final_notebook

May 24, 2022

0.1 What is Function?

If a group of lines is required frequently, it is not advised to write them separately each time. These statements must be defined as a single unit, which we may call any number of times based on our requirements without having to rewrite. This is called function.

Functions are also called as methods, procedures, and subroutines

0.2 Advantage

`Code reuse.`

1 Types of functions

`1. Built-in Features`
`2. Customized Functions`

2 Built in Functions

Built-in functions or pre-defined functions are functions that are automatically included with Python programme.

For instance, print(), max(), pow(), input() etc.

Python 3, there are 68 built-in functions

3 User defined functions

`User defined functions are functions that are written explicitly according to requirements by p`

[51]:
```
#Syntax to create user defined functions
"""
def function_name(parameters) :
    ---- -----
    ---- -----
    return value(s) # Optional
"""
```

[51]:
```
'    \ndef function_name(parameters) : \n    ---- ----- \n    ---- -----\n
return value(s) # Optional\n'
```

[4]:
```
"""Write a function to print "Humanity Above All""""
# Function definition
def myFunction():
    print("Humanity Above All")

#myFunction()

# Function call
for _ in range(5):
    myFunction()
```

```
Humanity Above All
Humanity Above All
Humanity Above All
Humanity Above All
Humanity Above All
```

4 Parameters

Parameters are the function's inputs. If a function has parameters, we must specify values when invoking the function; otherwise, we will receive an error.

[5]:
```
""" Function Example 1"""
def myFunction(name): # "name" is the formal parameter
    print("Hi!",name," How are you?")
    # Default return value is None

myFunction("Kapil") # Kapil is the actual parameters, call a function.
myFunction("Rani")
```

```
Hi! Kapil  How are you?
Hi! Rani  How are you?
```

[8]:
```
""" Function Example 2"""
def myFunction(x): # 'x' is the formal parameter
    print("Square of the",x,"is: ", x**2)
    # Default return value is None

myFunction(5) # 5 is the actual parameters
myFunction(19)
```

```
Square of the 5 is:  25
Square of the 19 is:  361
```

[14]:
```
""" Function Example 3: define and passing two parameters"""
def addSquare(x,y): # x and y are the formal parameter
    return x**2 + y**2

# Case 1
print("Case1: The result is: ",addSquare(5,5)) # Function call

#Case 2
z = addSquare(2,9)
print("Case2: The result is: ",z) # Function call
```

```
Case1: The result is:  50
Case2: The result is:  85
```

[9]:
```
""" Function Example 4 (returning a variable)"""
def addSquare(x,y):
    w = x**2 + y**2
    return w # Variable returning

print("Case 1 result: ",addSquare(2,3)) # Case 1

z = addSquare(6,3) #Case 2
print("Case 2 result: ",z)
```

```
Case 1 result:  13
Case 2 result:  45
```

[11]:
```
"""Example 5: More than one return values"""
def add_sub_mul_Function(x, y):
    mySum = x + y
    mySub = x - y
    myMul  = x*y
    return  mySum, mySub, myMul # Return more than one variables

aSum, bSub, cMul = add_sub_mul_Function(9, 3)
# Receiving more than one variables

print("The Sum is :", aSum)
print("The Subtraction is :", bSub)
print("The multiplication is: ",cMul)
```

```
The Sum is : 12
The Subtraction is : 6
The multiplication is:  27
```

5 Types of actual parameters

1. positional
2. keyword

3. default
4. variable length

6 1. Positional parameters

Correct positional order must be maintained

[14]:
```
""" 1. Positional parameters"""
def positionalFunction(x,y): # Normal argument: matched by position
    print(x," to the power", y, "is: ",x**y)

positionalFunction(2, 3)
positionalFunction(3, 2)
```

```
2  to the power 3 is:  8
3  to the power 2 is:  9
```

7 Notes on Positional parameters

1. The number of parameters and position of parameters must be the same.
2. If the orders of the parameters are changed, the outcome may be altered.
3. We will get an error if we vary the number of parameters.

