Functional Programming

Unit: 3

Problem Solving Using Advanced Python

Course Details
(B Tech 2nd Sem /1st Year)

NIET Greater Noida

GET FUTURE READY

Contents

- Map
- Filter
- Reduce
- Comprehensions
- Immutability
- Closures and Decorators
- Generators
- Co-routines, iterators
- Declarative programming

Course Objective

After you have read and studied this module, you should be able to:

- To learn the Object Oriented Concepts in Python.
- To learn the concept of map, filter and reduce functions.
- To impart the knowledge of declarative programming.
- To learn the concepts of closures ,decorators and generators.
- To explore the knowledge of comprehensions and co routines.

Course Outcome

Course Outcome (CO)	At the end of course , the student will be able :	Bloom's Knowledge Level (KL)
CO1	Define classes and create instances in python.	K1, K2
CO2	Implement concept of inheritance and polymorphism using python.	**K3**
CO3	Implement functional programming in python.	K3
CO4	Create GUI based Python application.	K2
CO5	Apply the concept of Python libraries to solve real world problems.	K3,K6

CO-PO and PSO Mapping

Mapping of Course Outcomes and Program Outcomes:

			Programming in Advanced Python									
CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3	3	2	2	1	-	1	-	2	2
CO2	**3**	**3**	**3**	**3**	**2**	**2**	**1**	**-**	**1**	**1**	**2**	**2**
CO3	3	3	3	3	3	2	2	-	2	1	2	3
CO4	3	3	3	3	3	2	2	1	2	1	2	3
CO5	3	3	3	3	3	2	2	1	2	1	2	2
Average	3	3	3	3	2.6	2	1.6	0.4	1.6	0.8	2	2.2

Mapping of Course Outcomes and Program Specific Outcomes:

Programming in Advanced Python				
CO.K	PSO1	PSO2	PSO3	PSO4
CO1	2	3	2	2
CO2	**2**	**3**	**2**	**2**
CO3	2	3	3	3
CO4	3	3	3	3
CO5	3	3	3	3
Average	2.4	5	2.6	2.6

Prerequisite and Recap

- Operators
- Loop
- Method
- Variables
- Class , Objects
- Constructor
- Data Hiding
- Inheritance
- Polymorphism

Topic Objective

- After you have read and studied this topic, you should be able to
  - Understand the concept of object oriented systems
  - Create closures and decorators in a program.
  - Understand the concept of Declarative programming.
  - Understand how comprehensions and co routines are done
  - Understand immutability using python

Functional Programming(CO3)

- Functional Programming is a ***programming*** paradigm with software primarily composed of functions processing data throughout its execution.

- **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.

Functional Programming Characteristics (CO3)

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports ***higher-order functions*** and ***lazy evaluation*** features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism

Advantages of Functional Programming (CO3)

- ***Bugs-Free Code*** − Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.

- ***Efficient Parallel Programming*** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.

- ***Efficiency*** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.

- ***Supports Nested Functions*** − Functional programming supports Nested Functions.

- ***Lazy Evaluation*** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Comparison between FP & OOPS(CO3)

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.

Anonymous or Lambda Function

A function without a name is called as Anonymous Function. It is also known as Lambda Function.

Anonymous Function are not defined using def keyword rather they are defined using lambda keyword.

Syntax:-

lambda argument_list : expression

Ex:-

lambda x : print(x)

