Closure
Closure is nested function.
A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory
Closures can avoid the use of global values and provides some form of data hiding or security
A closure is a inner function which performs operation using data of outer function.

Syntax
```
def <outer-function>(arg,arg,…):
        def <inner –function>(arg,…):
                statement-1
                statement-2
        return inner-function
```

Example:
```
def find_power(num):
    def power(p):
        return num**p
    return power


def main():
    power5=find_power(5)
    res1=power5(2)
    res2=power5(3)
    print(res1,res2)
    power6=find_power(6)
    res3=power6(2)
    res4=power6(3)
    print(res3,res4)
    res5=power5(4)
    print(res5)
main()
```

Output:
```
25 125
36 216
```

```
625
```

Example:

```
def calculator(num1,num2):
    def calculate(opr):
        if opr=='+':
            return num1+num2
        if opr=='-':
            return num1-num2
        if opr=='*':
            return num1*num2
        if opr=='/':
            return num1/num2
    return calculate


def main():
    calc1=calculator(5,2)
    res1=calc1('+')
    res2=calc1('-')
    res3=calc1('*')
    res4=calc1('/')
    calc2=calculator(6,3)
    r1=calc2('+')
    r2=calc2('-')
    print(res1,res2,res3,res4)
    print(r1,r2)
main()
```

Output:

```
======== RESTART: F:/python6pmaug/funtest45.py =======
7 3 10 2.5
9 3
```

Generators

which returns a generator iterator. It looks like a normal function except that it contains yield expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function.

generator iterator
An object created by a generator function.

yield keyword
generator function return value using yield keyword. After returning value yield keyword pause execution of function.

Example:
```
def fun1():
    yield 10
    yield 20
    yield 30


def main():
    f1=fun1() # return generator iterator object
    v1=next(f1)
    v2=next(f1)
    v3=next(f1)
    print(v1,v2,v3)

main()
```

Output:
```
10 20 30
```

Example:

```
def prime_generator(start,stop):
    for num in range(start,stop+1):
        c=0
        for i in range(1,num+1):
            if num%i==0:
                c+=1
        if c==2:
            yield num

def main():
    p=prime_generator(5,10)
    for n in p:
        print(n)
main()
```

Output:
```
5
7
```

Example:
```
import random
def random_generator(start,stop,count):
    for i in range(count):
        rn=random.randint(start,stop)
        yield rn

def main():
    rg=random_generator(5,10,3)
    n1=next(rg)
    print(n1)
    n2=next(rg)
    print(n2)
    n3=next(rg)
    print(n3)

main()
```

Output:
```
5
8
5
```

Example:
```
def custom_iter(coll):
    for i in range(-1,-(len(coll)+1),-1):
        yield coll[i]

def main():
    list1=list(range(10,110,10))
    print(list1)
    ci=custom_iter(list1)
    for value in ci:
        print(value)

main()
```

Output:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
100
90
80
70
60
50
40
30
20
10
```

Lambda Functions or Lambda Expressions

Lambda function is anonymous function; it is a function which does not any name.
Lambda functions are called higher order functions.
A function which is send as an argument to another function is called higher order function.

Syntax:

```
lambda arg1,arg2:expression
```

lambda expression can be defined,

1. With arguments
2. Without arguments