[17]:
```
"""Positional parameters"""
def positionalFunction(x,y):
    print(x," to the power", y, "is: ",x**y)

positionalFunction(2, 3)
#positionalFunction(2, 3, 5)
```

```
2  to the power 3 is:  8
```

8 Keyword Parameters

We can pass parameter values by keyword i.e., by parameter's name. The sequence of the paramete

[20]:
```
"""keyword Parameters"""
def keywordFunction(x,y):
    print(x," to the power", y, "is: ",x**y)

keywordFunction(x = 2, y = 3) #Keyword parameters: matched by name
#keywordFunction (y = 3, x = 2)
#keywordFunction (y = 3, x = 2, z = 1)
```

```
2  to the power 3 is:  8
```

Both positional and keyword parameters can be used at the same time. However, we must take positional parameters first, followed by keyword parameters, else we will get a syntax error.

[23]:
```
def pkFunction(x,y):
    print(x," to the power", y, "is: ",x**y)

pkFunction(2, 3)
pkFunction(2, y = 3)
#positionalFunction(x = 2, 3)
```

```
2  to the power 3 is:  8
2  to the power 3 is:  8
```

9 Default Parameters

For our positional parameters, we can sometimes assign default values.

[29]:
```
"""Default Parameters"""

def defaultPFunction(x = 3, y = 2):
    print(x," to the power", y, "is: ",x**y)

defaultPFunction()              # 3^2
defaultPFunction(2)             # 2^2
defaultPFunction(2, 3)          # 2^3
defaultPFunction(2, y = 3)      # 2^3
#defaultPFunction(x = 2, y = 3)  # 2^3
#defaultPFunction(y = 3, 5)
```

```
3  to the power 2 is:  9
2  to the power 2 is:  4
2  to the power 3 is:  8
2  to the power 3 is:  8
```

10 Variable length parameters

1. In variable length parameters, we may provide variable number of parameters.
2. The * symbol can be used to define a variable length parameter.

[36]:
```
"""Example of variable length parameters"""
def variablePF(*N):
    mulv = 1
    print("The received value: ",N)
    print("Number of received parameters: ", len(N))
    for n in N:
        mulv *= n

    print("The multiplication value is: ",mulv)

#variablePF() # No parameter
#variablePF(10)
```

```
variablePF(4,3)
#variablePF(range(10))
```

```
The received value:  (4, 3)
Number of received parameters:  2
The multiplication value is:  12
```

11 Mixing variable length parameter with positional parameter.

[42]:
```
""" Example 1: Mixing variable length param. with positional param"""
def variablePF(data, *N):
    mulv = data
    print("data is: ",data,", N is: ", N)

    for n in N:
        mulv *= n

    print("The multiplication value is: ",mulv)

#variablePF (2)
#variablePF (3, 4)
variablePF (3, 4, 5, 8, 9, 10)
#variablePF (2, 4, data = 5)
#variablePF (data = 2, 4, 5)
```

```
data is:  3 , N is:  (4, 5, 8, 9, 10)
The multiplication value is:  43200
```

[47]:
```
""" Example 2: Mixing variable length param. with default values param"""
def variablePF(*N, data = 1):
    mulv = data
    print("data is: ",data,", N is: ", N)
    for n in N:
        mulv *= n

    print("The multiplication value is: ",mulv)

#variablePF (2) # Default value of data is 1, N is:  (2,)
#variablePF (3, 4) # Default value of data is 1, N is:  (3,4)
#variablePF (2, 4, 5) # Default value of data is 1, N is:  (2, 4, 5)
variablePF (2, 4, data = 5) # Here the value of data is 5, N is:  (2, 4)
#variablePF (data = 5, 4, 5, 7)
```

```
data is:  5 , N is:  (2, 4)
The multiplication value is:  40
```

12 key word variable length parameters

Use **. Internally these keyword are stored as dictionary.

