Functional programming

Functional programming is a programming paradigm in which the primary method of computation is evaluation of functions.

Python is usually coded in an imperative way but functional programming is a declarative type of programming style. Its main focus is on "**what to solve**" in contrast to an imperative style where the main focus is "**how to solve**". It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.

Although, functional programming plays very small role in Python code but in some situations it's advantageous to use functional programming.

Concepts of Functional Programming

Any Functional programming language is expected to follow these concepts.

- **Pure Functions:** These functions have two main properties. First, they always produce the same output for the same arguments irrespective of anything else. Secondly, they have no side-effects i.e. they do not modify any argument or global variables or output something.
- **Recursion:** There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion.
- **Functions are First-Class and can be Higher-Order:** First-class functions are treated as first-class variable. The first-class variables can be passed to functions as a parameter, can be returned from functions or stored in data structures.
- **Variables are Immutable:** In functional programming, we can't modify a variable after it's been initialized. We can create new variables – but we can't modify existing variables.

Advantages of Functional Programming

The functional paradigm is popular because it offers several advantages over other programming paradigms. Functional code is:

- **High level:** You're describing the result you want rather than explicitly specifying the steps required to get there. Single statements tend to be concise but pack a lot of punch.
- **Transparent:** The behavior of a pure function depends only on its inputs and outputs, without intermediary values. That eliminates the possibility of side effects, which facilitates debugging.
- **Parallelizable:** Routines that don't cause side effects can more easily run in parallel with one another.

Comprehension

i. List comprehension ii. Dictionary comprehension

List Comprehension

List comprehensions are used for creating new lists from other iterables. This is an elegant way to create a list without using loops.

Example:

```
a = [3,5,6,7]
b = [x**2 for x in a]
b
```

```
[9, 25, 36, 49]
```

```
students = ["Sandeep", "Mudra", "Yutika", "Manjika"]
ls = [x for x in students if "i" in x]
ls
```

```
['Yutika', 'Manjika']
```

```
a = [3,5,6,7,44,78,57]
c = [x for x in a if x%2 != 0]
c
```

```
[3, 5, 7, 57]
```

Dictionary comprehension

Like List Comprehension, Python allows dictionary comprehensions. We can create dictionaries using simple expressions. A dictionary comprehension takes the form *{key: value for (key, value) in iterable}.*

Example:

```
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]

#  dict comprehension
myDict = { k:v for (k,v) in zip(keys, values)}

print (myDict)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
#another way
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]

#  dict comprehension
myDict = dict(zip(keys, values))

print (myDict)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
myDict = {i: i**2 for i in range(1,6)}
print (myDict)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
myDict = {x: x**2 for x in [1,2,3,4,5]}
print (myDict)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

map(), filter() and reduce()

These are three functions which facilitate a functional approach to programming. All three of these are convenience functions that can be replaced with List Comprehensions or loops, but provide a more elegant and short-hand approach to some problems.

All three of these methods takes two arguments: *(function, List)*.

The map() Function

The map() function iterates through all items in the given iterable and executes the function we passed as an argument on each of them.

```
#find square of elements of a list
a = [3,5,7,8,9]
b = list(map(lambda x:x*x, a))
print(b)
```

```
[9, 25, 49, 64, 81]
```

```
#find areas of circles if their radii is given
rad = [3,4,6,8,2.5]
area = list(map(lambda r : 3.14*r*r, rad))
print(area)
```

```
[28.259999999999998, 50.24, 113.03999999999999, 200.96, 19.625]
```

```
#find volumes of spheres if their radii is given
rad = [2,4,6,9]
volume = list(map(lambda r: round((4/3)*3.14*r**3), rad))
volume
```

```
[33, 268, 904, 3052]
```

```
#convert names in given list into uppercase
students = ["Sandeep", "Mudra", "Yutika", "Manjika"]
cap = list(map(lambda x: str.upper(x), students))
cap
```

```
['SANDEEP', 'MUDRA', 'YUTIKA', 'MANJIKA']
```

The filter() function

As the name suggests, filter() forms a new list that contains only elements that satisfy a certain condition, i.e. the function we passed returns True.

