---: Higher order function :---

A function in Python with another function as an argument or returns a function as an output is called the High order function. A function that is having another function as an argument or a function that returns another function as a return in the output

- The function can be stored in a variable.
- The function can be passed as a parameter to another function.
- The high order functions can be stored in the form of lists, hash tables, etc.
- Function can be returned from a function.

1. **Map()**
2. **Filter()**
3. **Lambda()**
4. **Reduce()**
5. **Decorators()**
6. **Generators()**

---: Map :----

Python's map() is a built-in function that enables the processing and transformation of all items in an iterable without the need for an explicit for loop, a technique referred to as mapping. This function is particularly useful when you want to apply a transformation function to each element in an iterable, producing a new iterable as a result. map() is one of the tools that facilitate a functional programming approach in Python.

map() Syntax

map(function, iterable, ...)

map() Arguments

The map() function takes two arguments:

1. function - a function
2. iterable - an iterable like sets, lists, tuples, etc

The map() function returns an object of map class. The returned value can be passed to functions like list() - to convert to list, set() - to convert to a set, and so on.

Example:-1

```
# Map() higher order function----------

my_list=[10,20,30,40]

def sqr(n):
    return n*n

x=map(sqr,my_list)
print(x)
print(list(x))

O/P:--
<map object at 0x000001EA310E3490>
[100, 400, 900, 1600]
```

Example:-2

```
my_tuple=(10,20,30,40)

def sqr(n):
    return n*n

x=map(sqr,my_tuple)
print(x)
print(tuple(x))

O/P:-
<map object at 0x0000019833A83490>
(100, 400, 900, 1600)
```

Example:-3

```
my_str="Neeraj"
def add(n):
    x=ord(n)
    return x
x=map(add,my_str)
print(x)
print(list(x))

O/P:-
<map object at 0x000001D03A4E3490>
[78, 101, 101, 114, 97, 106]
```

Example:-4

```
my_str="Neeraj"
def add(n):
    x=ord(n)
    return chr(x+5)
x=map(add,my_str)
print(x)
print(list(x))

O/P:-
<map object at 0x0000026D8F1634C0>
['S', 'j', 'j', 'w', 'f', 'o']
```

---: Filter :---

The filter function extracts elements from an iterable (such as a list or tuple) based on the results of a specified function. This function is applied to each element of the iterable, and if it returns True that element is included in the output of the filter() function.

filter() Syntax

The syntax of filter() is: **filter(function, iterable)**

filter() Arguments

The filter() function takes two arguments:

1. **function** - a function
2. **iterable** - an iterable like sets, lists, tuples etc.

The filter() function returns an iterator.

Example 1:-

```
# filter() higher order function ----------------
my_list=[60,10,70,90,55,75,10,20,40]

def fun(n):
    if n>=60:
        return True

x=filter(fun , my_list)
print(list(x))

O/P:--
[60, 70, 90, 75]
```

Example 2:-

```
def check_even(number):
    if number % 2 == 0:
        return True
    return False

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers_iterator = filter(check_even, numbers)
even_numbers = list(even_numbers_iterator)
print(even_numbers)

O/P:--
[2, 4, 6, 8, 10]
```

Example 3:-

```
def check_odd(number):
    if number % 2 != 0:
        return True
    return False

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers_iterator = filter(check_odd, numbers)
odd_numbers = list(odd_numbers_iterator)
print(odd_numbers)

O/P:--
[1, 3, 5, 7, 9]
```

---: Lambda :---

a lambda function is a special type of function without the function name. For example, lambda : print('Hello World').

A lambda function can take any number of arguments, but can only have one expression.