[52]:
```
""" key word variable length parameters"""
def vlkParameters(**kwargs):
    for key, value in kwargs.items():
        print(key, ":", value)

#vlkParameters(PI = 3.14)
#vlkParameters(Name = "Kapil", Address = "Chandigarh")
vlkParameters(a=1,b=9,c=7)
```

```
a : 1
b : 9
c : 7
```

[54]:
```
# use positional, *args and **kwargs togethor
def examplePAK(a, b, *args, **kwargs):
    print("The values of postional parameters are: ",a,",",b)
    sum =0
    for i in args:
        sum +=i
    print("The sum of variable length parameters is: ",sum)

    #print(kwargs)
    for j in kwargs.items():
        print("The key word values are: ",j)

#examplePAK(2,3, 10, 11, 12, x = "Great", y = "Goal") # Function call
examplePAK(7,9, 10, 11, 12, 15, 17, x = "God", y = "is", z = "Great") #␣
 ↪Function call
```

```
The values of postional parameters are:  7 , 9
The sum of variable length parameters is:  65
The key word values are:  ('x', 'God')
The key word values are:  ('y', 'is')
The key word values are:  ('z', 'Great')
```

13 Difference between Function, Module, Library

1. A set of lines saved with name is called a function.
2. A module consists of set of functions.
3. A library consists of set of modules.

14 Types of variables

1. Global Variables

2. Local Variables

15 Global Variables

Global variables are variables that are defined outside of a function. All functions in that module can access these variables

[57]:
```
"""Example of global variable"""
PI = 3.14
def globalVFunction1():
    print("The value of PI in F1",PI)

def globalVFunction2():
    print("The value of PI in F2",PI)

globalVFunction1()
globalVFunction2()
```

```
The value of PI in F1 3.14
The value of PI in F2 3.14
```

Notes on local variables 1. Local variables are variables that are declared within a function. 2. These are only available in the function where they were declared. 3. We can't access it from outside the function.

[58]:
```
"""Example of local and global variables"""
A = 10
def VFunction1():
    A = 20
    print("F1: Print local value: ",A)

def VFunction2():
    print("F2: Print global value",A)

VFunction1()
VFunction2()
```

```
F1: Print local value:  20
F2: Print global value 10
```

[61]:
```
""" global key word"""
A = 10
def VFunction1():
    global A
    A = 20
    print("F1: Print local value: ",A)

def VFunction2():
    print("F2: Print global value: ",A)
```

```
VFunction1()
VFunction2()
```

```
F1: Print local value:  20
F2: Print global value:  20
```

[63]:
```
""" Accessing global variable and suppressing local variables"""
A = 50
def VFunction1():
    A = 20 # Local variables
    print("F1: Print local value: ",A) # Local value
    print("F1: This is global value: ",globals()['A']) # suppressing local␣
 ↪variables

def VFunction2():
    print("F2: Print global value: ",A)

VFunction1()
VFunction2()
```

```
F1: Print local value:  20
F1: This is global value:  50
F2: Print global value:  50
```

[65]:
```
var1 = 10

def localF():
    var1 = 50
    print("Print the local variable: ",var1)

print("Print the global variable: ",var1)
localF()
print("Print the global variable: ",var1)
```

```
Print the global variable:  10
Print the local variable:  50
Print the global variable:  10
```

[67]:
```
var = 11
def myF():
    global var # Declare global
    var += 1
    print("Print the variable: ",var)

print("Print the variable: ",var)
var +=1
myF()
```

```
Print the variable:  11
Print the variable:  13
```

[68]:
```
var = 10 # outer global variable global
def myF1():
    var = 13 # inner global variable

    def myF2():
        print("Inner global is printed: ",var)
        # Print variables defined inside the parent function
    myF2()

myF1()
print("Inner global is printed: ",var)
```

```
Inner global is printed:  13
Inner global is printed:  10
```

[70]:
```
# Examples: Is forward reference okay?
def myf1():
    val = 30
    myf2(val) # Forward reference okay

def myf2(val):
    print("The value is: ",val)

myf1()
```

```
The value is:  30
```

16 The nonlocal keyword

It's used to work with variables inside nested functions that shouldn't be part of the inner function.