```
# separate positive and negative numbers from given list
a = [2,4,6,9,-5,-8,5,2,-11]
b = list(filter(lambda x: (x<0), a))
c = list(filter(lambda x: (x>0), a))
print(b)
print(c)
```

```
[-5, -8, -11]
[2, 4, 6, 9, 5, 2]
```

```
# separate even and odd numbers from given list
a = [2,4,6,9,5,8,19]
odd = list(filter(lambda x: (x%2!=0), a))
even = list(filter(lambda x: (x%2==0), a))
print(odd)
print(even)
```

```
[9, 5, 19]
[2, 4, 6, 8]

[2, 4, 6, 9, 5, 8, 19]
```

```
#select names that satrt with 'M'
students = ["Sandeep", "Mudra", "Yutika", "Manjika"]
start_with_M = list(filter(lambda x: x[0] == 'M', students))
print(start_with_M)
```

```
['Mudra', 'Manjika']
```

The reduce() function

reduce() works differently than map() and filter(). It does not return a new list based on the function and iterable we've passed. Instead, it returns a single value.

Also, reduce() isn't a built-in function anymore, and it can be found in the **functools** module.

```
from functools import reduce
# find sum of elements of a list
a = [2,4,5,7]
sum = reduce((lambda x,y:x+y), a)
print(sum)
```

```
18
```

```
# find largest elements of a list
a = [3,45,6,49,59,53,23]
large = reduce(lambda x,y: x if x>y else y, a)
print(large)
```

```
59
```

Immutability

Immutability in a functional programming paradigm can be used for debugging as it will throw an error where the variable is being changed not where the value is changed. Python too supports some immutable data types like string, tuple, numeric, etc.

Example:

```
# String data types
st = "NIET Greater Noida"
st[1] = 'K'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-7d5434dc6764> in <module>
      1 # String data types
      2 st = "NIET Greater Noida"
----> 3 st[1] = 'K'

TypeError: 'str' object does not support item assignment
```

Iterators

Iterator is an object that is used to iterate over iterables,

Iterables are anything that we can loop over. List, tuple, set and dictionary are iterables.

Iterator returns one element at a time while iterating over it.

iterator must implement two magic methods: __iter__() and __next__(). These two methods are collectively called iterable protocols.

An object is called iterable if we can get an iterator from it.

Iterables are objects which implement __iter__() method.

The iter() function(which in turn calls __iter__()) returns an iterator from iterables.

next() function(which in turn calls __next__()) is used to manually iterate through all the items of an iterator. next() method returns the next value in iteration and updates the state to the next value. When we reach the end and there is no more data to be returned, it will raise the ***StopIterationException***.

```
# __iter__() returns an iterator from iterables
a = [2,5,7]
value = a.__iter__()
print(value)
```

```
<list_iterator object at 0x0000019A7AB6F748>
```

```
# __next__() returns the next value in iteration
a = [2,5,7]
value = a.__iter__()
print(value.__next__())
print(value.__next__())
print(value.__next__())
```

```
2
5
7
```

```
# iter() and next() do the same thing
a = [2,5,7]
value = iter(a)
print(next(value))
print(next(value))
print(next(value))
```

```
2
5
7
```

```
# StopIteration Exception
a = [2,5,7]
value = iter(a)
print(next(value))
print(next(value))
print(next(value))
print(next(value))
```

```
2
5
7

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-19-8cdec3f6a3fc> in <module>
      5 print(next(value))
      6 print(next(value))
----> 7 print(next(value))

StopIteration:
```

To handle StopIteration Exception:

```
# To handle StopIteration Exception
a = [3,5,6]
value = iter(a)

while(True):
    try:
        item = next(value)
        print(item)
    except StopIteration:
        break
```

```
3
5
6
```

The for loop internally uses while loop to iterate through sequences.

```
a = [2,5,7]
for item in a:
    print(item)
```

```
2
5
7
```

Creating custom iterators

```
# custom iterator to produce even numbers upto max
class Even:
    def __init__(self, max):
        self.max = max
        self.i = 2

    def __iter__(self):
        return self

    def __next__(self):
        if self.i <= self.max:
            result = self.i
            self.i += 2
            return result
        else:
            raise StopIteration

x = Even(40)

for i in x:
    print(i, end = " ")
```

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

The advantage of using iterator is that they save resources. Iterators are powerful tools for dealing with the large set of data, if you use regular list to store these values, our computer may run out of memory. With iterator, however, we can save resources as they return only one element at a time, So, we can deal with infinite data with finite memory.

Generally, iterators are implemented using **generators**.

Generators

Python provides a generator to create our own iterator function.

A generator is a special type of function which does not return a single value; instead it returns an iterator object with sequence of values.

Creating a generator function is as easy as defining a normal function, but with a **yield** statement, instead of a return statement.