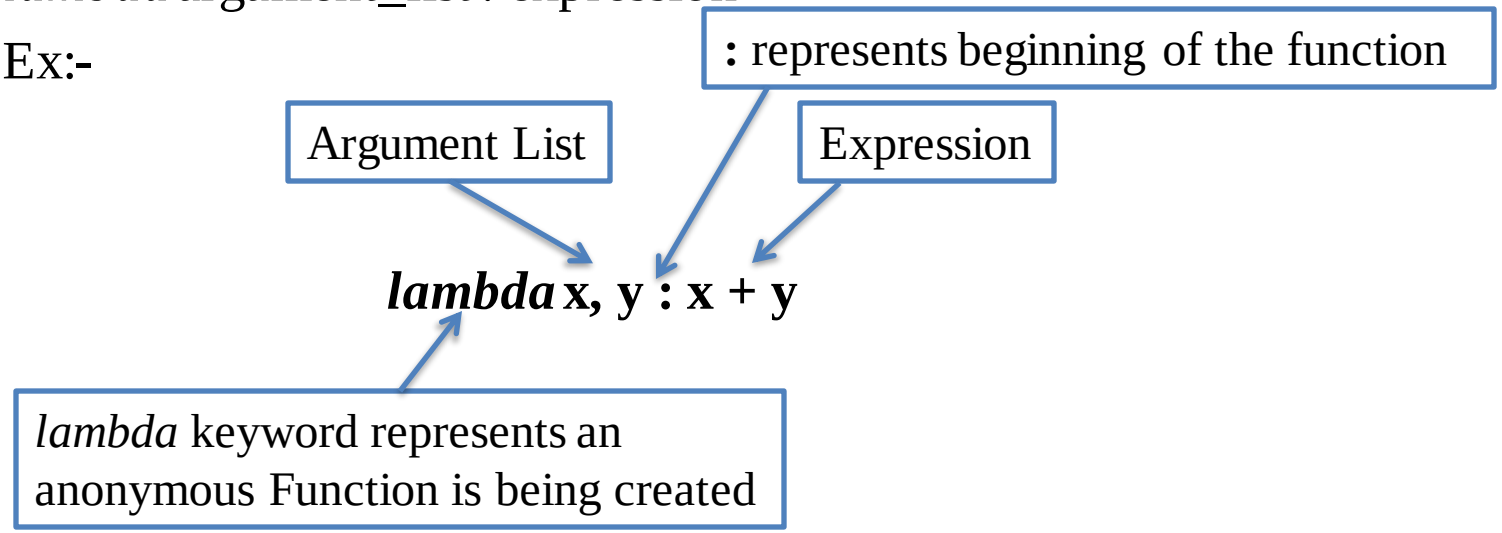
lambda x, y : x + y

Creating a Lambda Function

Syntax:-

lambda argument_list : expression

Ex:-



Calling Lambda Function

```
sum = lambda x : x + 1
sum(5)

add = lambda x, y : x + y
add(5, 2)
```

Anonymous Function or Lambdas

- Lambda Function doesn't have any Name

  Ex:- *lambda* x : print(x)

- Lambda function returns a function

  Ex:- show = *lambda* x : print(x)

- Lambda function can take zero or any number of argument but contains only one expression

  Ex:- *lambda* x, y : x + y

- In lambda Function there is no need to write return statement
- It can only contain expressions and can't include statements in its body
- You can use all the type of Actual Arguments

Function- Map()(CO2)

- Lambda functions are mainly used in combination with the functions **filter(), map() and reduce().**

- The lambda feature was added to Python due to the demand from Lisp programmers.

Function- Map()(CO2)

- The advantage of the lambda operator can be seen when it is used in combination with the map() function.
  map() is a function with two arguments:

  r = map(func, Seq)
- The first argument func is the name of a function and the second a sequence (e.g. a list) seq.
- map() applies the function func to all the elements of the sequence seq.
- It returns a new list with the elements changed by func.

Function- Map() Program(CO2)

```
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(map(lambda x: x*2, li))
print(final_list)
```

OUTPUT
25,49,484,9409,2916,3844,5929,529,5329,3721

Function- reduce()(CO2)

- The reduce() function in Python takes in a function and a list as an argument.
- The function is called with a lambda function and an iterable and a new reduced result is returned.
- This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the ***functools*** module.

Function- reduce() Program(CO2)

```
# reduce() with lambda()
# to get sum of a list

from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

OUTPUT
193

Function- filter() (CO2)

- The filter() function in Python takes in a function and a list as arguments.

- This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.

Function- filter() Program(CO2)

```
# filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

OUTPUT
5,7,97,77,23,73,61

Comprehension

- Python provides compact syntax for deriving one list from another. These expressions are called ***list comprehensions***.