[83]:
```
data = 5
def myfunc1():
    #global data
    data = 10
    print("Data1: ",data)
    def myfunc2():
        nonlocal data
        #global data
        data = 20
        print("Data2: ",data)
    myfunc2()

    print("Data3: ",data)
    return data
```

```
print("Data4: ",myfunc1())
print("Data5:",data)
```

```
Data1:  10
Data2:  20
Data3:  20
Data4:  20
Data5: 5
```

[84]:
```
data = 5
def myfunc1():
    #global data
    #nonlocal data
    data = 10
    print("Data1: ",data)
    return data
print("Data2: ",myfunc1())
print("Data3:",data)
```

```
Data1:  10
Data2:  10
Data3: 5
```

[1]:
```
def myfunc1():
    data = 10
    print("Data1: ",data)
    def myfunc2():
        #nonlocal data
        data = 20
        print("Data2: ",data)
        def myfunc3():
            nonlocal data
            data = 30
            print("Data3: ",data)
        myfunc3()
        print("Data5: ",data)
    myfunc2()
    print("Data6: ",data)
    return data

print("Data7: ",myfunc1())
```

```
Data1:  10
Data2:  20
Data3:  30
Data5:  30
Data6:  10
Data7:  10
```

17 Recursive function

Function call itself for sepecified time.

[88]:
```
""" Recursive function"""
def factorial(n):
    if n == 0:
        return 1
    else:
        return(n*factorial(n-1))

print("Factorial of 4 is :",factorial(4))
print("Factorial of 5 is :",factorial(5))
```

```
Factorial of 4 is : 24
Factorial of 5 is : 120
```

18 Anonymous functions or lambda functions

Sometimes we declare a function without giving it a name; these functions are known as anonymous functions or lambda functions. * lambda Function: defined by “lambda” keyword * lambda argument_list : expression * Example: lambda x:x*x

[86]:
```
"""Example of lambda function (with single parameter)"""
myF=lambda n:n*n

#print("The Square of 4 is :", myF (4))
print("The Square of 5 is :", myF (5))
#print("The Square of XX is :", myF (4,5))
```

```
The Square of 5 is : 25
```

[94]:
```
"""Example of lambda function (with more than one parameter)"""
myF=lambda a,b: a**b

print(" 2 to the power 4:", myF (2,4))
print(" 3 to the power 2:", myF (3,2))
#print(" X to the power X:", myF (3))
```

```
 2 to the power 4: 16
 3 to the power 2: 9
```

[88]:
```
def myfunc():
    x = 4
    result = (lambda n: x ** n) # x is remembered
    return result

# object of myfunc is defined, and objecr value is used to access the lamda
value = myfunc()
```

```
print("The output is: ",value(2)) # n = 2, input of lambda
```

```
The output is:  16
```

[94]:
```
def myFunction():
    data = []
    for i in range(10): # Use defaults instead
        data.append(lambda x, i=i: i ** x) # Remember i
    return data
value = myFunction()
#print(value)
value[2](10) # i = 3, x = 2
```

[94]:
```
1024
```

19 Notes on lambda function

1. We may build very brief code with lambda functions.
2. It improves the readability of the programme.
3. No explicit returns statement is required.
4. Using lambda function we can pass function as argument to another function.

[95]:
```
# More examples of lambda in list
myList = [lambda x: x + 1, lambda x: x - 1, lambda x: x ** 0.5, lambda x: x**2]
for f in myList:
    print("The values are: ",f(5))

print("The last result is: ",myList[2](5))
# 0: inc, 1: Dec, 2: SRoot, 3: Square
```

```
The values are:  6
The values are:  4
The values are:  2.23606797749979
The values are:  25
The last result is:  2.23606797749979
```

[99]:
```
# More examples of lambda in dictionary
myList = {'Inc':(lambda x: x + 1),
          'Dec':(lambda x: x - 1),
          'SRoot': (lambda x: x ** 0.5)}

for f in myList:
    print("The value of", f, "is: ",myList[f](4))

#print("The last result is: ",myList[0](3))
# 0: inc, 1: Dec, 2: SRoot
```

```
The value of Inc is:  5
The value of Dec is:  3
```

```
The value of SRoot is:  2.0
```

