def inc(x):
    return x + 1

x = 5
x2 = twice(x)
x3 = inc(x2)
print(x3) # the value of every variable remains same here
```

The characteristics of functional programming is as follows:

- Treating the data as immutable.
- Will keep the data separate from the mutations (functions) or operations we perform on it.

This is different from OOPs because the data variables and actions we can take using them are grouped together in OOPS But in functional programming, we keep the data and actions performed on it separate.

Lambda functions

- Anonymous functions, declared without name.

```
double = lambda x: x * 2
print(double(4))
```

Declarative: Map, Filter, Reduce

Let's say, I have to apply the function which the doubles the value on every element of the list [1, 2, 3, 4] individually.

```
def double(x):
    return 2 * x

arr = [1, 2, 3, 4]
doubled_arr = [2*x for x in arr] # list comprehension, bad
doubled_arr = [double(x) for x in arr] # list comprehension, better

m = list(map(double, arr)) # good
```

Notice that we are not using double() here because that will mean that we are calling the function double and passing it's return value to the map function. We need to pass the whole function itself.

We can make this look more nice by using a lambda function here.

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

output = list(map(double, arr)) # good
output = list(map(lambda x: x*2, input)) # notice we didn't have to even
name the function
```

Remember that we use lambda functions to declare the anonymous functions. Any we create anonymous functions when you need a function for a short period of time.

Lambda functions revisited

Are also created when a function is declared when we need to pass is to or returning from a higher order function.

```
def multiply_by(num):
    return labmda x : x * num

multiply_by_1000 = multiply_by(1000)
multiply_by_100 = multiply_by(100)

print(multiply_by_1000(9))
print(multiply_by_100(9))
```

Iterators

Notice that we had to use list comprehension structure to generate a list from the map output. It is possible because, here the map object is actually an iterator.

- Iterator is any 'type' which has countable number of values and can be used with a 'for in loop' expression.
- Lists, tuples, strings, dicts and sets are all iterable (not iterator technically).
- Remember range(x) ? # map and range are iterator but others are iterable
- iter() function can be used to convert them (iterable objects) to iterator objects.

Now what does it take for an object to be iterator - it should essentially have a methods __iter__ and __next__ which should return the next value every time it is called. The 'for i in loop' automatically calls the __next__ method internally.

```
class Test:
    def __init__(self, limit):
        self.limit = limit
    def __iter__(self):
        self.x = 10 # caled when iteration is initialised
        return self.x
    def __next__(self):
        x = self.x
        if x > self.limit:
            raise StopIteration
        self.x = x + 1
        return self.x

for i in Test(15):
    print(i)

normal_oject = Test(10)
iterator_object = iter(normal_object)
print(next(iterator_object)) #try printing it multiple times
```

Generators in Python

Like a normal function but uses “yield" keyword whenever it requires to generate a value.

```
def simpleGeneratorFunc():
    yield 1
    yield 2
    yield 3

for value in simpleGeneratorFunc():
    print(value)

x = simpleGeneratorFunc()
print(next(x))
print(next(x))
print(next(x))
```

Let's take an example of Fibonacci generator.

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a+b

#x= fib(5)
# print(next(x)) # keep repeating it

for i in fibonacci(limit=5):
    print(i)
```

Filter Function

Filter function filters your iterable depending upon a condition.

```
f = list(filter(lambda x: x%2 == 0, arr))
```

But how it is related to functional programming? You see, the original arr remains same.

Declarative VS Imperative programming

```
declarative = list(map(lambda x: x*2, input))

imperative = []
for i in range(0, len(arr)):
    v = arr[i]
    imperative.append(v * 2)
```

Reduce

Unlike map, and filter, reduce doesn't operate on the elements of the iterator individually but takes two elements and use it's output as the first operand for the subsequent computation. It's called reduce because we have reduced the list of elements to one number.

```
from functools import reduce
r = reduce(lambda x, y: x+y, arr)
print(r)
```

Questions

Print the total length in meters

```
cms = [10, 100, 100, 0 , -5]
valid_cms = list(filter(lambda x: x>=0 cms))
meters = list(map(lambda x: x/100, valid_cms))
total = reduce(lambda x, y: x+y, meters)
```

Calculate maximum number from the list

```
arr = [100, 3, 4, 5, 1, 9, 18, 1000]
r = reduce(labmda x, y: max(x, y), arr)
r = reduce(lambda x, y: x if x > y else y, arr)
print(r)
```

Map function on more than two iterables

```
a = [1, 2, 3, 4, 5]
b = [4, 5, 3, 2, 7]
m = list(map(lambda x, y, z: x*y*z, a, b, c))
print(m)
```

map ignores extra elements if lists are of different sizes.

Zip function

```
z = list(zip(a, b)) # can be useful when working with coordinate geometery
z = dict(zip(a, b))
print(z)
```