- List comprehensions are one of the most powerful tools in Python. Python's list comprehension is an example of the language's support for functional programming concepts.

- The Python list comprehensions are a very easy way to apply a function or filter to a list of items. List comprehensions can be very useful if used correctly but very unreadable if you're not careful

Comprehension

Syntax

The general syntax of list comprehensions is −

```
[expr for element in iterable if condition]
```

Above is equivalent to −

```
for element in iterable:
    if condition:
        expr
```

List Comprehension vs For Loop

```
#USING FOR LOOP
evens = []
for i in range(10):
    if i % 2 == 0:
        evens. append(i)
print(evens)
```

Output
[0, 2, 4, 6, 8]

A better and faster way to write the above code is through list comprehension.

```
>>> [i for i in range(10) if i % 2 ==0]
```

Output
[0, 2, 4, 6, 8]

List comprehensions vs lambda function

List comprehensions are clearer than the map built-in function for simple cases. the map requires creating a lambda function for the computation, which is visually noisy.

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 >>> list(map(lambda x: x**2, lst))
```

Output

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

List comprehensions with conditional expression

Example:

when you only want to compute the squares of the numbers that are divisible by 2.

```
–>>> lst=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> even_squares = [x**2 for x in lst if x % 2 == 0]
>>> print(even_squares)
```

Output :
[4, 16, 36, 64, 100]

List comprehensions with conditional expression

```
# a list of even numbers between 1 and 100

>>> evens = [i for i in range(1,100) if not i % 2]
> >>print(evens)
```

Output:

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]

Nested IF with List Comprehension

```
>>> num_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
>>> print(num_list)
```

Output:
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

if...else With List Comprehension

```
>>> obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
>>> print(obj)
```

Output:
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']

Python for-loop to list comprehension?

- List comprehensions offer a concise way to create lists based on existing lists.
- When using list comprehensions, lists can be built by leveraging any iterable, including strings and tuples.
- List comprehensions consist of an iterable containing an expression followed by a for the clause. This can be followed by additional for or if clauses.
- Let's look at an ***example*** that creates a list based on a string:

```
>>> hello_letters = [letter for letter in 'hello']
>>> print(hello_letters)
```

Output:
['h', 'e', 'l', 'l', 'o']

Python for-loop to list comprehension? (cont.)

String hello is iterable and the letter is assigned a new value every time this loop iterates. This list comprehension is equivalent to:

```
>>> hello_letters = []
>>> for letter in 'hello':
>>>    hello_letters.append(letter)
```

Nested Loops in List Comprehension

We need to compute the transpose of a matrix that requires nested for loop. Let's see how it is done using normal for loop first.

Transpose of Matrix using Nested Loops

```
transposed = []
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
for i in range(len(matrix[0])):
        transposed_row = []
        for row in matrix:
                transposed_row.append(row[i])
        transposed.append(transposed_row)
print(transposed)
```

Output:
[[1, 4], [2, 5], [3, 6], [4, 8]]

Nested Loops in List Comprehension

- We can also perform nested iteration inside a list comprehension. In this section, we will find transpose of a matrix using nested loop inside list comprehension.'
- Transpose of a Matrix using List Comprehension

```
>>> matrix = [[1, 2], [3,4], [5,6], [7,8]]
>>> transpose = [[row[i] for row in matrix] for i in range(2)]
>>> print (transpose)
```

Output:
[[1, 3, 5, 7], [2, 4, 6, 8]]

Python Dictionary Comprehension

- We need the key: value pairs to create a dictionary. To get these key-value pairs using dictionary comprehension the general statement of dictionary comprehension is as follows:

  {key: value for ___ in iterable}

Let's see how to generate numbers as keys and their squares as values within the range of 10. Our result should look like

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}.

For this result the below code is as follows:-

Python Dictionary Comprehension