Here, we have created a lambda function that prints 'Hello World'.

lambda Function Declaration: We use the **lambda keyword** instead of def to create a lambda function. Here's the syntax to declare the lambda function:

Syntex:----

lambda argument(s) : expression

argument(s) - any value passed to the lambda function

expression - expression is executed and returned

Let's see an example,

```
# without argument
greet = lambda : print('Hello World')
greet()

O/P:--
Hello World
```

greet = lambda : print('Hello World'). Here, we have defined a lambda function and assigned it to the variable named greet. In the above example, we have defined a lambda function and assigned it to the greet variable. When we call the lambda function, the print() statement inside the lambda function is executed.

```
# with argument
x=lambda p,q,r:3*p+4*q+5*r+5
print(x(10,20,30))
O/P:-
265
```

```
# with argument
user = lambda name : print('Hello', name)
user('Neeraj')

O/P:--
Hello Neeraj
```

```
add = lambda x,y,z : print(x+y+z)

# add(1,2,3)

add = lambda x,y,z : (x+y+z)
print(add(1,2,3))

# lambda with if-else.
x = lambda n : "+ Positive" if n>0 else "- Negative" if n<0 else "Zero"
n = int(input("Enter any value :"))
print(x(n))

# lambda with map()

l1 = [1,2,3,4,5]

res = list(map(lambda x: x**2,l1))
print(res)

# lambda with filter()

l1 = [1,2,3,4,5]

res = list(filter(lambda x: x if (x%2!=0) else False ,l1))
print(res)
```

```python
# lambda with reduce()

from functools import reduce

l1 = [1,2,3,4,5]
res = reduce(lambda x,y: x+y,l1)
print(res)

# maximum.....
l1 = [1,2,3,4,5]
res = reduce(lambda x,y: x if x>y else y,l1)
print(res)

# minimum.....
l1 = [1,2,3,4,5]
res = reduce(lambda x,y: x if x<y else y,l1)
print(res)

# multiplication
l1 = [1,2,3,4,5]
res = reduce(lambda x,y: x*y,l1)
print(res)


res = lambda x: [print(i) for i in x]
```

---: Reduce :---

The reduce() function in Python is part of the functools module, which needs to be imported before it can be used.

This function performs functional computation by taking a function and an iterable (such as a list, tuple, or dictionary) as arguments. It applies the function cumulatively to the elements of the iterable, reducing it to a single value. Unlike other functions that may return multiple values or iterators, reduce() returns a single value, which is the result of the entire iterable being condensed into a single integer, string, or boolean.

Steps of how to reduce function works:

1. The function passed as an argument is applied to the first two elements of the iterable.
2. After this, the function is applied to the previously generated result and the next element in the iterable.
3. This process continues until the whole iterable is processed.
4. The single value is returned as a result of applying the reduce function on the iterable.

```
from functools import reduce

def product(x,y):
    return x*y

ans = reduce(product, [2, 5, 3, 7])
print(ans)

O/P:--
210
```

```
import functools

my_list=(10,20,60,30,40)
def greater(a,b):
    if a>b:
        return a
    else:
        return b
x=functools.reduce(greater,my_list)
print(x)

O/P:-
60

my_list=(10,20,60,30,40)
def lowest_digit(a,b):
    if a<b:
        return a
    else:
        return b
x=functools.reduce(lowest_digit,my_list)
print(x)

O/P:-
10

my_str="Neeraj"
def greater(a,b):
    if a>b:
        return a
    else:
        return b
x=functools.reduce(greater,my_str)
print("This char have greater asci value:",x)

O/P:-
This char have greater asci value: r
```