```
#Generator function
def fun():
    yield 1
    yield 2
    yield 3
for value in fun():
    print(value)
```

```
1
2
3
```

Generator is a function that contains at least one **yield** statement(It may contain other yield or return statement).

Difference between return and yield statement:

- return terminates the function entirely
- yield statement pauses the function saving all its states and later continues from there on successive calls.

```
# creating generator object
def fun():
    yield 1
    yield 2
    yield 3
x = fun()     # x is generator object
type(x)
```

```
generator
```

```
# accessing values through generator object
def fun():
    yield 1
    yield 2
    yield 3
x = fun()
# acessing method 1
print(x.__next__())
print(x.__next__())
print(x.__next__())
```

```
1
2
3
```

```python
# accessing values through generator object
def fun():
    yield 1
    yield 2
    yield 3
x = fun()
# acessing method 2
print(next(x))
print(next(x))
print(next(x))
```

```
1
2
3
```

```python
# accessing values through generator object
def fun():
    yield 1
    yield 2
    yield 3
x = fun()

#accessing method 3

for i in x:
    print(i)
```

```
1
2
3
```

```python
# accessing List element one by one to save memory
def fun(ls):
    yield ls[0]
    yield ls[1]
    yield ls[2]
ls = [5,8,17]
x = fun(ls)
for i in x:
    print(i)
```

```
5
8
17
```

Write a generator function to generate odd numbers within a given range.

```
def ODD_gen(start, max):
    n = start
    while n<=max:
        yield n
        n +=2

b = ODD_gen(7,11)

for i in b:
    print(i)
```

```
7
9
11
```

Write a generator function to print Fibonacci series.

```
# create generator to print fibonacci series

def fib():
    a = -1
    b = 1
    while(True):
        c = a+b
        yield c
        a = b
        b = c


f = fib()
n = int(input("Enter number of terms to be printed: "))
for i in range(n):
    print(next(f), end = " ")
```

```
Enter number of terms to be printed: 7
0 1 1 2 3 5 8
```

Write a generator function to print first 10 perfect square.

```
# create generator to print first ten perfect squares

def gen_sq():
    n = 1
    while n<=10:
        sq = n*n
        yield sq
        n +=1

x = gen_sq()

for i in x:
    print(i, end = " ")
```

```
1 4 9 16 25 36 49 64 81 100
```

Pipelining generators

A generator pipeline is taking more than one generators (expressions or functions) and chaining them together.

```
#pipelining generators
def fib(num):
    a = -1
    b = 1
    for i in range(num):
        c = a+b
        yield c
        a = b
        b = c

def square(num):
    for i in num:
        yield i*i


print(sum(square(fib(10))))       #pipelining
```

```
1870
```

Advantages of generators

1. Easy to implement than iterators
2. Memory efficient
3. Represent infinite stream of data
4. Pipelining generators.

Generator Expression

Simple generators can be easily created using generator expressions.

Similar to lambda functions which creates anonymous functions, generator expression creates anonymous generator functions.

The syntax for generator expression is similar to that of a list comprehension, but the **[ ]** brackets are replaced by **( )**.

The major difference between a list comprehension and a generator expression is that a list comprehension produces the entire list while the generator expression produces one item at a time.

They have lazy execution(producing item only when asked for). For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
a = [3,5,6,8,14,13]
b = [x**2 for x in a]          # List comprehension
c = (x**2 for x in a)          # generator expression

print(type(b))
print(b)

print(type(c))
for i in c:
    print(i, end = " ")
```

```
<class 'list'>
[9, 25, 36, 64, 196, 169]
<class 'generator'>
9 25 36 64 196 169
```

Closures

A closure is a nested function which has access to a free variable from an enclosing function that has finished its execution.

Nested function is a function defined inside another function. Outer function is called enclosing function and inner function is called nested function.

```
# Nested function but its not a closure.
def fun1(msg):
    def fun2():
        print(msg)
    fun2()

fun1("NIET")
```

```
NIET
```

A closure is a function object that remembers values in enclosing scope even if they are not present in memory.