```
# creating the dictionary
>>> squares = {i: i ** 2 for i in range(10)}
>>> print(squares)
```

Output

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

Let's see how to create a dictionary from **[1, 2, 3, 4, 5]** and **[a, b, c, d, e].**

```
# keys
>>> key=['a','b','c','d']
# value=[3,5,6,7]
# creating a dict from the above lists
>>> dictionary = {key: value for (key, value) in zip(keys, values)}
>>> print(dictionary)
{'a': 3, 'b': 5, 'c': 6, 'd': 7}
```

IMMUTABILITY

- Every variable in python holds an instance of an object. There are two types of objects in python i.e. **Mutable** and **Immutable objects**. Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can't be changed afterwards. However, it's state can be changed if it is a mutable object.

- To summarize the difference, mutable objects can change their state or contents and immutable objects can't change their state or content.

- In python, the string data types are immutable. Which means a string value cannot be updated. We can verify this by trying to update a part of the string which will led us to an error.

IMMUTABILITY

Example:

```
# Can not reassign
 >>> t= “Mumbai"
>>> print type(t)
 >>> t[0] = “P”.
```

When we run the above program, we get the following output −

t[0] = "M" TypeError: 'str' object does not support item assignment

IMMUTABILITY

- We can further verify this by checking the memory location address of the position of the letters of the string.

```
>>> x = 'banana' for idx in range (0,5):
>>> print x[idx], "=", id(x[idx])
```

When we run the above program we get to see that N and N also point to the same location

CLOSURE

- A *closure* is a nested function which has access to a free variable from an enclosing function that has finished its execution. Three characteristics of a Python closure are:
  - it is a nested function
  - it has access to a free variable in outer scope
  - it is returned from the enclosing functio

- A *free variable* is a variable that is not bound in the local scope
- Python closures help avoiding the usage of global values and provide some form of data hiding.

CLOSURE

When we define a function inside of another, the inner function is said to be nested inside the outer one. Let's take an *example.*

```
def outerfunc(x):
        def innerfunc():
                print(x)
        innerfunc()
outerfunc(7)
```

OUTPUT
7

CLOSURE

Innerfunc() could read the variable 'x', which is nonlocal to it. And if it must modify 'x', we declare that it's nonlocal to innerfunc()

Let's *define a closure*:-

```
def outerfunc(x):
        def innerfunc():
                print(x)
                return innerfunc        #Return the object instead of calling the function

myfunc=outerfunc(7)
 myfunc()
```

OUTPUT
7

CLOSURE

- The point to note here is that instead of calling innerfunc() here, we returned it (the object).
- Once we've defined outerfunc(), we call it with the argument 7 and store it in variable myfunc()
- when we call myfunc next, how does it remember that 'x' is 7?
- ***A Python3 closure is when some data gets attached to the code.***
- So, this value is remembered even when the variable goes out of scope, or the function is removed from the namespace.

CLOSURE

- Python closure is used when a nested function references a value in its enclosing scope.

- These *three* conditions must be met:

  1. We must have a nested function.
  2. This nested function must refer to a variable nonlocal to it(a variable in the scope enclosing it).
  3. The enclosing scope must return this function.

Benefits of Python Closure

While it seems like a very simple concept, a closure in python3 helps us in the following ways:

- With Python closure, we don't need to use global values. This is because they let us refer to nonlocal variables. A closure then provides some form of data hiding.
- When we have only a few Python methods (usually, only one), we may use a Python3 closure instead of implementing a class for that. This makes it easier on the programmer.
- A closure, lets us implement a Python decorator.
- A closure lets us invoke Python function outside its scope.

Decorators(CO3)

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.
- Decorators are usually called before the definition of a function.