20 Lambda with filter

```
The filter() function is used to filter values
(based on some condition)from the given sequence.
Syntax: filter(function,sequence)
```

[100]:
```
"""Lambda with filter"""
data = range(10)
LFF1 = list(filter(lambda x:x%2==0,data)) # filter(function,sequence)
LFF2 = list(filter(lambda x:x>=5,data))

print("Print even numbers: ",LFF1)
print("List value square: ",LFF2)
```

```
Print even numbers:  [0, 2, 4, 6, 8]
List value square:  [5, 6, 7, 8, 9]
```

21 Lambda with map() function

For every data present in the given sequence, apply some functionality and generate new dataset with the required modification.

syntax: map(function,sequence)

[98]:
```
"""Lambda with map() function (Example 1)"""
data=range(5)
X=list(map(lambda x:x**2,data))

print("Square the value: ",X)
```

```
Square the value:  [0, 1, 4, 9, 16]
```

[105]:
```
"""Lamda with map() function (Example 2)"""
data1=range(5)
data2=range(6,10)
X=list(map(lambda x,y:x**y,data1, data2))

print("data1 square data2: ",X)
```

```
data1 square data2:  [0, 1, 256, 19683]
```

22 Lamda with reduce() function

It reduces sequence of elements into a single element by applying some function.

syntax: reduce(function,sequence)

[107]:
```
from functools import *
data=[2, 1, 4, 3]
result=reduce(lambda x,y:x+y,data)
print("The reduce resulty is: ",result)
```

```
The reduce resulty is:  10
```

23 Function Aliasing

Renaming a function

[112]:
```
def addMy(x,y):
    print("Addition is:",x+y)

sumMy=addMy

print("The id is: ",id(sumMy(2,6)))
print("The id is: ",id(addMy(4,5)))

sumMy(2,6)
#del addMy
#addMy(10, 12)
#sumMy(10,6)
```

```
Addition is: 8
The id is:  140726057277568
Addition is: 9
The id is:  140726057277568
Addition is: 8
```

If we delete a function still we can access that function using alias name. * Syntax to delete: del sumMy

24 Nested Functions

declare a function inside another function

[113]:
```
"""Nested function example"""

def outerF():
    print("outer function initiated _1")

    def innerF():
        print("Inside the inner function _3")

    print("outer function called inner function _2")
    innerF() # Calling inside the outer Function
```

```
#inner() # Calling outside the outer Function
outerF()
```

```
outer function initiated _1
outer function called inner function _2
Inside the inner function _3
```

25 Functions are first-class objects

Everything in Python is viewed as an object. In Python, functions are first class objects since they can refer to, pass to variables, and return from other functions.

The function can also be called, supplied as a variable, and returned from other functions.

The functions can be declared inside another function and supplied to it as an argument.

26 Passing function as argument to another function

[114]:
```
"""Example 1: Passing function as argument to another function"""
data=[1,2,3, 5, 6]
def mySquare(x):
    return x**2

result=list(map(mySquare,data))
print("Passing a function:",result)
```

```
Passing a function: [1, 4, 9, 25, 36]
```

[115]:
```
#Example 2: Functions can return another function
def OuterAdder(x):
    print("The x value is:",x)
    def InnerAdder(y):          # Inner function
        print("The y value is:",y)
        return x+y

    return InnerAdder # Returning InnerAdder as argument

addingValues = OuterAdder(5) # The x value is 5, # Assign function object
#print("\nPrinting addingValues:\n",addingValues)
finalResult = addingValues(10) # The x value is 5, the y value is 10
print("The final result:",finalResult)
finalResult = addingValues(13) # The x value is 5, the y value is 10
print("The final result:",finalResult)
```

```
The x value is: 5
The y value is: 10
The final result: 15
The y value is: 13
The final result: 18
```