A free variable is a variable that is not bound in the local scope.

Three characteristics of a Python enclosure are:

1. It is a nested function
2. It has access to a free variable in enclosing scope
3. It is returned from the enclosing function.

```
# Its a closure.
def fun1(msg):
    def fun2():
        print(msg)
    return fun2

fun = fun1("NIET")

fun()
```

```
NIET
```

In order for closure to work with immutable variables such as numbers and strings, we have to use **nonlocal** keyword.

By using **nonlocal** keyword, the variable inside enclosing scope becomes free variables.

```python
# closure to count number of calls
def counter():
    count = 0
    def inner():
        nonlocal count
        count += 1
        return count
    return inner

counting = counter()

x = counting()
print(x)

x = counting()
print(x)

x = counting()
print(x)

x = counting()
print(x)
```

```
1
2
3
4
```

Find total run made by a batsman, runs will be entered one by one.

```
# to find total run made by batsmen

def add_values():
    data = []
    def add(run):
        data.append(run)        #because data is a list which is mutable,
                                 #so, we need not to write nonlocal keyword
        total_runs = sum(data)
        return total_runs
    return add

run_made = add_values()

x = run_made(2)
print(x)

x = run_made(1)
print(x)

x = run_made(4)
print(x)

x = run_made(6)
print(x)
```

```
2
3
7
13
```

coroutines

Coroutines are designed to be an extension of generators.

Coroutines are basically functions whose execution can be paused/suspended at a particular point and then we can resume the execution from that same point later.

It implements two methods: **send( ) and close( )**.

Generators use the yield keyword to return a value from the function but in coroutine yield can also be used on the right hand side of an = operator to signify that it will accept a value at that point in time.

Generator function that implements send( ) and close( ) method are called coroutines.

Generators are data producer while coroutines are data consumer.

Example:

```
def fun():
    print("Hello started")
    a = yield
    print(a)

    b = yield
    print(b)


try:
    f = fun()
    next(f)
    f.send(20)
    f.send("NIET")
except StopIteration:
        print("Stopped")
```

```
Hello started
20
NIET
Stopped
```

The reason we used *next* before using *send* is we can only use *send* when we are at the yield checkpoint, and yield is on the right side of the expression. So to reach that first yield, we have to use the *next* function.

```
#Another Example
def fun():
    x = yield     # coroutine feature
    yield x       # generator feature
f = fun()
next(f)
result = f.send("Greater Noida")
print(result)
```

```
Greater Noida
```

Program to search a string using coroutine.

```
def searcher():
    txt = "Hello how are you class?"
    while(True):
        item = yield
        if item in txt:
            print("{} is in text".format(item))
        else:
            print("{} is NOT in text".format(item))
search = searcher()
next(search)
search.send("Hello")

x = input("enter what to search: ")
search.send(x)

x = input("enter what to search: ")
search.send(x)

search.close()
x = input("enter what to search: ")   #this is after closing coroutine.
search.send(x)
```

```
Hello is in text
enter what to search: class
class is in text
enter what to search: college
college is NOT in text
enter what to search: how

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
```

Now here comes an interesting application of coroutines. Suppose we want to switch back and forth between two functions.

```
def fun1():
    print("fun1 started")
    yield
    print("A")
    yield
    print("B")
    yield
    print("C")

def fun2():
    print("fun2 started")
    yield
    print("1")
    yield
    print("2")
    yield
    print("3")

a = fun1()
b = fun2()

next(a)
next(b)
next(a)
next(a)
next(b)
next(a)
next(b)
next(b)
```

```
fun1 started
fun2 started
A
B
1
C
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
```

Declarative Programming

Functional languages are **declarative** languages, they tell the computer what result they want. This is usually contrasted with **imperative** languages that tell the computer what steps to take to solve a problem. Python is usually coded in an imperative way but can use the declarative style if necessary.

Python allows us to code in a functional, declarative style. It even has support for many common functional features like Lambda Expressions and the map and filter functions.

However, the Python community does not consider the use of Functional Programming techniques best practice at all times. Even so, we've learned new ways to solve problems and if needed we can solve problems leveraging the expressivity of Functional Programming.