```
def uppercase_decorator(function):
    def wrapper():
            func = function()
            make_uppercase=func.upper()
            return make_uppercase
return wrapper
```

Note: Our decorator function takes a function as an argument, therefore, define a function and pass it to our decorator.

Call of Decorators(CO3)

```
def say_hi():
    return 'hello there'
decorate = uppercase_decorator(say_hi)


decorate()
```

Output:
'HELLO THERE'

Call of Decorators(CO3)

```
# call  of decoratots
@uppercase_decorator
def say_hi():
return 'hello there‘


say_hi()
```

Output:
‘HELLO THERE’

Applying Multiple Decorators to a Single Function(CO3)

```
def split_string(function):
    def wrapper():
            func = function()
            splitted_string = func.split()
            return splitted_string
return wrapper
@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'
say_hi()
```

Output:
['HELLO ','THERE']

Accepting Arguments in Decorator Functions(CO3)

```
def decorator_with_arguments(function):
    def wrapper_accepting_arguments (arg1, arg2):
        print("My arguments are: {0}, {1}".format(arg1,arg2))
        function(arg1, arg2)
        return wrapper_accepting_arguments
@decorator_with_arguments
def cities(city_one, city_two):
        print("Cities I love are {0} and {1}".format(city_one,
    city_two))
# call of Decorators
 cities("Nairobi", "Accra")
```

Output:

My arguments are: Nairobi, Accra Cities I love are Nairobi and Accra

Generator Functions

- A Python generator is a kind of an iterable, like a Python list or a python tuple. It generates for us a sequence of values that we can iterate on.
- You can use it to iterate on a for-loop in python, but you can't index it. Let's take a look at how to create one with python generator example.

- To create a python generator, we use the yield statement, inside a function, instead of the return statement.

```
>>> def counter():
        i=1
        while(i<=10):
            yield i
            i+=1
```

Generator Functions

- With this, we define a Python generator called counter() and assign 1 to the local variable i. As long as i is less than or equal to 5, the loop will execute.
- Inside the loop, we yield the value of i, and then increment it.

- Then, we iterate on this generator using the for-loop.

```
>>> for i in counter():
        print(i)
```

Output:

```
1
2
3
4
5
```

Working of Python Generator

- To understand how this code works, we'll start with the for-loop. For each item in the Python generator (each item that it yields), it prints it, here.
- We begin with i=1. So, the first item that it yields is 1. The for-loop prints this because of our print statement. Then, we increment I to 2.
- And the process follows until i is incremented to 11. Then, the while loop's condition becomes False.
- However, if you forget the statement to increment I, it results in an infinite generator. This is because a Python generator needs to hold only one value at a time.
- So, there are no memory restrictions.

Working of Python Generator

```
def even(x):
        while x%2==0:
                yield 'Even'
for i in even(2):
print(i)
```

since 2 is even, 2%2 is always 0. Hence, the condition for while is always true.

Output:

```
Even
Even
Even
Even
Even
EvenEvenTraceback (most recent call last):File "<pyshell#24>", line 2, in <module>print(i)
KeyboardInterrupt
```

Working of Python Generator

Generator may contain more than one Python yield statement. This is comparable to how a Python generator function may contain more than one return statement.

```
def my_gen(x):
        while(x>0):
                if x%2==0:
                        yield 'Even'
                else:
                        yield 'Odd'
        x-=1
for i in my_gen(7):
print(i)
```

Output: Odd
Even
Odd
Even
Even
Odd

Yielding into a Python List

- This one's a no-brainer. If you apply the list() function to the call to the Python generator, it will return a list of the yielded values, in the order in which they are yielded.
- Here, we take an example that creates a list of squares of numbers, on the condition that the squares are even.

```
def even_squares(x):
        for i in range(x):
                if i**2%2==0:
                        yield i**2