[123]:
```
#Example 3: Function returning another function
def outerF(num):
    def even():
        print("The", num, "is an even number")

    def odd():
        print("The", num, "is an odd number")

    if num%2 == 0:
        return even
    else:
        return odd
test = outerF(10) # Assign function object
test() # Call the function

test = outerF(13)
test()
```

```
The 10 is an even number
The 13 is an odd number
```

[124]:
```
#Example 4: Function returning another function
def outer(x):
    if x%2 == 0:
        def inner1():
            print(x,"is an even number")
        return inner1
    else:
        def inner2():
            print(x,"is an odd number")
        return inner2
    return outer

x = outer(10)
x()
x = outer(13)
x()
outer(11)()
```

```
10 is an even number
13 is an odd number
11 is an odd number
```

27 Closure or factory function

Depending on whom you ask, this sort of behavior is also sometimes called a closure or factory

[73]:
```
def outer(x):
    def inner(a):
         return a ** x # retains x from enclosing scope
    return inner

f1 = outer(2) # x = 2, object of the outer function is created
result1 = f1(3) # a = 3, Accessing the inner function, since outer holds inner
print("Result 1: ",result1) # 3^2 = 9

f2 = outer(4) # x = 4, In this case x = 4
result2 = f2(3) # a = 3,
print("Result 2: ",result2) # 3^4 = 81
print("Each call to make Inner creates a new instance (here  value of a) of␣
 ↪this function, but each instance has a link to a different binding of␣
 ↪variable")
```

```
Result 1:  9
Result 2:  81
Each call to make Inner creates a new instance (here  value of a) of this
function, but each instance has a link to a different binding of variable
```

28 Decorator function

- A decorator takes a function as an argument and extends its functionality, returning a modified function with the enhanced functionality.
- The fundamental goal of decorator functions is to allow us to increase the functionality of existing functions without having to change them. *It allows programmers to change a function behaves.* In Decorators, functions are passed as an parameter into another function and then called inside the wrapper function. *It is also known as "meta programming". Here one portion of a programme tries to change another component of the programme at compile time.

29 Higher Order Function

```
A function that accepts other function as an argument is called
higher order function.
```

[45]:
```
"""Example of higher order function"""
def Inc(x):
    return x+1
def Dec(x):
    return x-1

def operation(func, x): # func is a function, passing as parameter
    cal = func(x)
    return cal
```

```
print("The incremented operator: ",operation(Inc,10))
print("The decremented operator: ",operation(Dec,10))
```

```
The incremented operator:  11
The decremented operator:  9
```

[116]:
```
def divide(x,y):
    print("The result is: ",x/y)

def outerDiv(func):
    def inner(x,y):
        if y != 0:
            return func(x,y)
    return inner

divideResult1 = outerDiv(divide)
divideResult1(2,4)
```

```
The result is:  0.5
The result is:  12
```

[119]:
```
def outer_div(func):
    def inner(x,y):
        if y != 0:
            return func(x,y)
        else:
            print("y should not be 0")
    return inner

# decorators wrap a function, modifying its behavior.
@outer_div
def divide(x,y):
     print("The result is: ",x/y)

divide(2,4)
divide(6,2)
divide(5,0)

# Can we wrap another function??
```

```
The result is:  0.5
The result is:  3.0
y should not be 0
```

[124]:
```
# More than one decorators
def display(func):
    def inner(x,y):
        print("\nThe value of x is",x,"& the value of y is",y)
        return func(x,y)
```

```
    return inner

def outer_div(func):
    def inner(x,y):
        if y != 0:
            return func(x,y)
        else:
            print("y should not be zero!")
    return inner

@outer_div
@display
def divide(x,y):
     print("The result is: ",x/y)

divide(2,4)
divide(6,2)
divide(5,0)
```

```
The value of x is 2 & the value of y is 4
The result is:  0.5

The value of x is 6 & the value of y is 2
The result is:  3.0
y should not be zero!
```

