```
MAP / Filter / Zip / Reducer

def Multiply_by2(l1): #Pure Function
    multipl_number=[]
    for item in l1:
        multipl_number.append(item*2)
    return multipl_number

print(Multiply_by2([1,2,3]))
*****************************************
#Using Map
def Addition_2(item):
    return item*2

print(list(map(Addition_2,[1,2,3])))
*****************************************
#Using Filter

def odd_output (item):
    return item %2 !=0
print(list(filter(odd_output,[1,2,3])))
*****************************************
#Using Zip
my_list = [1,2,3]
your_list=(10,20,30)
print(list(zip(my_list,your_list)))
*****************************************
# Using reducer
from functools import reduce
def accumilator(acc , item):
    print(acc , item)
    return acc + item

print(reduce(accumilator,my_list,0))
```

```
 LAMBDA
(i)
my_list = [1,2,3]

print("lambda function (by map):  ",list(map(lambda item : item*2 ,
my_list)))

print("lambda function (by filter):  ",list(filter(lambda item :
item%2!=0 , my_list)))

print("lambda function (by reducer):  ",reduce(lambda acce,
item:acce+item, my_list))



(ii)
from functools import reduce
my_list = [1,2,3]

print("lambda function (by map):  ",list(map(lambda item : item*2 ,
my_list)))

print("lambda function (by filter):  ",list(filter(lambda item :
item%2!=0 , my_list)))

print("lambda function (by reducer):  ",reduce(lambda acce,
item:acce+item, my_list))
```

```
 List / set / Dictionary Comprehension ()/{}/{:}
**************************************
my_data_list   =[char for char in 'hello']
my_data_list2  =[num for  num in range(0,100)]
my_data_list3  =(num**2 for num in range (0,100))
my_data_list4  = [num**2 for num in range (0,100) if num %2==0]

print('\nList Comprehension (Character): ',my_data_list)
print('\nList Comprehension (range): ',my_data_list2)
print('\nList Comprehension (square root): ',my_data_list3)
print('\nList Comprehension(even from square root): ',my_data_list4)

simple_dict={
    'a':1,
    'b':2
}
my_dict={key : value*2 for key,value in simple_dict.items() if value
%2==0}
print('\nDict Comprehension(only even): ',my_dict)  # Dict Comprehension

my_dict2= {value : value*2 for value in [1,2,3]}  # Dict Comprehension
print('\nDict Comprehension: ',my_dict2)
```

```
 Decorators
****************
#using fuctions as varables/ parameter for other functions.
# high order functions - a function can accept or return or both
another  function

def my_decorators(func):
    def wrap_func():
        print("********************")
        func()
        print("********************")
    return wrap_func

@my_decorators
def hello():
    print('helloooo')   #************************ (1) Using Decorator

def hey():
    print('heyyyy')     #************************ (2) Normal Function

hello()  #************************ (1) Using Decorator

hey2=my_decorators(hey)
hey2()                  #************************ (2)  Normal Function
#wrapping Normal function with Decorator how simple is that! Decorator's
are doing (2)
```

```
Try:  except: else: finally
**************************
(i)

while True:
    try:
        age = int(input("Enter Your Age : "))
        10/age
        print(age)
    except ValueError:
        print("Please Enter a number ")
    except ZeroDivisionError:
        print("Please enter a number Other than 0")
    else:
        print("Thankyou")
        break
    finally:
        print("ok done")  #break function not working!


(ii)

def sum(num1, num2):
    try:
        return num1/num2
    except (TypeError,ZeroDivisionError) as err:
        print(err)

print(sum(1,0))
print(sum(1,'2'))

(iii)

while True:
    try:
        age = int(input("Enter Your Age : "))
        10/age
        raise ValueError('Hey cut it out')  # creating a Error..
    except ZeroDivisionError:
        print("Please enter a number Other than 0")
```

```
 Generators
 ***************************************
def generator_function(num):
    for i in range(num):
        yield i*2
# From this loop Generator's Can able to Execute one at a time. that's
what generator's do.
# next function in print used to call output
# iter or __iter__ function used to call next until stopIteration

g=generator_function(10)
print(next(g))
next(g)
print(next(g))
```