print(list(even_squares(10)))
```

Output:
[0, 4, 16, 36, 64]

Python List vs Generator in Python

- A list holds a number of values at once. But a Python generator holds only one value at a time, the value to yield.

- This is why it needs much less space compared to a list. With a generator, we also don't need to wait until all the values are rendered.

Introduction to Sub routine...

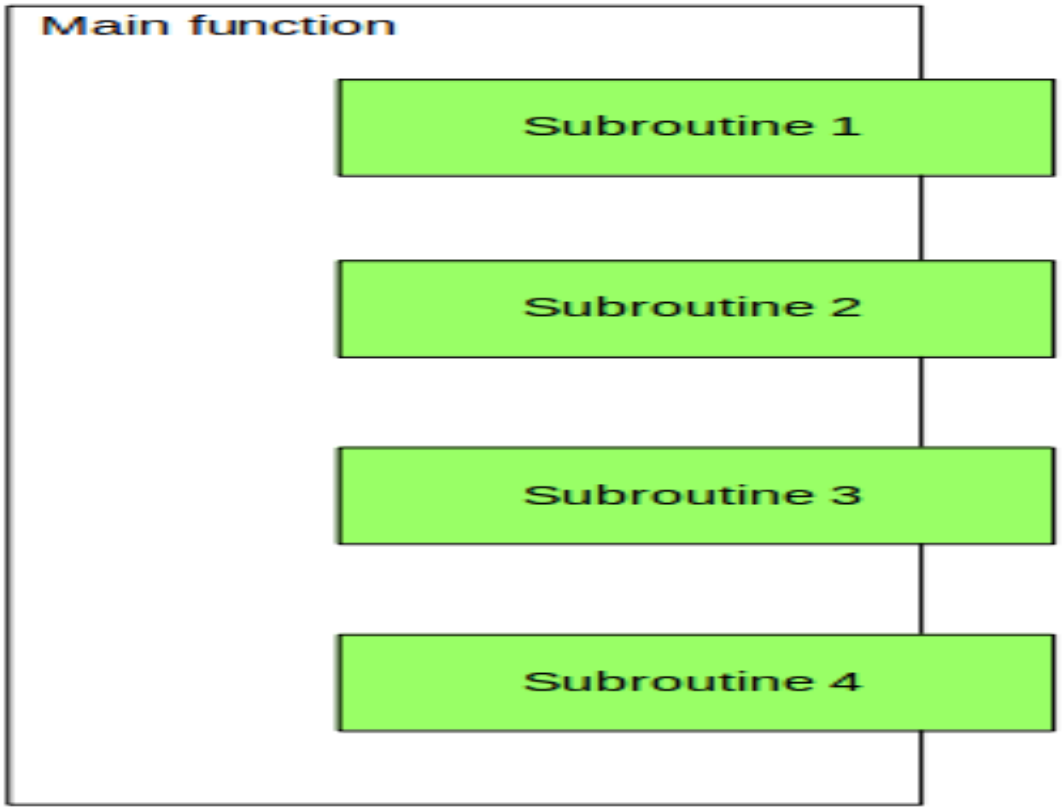
Function which is also known
as **subroutine**, **procedure**, **subprocess** etc."

A function is a sequence of instructions packed as a unit to perform a certain task.

When the logic of a complex function is divided into several self-contained steps that are themselves functions, then these functions are called helper functions or **subroutines**.

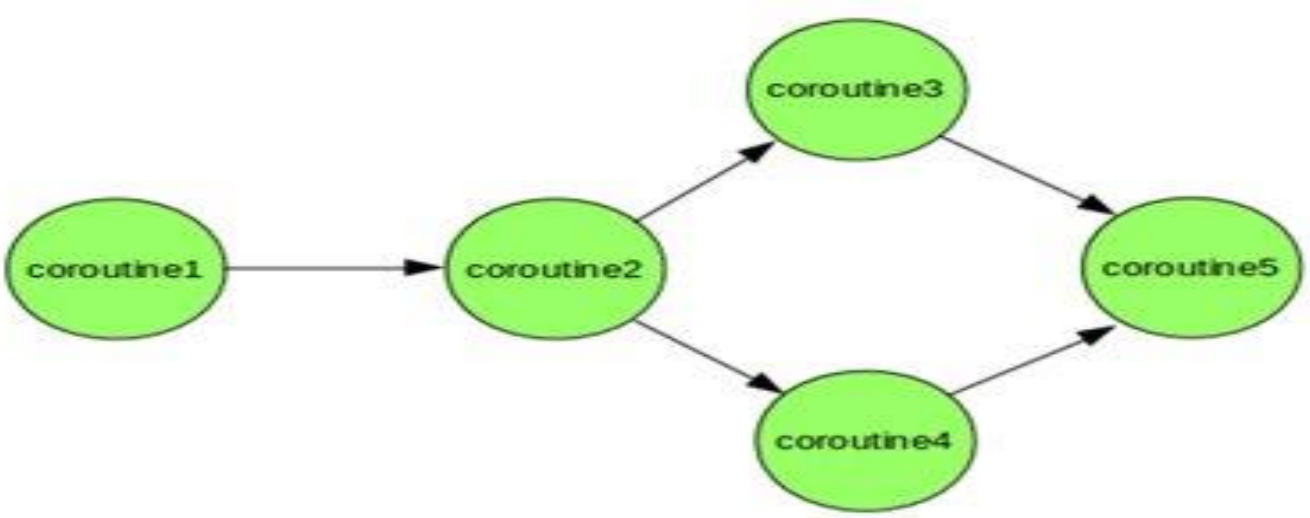
Introduction to Sub routine...

Subroutines in Python are called by **main function** which is responsible for coordination the use of these subroutines. Subroutines have single entry point.



Co routines ...

"Coroutines are generalization of subroutines." They are used for cooperative multitasking where a process voluntarily **yield** (give away) control periodically or when idle in order to enable multiple applications to be run simultaneously.



Python program for demonstrating # coroutine execution