[2]:
```
# More than one decorators
def mul(func):
    def inner(x,y):
        print("\nThe value of x is",x,"& the value of y is",y)
        print("Multiplication is: ", x*y)
        return func(x,y)
    return inner

def div(func):
    def inner(x,y):
        if y != 0:
            print("The division is: ",x/y)
            return func(x,y)
        else:
            print("y should not be zero!")
    return inner

@mul
@div
def evaluate(x,y):
```

```
    print("Evaluation is done!")

evaluate(2,4)
evaluate(6,2)
evaluate(5,0)
```

```
The value of x is 2 & the value of y is 4
Multiplication is:  8
The division is:  0.5
Evaluation is done!

The value of x is 6 & the value of y is 2
Multiplication is:  12
The division is:  3.0
Evaluation is done!

The value of x is 5 & the value of y is 0
Multiplication is:  0
y should not be zero!
```

30 Generator Function

It is similar to the normal function defined by the def keyword and uses a yield keyword instead of return.

[45]:
```
#Example 1
def myYield():
    str1 = "First time"
    yield str1

    str2 = "Second time"
    yield str2

    str3 = "Third time"
    yield str3

obj = myYield()
print(next(obj))
print(next(obj))
#print(next(obj))
```

```
First time
Second time
```

[48]:
```
#Example 2
#n=1
def myGen():
```

```
    n = 1
    n+=2
    yield n

    n+=2
    yield n

    n+=2
    yield n

output=myGen()
print(next(output))
print(next(output))
print(next(output))
```

```
3
5
7
```

31 Loop with Generator Function

[54]:
```
# Example 3
def myGen(x):
    for i in range(x):
        yield i

mySeq = myGen(3) # x = 3
print(next(mySeq))
print(next(mySeq))
print(next(mySeq))
#print(next(mySeq))
```

```
0
1
2
```

32 Handling the StopIteration error

[55]:
```
# Example 4, Handling the StopIteration error using try and except
def myGen(x):
    for i in range(1,x):
        yield i

mySeq = myGen(6) # x = 6

while True:
```

```
    try:
        print ("Received on next(): ", next(mySeq))
    except StopIteration:
        break
```

```
Received on next():  1
Received on next():  2
Received on next():  3
Received on next():  4
Received on next():  5
```

[5]:
```
# Example 5, Handling the StopIteration error without try and except
def myFun1(x):
    for i in range(0,5):
        yield x**i


for n in myFun1(2):  #  x = 2, 2^i
    print(n)
```

```
1
2
4
8
16
```

33 Difference between Generator function and Normal function

1. Normal function contains only one return statement where as generator function can contain one or more yield statements.
2. Local variable and their states are remembered between successive calls.

[60]:
```
# Example 6: fibonacci using function generator
def fibonacciGenerator():
    n1=0; n2=1
    while True:
        yield n1
        n1, n2 = n2, n1 + n2

fibSeq= fibonacciGenerator()
print(next(fibSeq))
print(next(fibSeq))
print(next(fibSeq))
print(next(fibSeq))
print(next(fibSeq))
#print(next(fibSeq))
#print(next(fibSeq))
```

```
0
1
1
2
3
```

34 Generator Expression

- (x*x for x in range(n)) is a generator expression
- The first part of an expression is the yield value
- The second part is the for loop with the collection

[63]:
```
# Example 7: (Generator Expression is same as lambda function)
list = [1,2,3,4]

z = (x**0.5 for x in list)
print("Output of generator function: ",next(z))
print("Output of generator function: ",next(z))
print("Output of generator function: ",next(z))
#print(x)
```

```
Output of generator function:  1.0
Output of generator function:  1.4142135623730951
Output of generator function:  1.7320508075688772
```

35 List comprehension Vs Generator Expression

[64]:
```
[x**2 for x in range(5)]  #List comprehension
```

[64]:
```
[0, 1, 4, 9, 16]
```

[66]:
```
GE = (x**2 for x in range(5))  #Generator Expression
next(GE)
next(GE)
next(GE)
```

[66]:
```
4
```

36 Generate Infinite Sequence

[69]:
```
# Example 8: Generate Infinite Sequence
def iS():
    n = 0
    while True:
        yield n
        n += 1
```

```
#for i in iS():
#    print(i)
```