```
def print name(prefix):
    print("Searching prefix:{}".format(prefix))
    while True:
        name = (yield)
        if prefix in name:
            print(name)

# calling coroutine, nothing will happen
corou = print_name("Dear")

# This will start execution of coroutine and
# Prints first line "Searchig prefix..."
# and advance execution to the first yield expression
```

Python program for demonstrating # coroutine execution ...

```
corou.__next__()

# sending inputs
corou.send(“XYZ")
corou.send("Dear XYZ")
```

Output:
Searching prefix:Dear Dear XYZ

Closing a Coroutine

```
def print_name(prefix):
    print("Searching prefix:{}".format(prefix))
    try :
        while True:
            name = (yield)
            if prefix in name:
                print(name)
    except GeneratorExit:
        print("Closing coroutine!!")
```

```
corou = print_name("Dear")
corou.__next__()
corou.send("Atul")
corou.send("Dear Atul")
corou.close()
```

OUTPUT
Searching prefix:Dear Dear XYZ Closing coroutine!!

Iterator

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().

Iterator vs Iterable

- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.
- All these objects have a iter() method which is used to get an iterator:

Example:

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

OUTPUT

Apple
Banana
cherry

Iterator vs Iterable

"Even strings are iterable objects, and can return an iterator"

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example:

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
    print(x)
```

Output:

```
apple
banana
cherry
```

Create an Iterator

- To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.
- All classes have a function called __init__(), which allows to do some initializing when the object is being created.
- The __iter__() method acts similar, and can do operations (initializing etc.), but must always return the iterator object itself.
- The __next__() method also allows to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will

increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

def __next__(self):
    x = self.a
    self.a += 1
    return x.
```

Example

```
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:-

1
2
3
4
5

Stop Iteration

- The example above would continue forever if you had enough next() statements, or if it was used in a for loop.
- To prevent the iteration to go on forever, we can use the StopIteration statement.
- In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

StopIteration

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
  print(x)
```

Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Declarative Programming Language

Declarative Programming Languages focus on :

- ***describing what should be computed*** - and avoid mentioning how that computation should be performed. This means avoiding expressions of control flow: loops and conditional statements are removed and replaced with higher level constructs that describe the logic of what needs to be computed.
- The usual ***example*** of a declarative programming language is ***SQL***. It lets you define what data you want computed - and translates that efficiently onto the database schema.
- ***Python isn't a pure Declarative Language*** - but the same flexibility that contributes to its sluggish speed can be be leveraged to create Domain Specific API's that use the same principles.