Combine example for map, filter, reduce and lambda

```
x = lambda x : x+5
print(x(5))

x = lambda x : print(x+5)
x(5)
x(10)

# WAP to print squar of all odd numbers in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers = filter(lambda x: x % 2 != 0, my_list)
squared_odds = map(lambda x: x ** 2, odd_numbers)
print(list(squared_odds))

# WAP to print squar of all objects in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squar_data = list(map(lambda x : x**x,my_list))
print(squar_data)

# WAP to print squar of all odd no objects in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_data = filter(lambda x : x%2 !=0,my_list)
squar_data = list(map(lambda x : x**2,odd_data))
print(squar_data)

# WAP to print squar of all even no objects in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_data = filter(lambda x : x%2 ==0,my_list)
squar_data = list(map(lambda x : x**2,even_data))
print(squar_data)

from functools import reduce

# WAP to print sum of squar of all objects in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squar_data = list(map(lambda x : x**x,my_list))
data = reduce(lambda x,y :x+y,squar_data)
print(squar_data)
```

```
print(data)

# WAP to print sum of squar of all odd no objects in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_data = filter(lambda x : x%2 !=0,my_list)
squar_data = list(map(lambda x : x**2,odd_data))
data = reduce(lambda x,y :x+y,squar_data )
print(squar_data)
print(data)

# WAP to print sum of squar of all even no objects in my_list.
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_data = filter(lambda x : x%2 ==0,my_list)
squar_data = list(map(lambda x : x**2,even_data))
data = reduce(lambda x,y :x+y,squar_data )
print(squar_data)
print(data)


my_list = [10,15,20,25,35,40,45,50]
even_data = filter(lambda x:x%2 == 0,my_list)
print(list(even_data))


my_list = [10,15,20,25,35,40,45,50]
odd_data = filter(lambda x:x%2 != 0,my_list)
print(list(odd_data))

# WAP to check greater no amoung these three numbers.
max_digit = lambda x,y,z : x if (x>y and x>z) else y if (y>z and y>x) else z
print(max_digit(10,20,30))
```

---: Decorators :---

Decorators are the most common use of higher-order functions in Python. They enable programmers to modify the behavior of a function or class. By wrapping one function with another, decorators allow us to extend the behavior of the wrapped function without permanently changing it. In this process, functions are passed as arguments to another function and then called within the wrapper function.

```
# defining a decorator
def decorator(func):
   def inner1():
      print("Hello, this is before function execution")
      func()
      print("This is after function execution")
   return inner1
def function():
    print("This is inside the function !!")

function_used = decorator(function)
function_used()

O/P:--
Hello, this is before function execution
This is inside the function !!
This is after function execution

def decorator(func):
   def inner1():
      print("Hello, this is before function execution")
      func()
      print("This is after function execution")
   return inner1

@decorator # second method for calling
def function():
   print("This is inside the function !!")

function() # second method for calling
```

```
O/P:--
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

Examples:---

```
def greet(fun):
    def inner():
        print("Good morning")
        fun()
        print("Thanks for using")
    return inner

def hello():
    print("Hello world")

var=greet(hello)
var()

O/P:--
Good morning
Hello world
Thanks for using
```

With @decorator :------

```
def greet(fun):
    def inner():
        print("Good morning")
        fun()
        print("Thanks for using")
    return inner
@greet
def hello():
    print("Hello world")

hello()
```

```
O/P:--
Good morning
Hello world
Thanks for using
```

Nested decorators :---

```
def decorator1(fun):
    def inner():
        a=fun()
        add = a+5
        return add
    return inner

def decorator2(fun):
    def inner():
        b=fun()
        add = b+5
        return add
    return inner

def fun():
    return 100

fun = decorator2(decorator1(fun))
print(fun())

O/P:--
110
```

With @ decorators :-----

```
def decorator1(fun):
    def inner():
        a=fun()
        add = a+5
        return add
```

```
    return inner

def decorator2(fun):
    def inner():
        b=fun()
        add = b+5
        return add
    return inner

@decorator2
@decorator1
def fun():
    return 100
print(fun())

O/P:--
110
```

--- : Generators :---

Generators are similar to functions but produce a sequence of values that can be iterated over using loops. Instead of using return statements, generators use yield statements to return values one at a time.

In Python, a generator is a special type of function that allows you to create an iterator. It's a way to produce a sequence of values on demand, rather than generating them all at once and storing them in memory.

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
for y in var:
    print(y)

O/P:--
5
6
7
8
9
10
```

Next() function in generators:

If we want to retrieve elements from a generator, we can use the next function on the iterator returned by the generator. This is the other way of getting the elements from the generator. (The first way is looping in through it as in the examples above).

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
print("first object from generator :",next(var))
```

```
print("Second object from generator :",next(var))
for y in var:
    print(y)

O/P:--
first object from generator : 5
Second object from generator : 6
7
8
9
10
```

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))

O/P:--
object from generator : 5
object from generator : 6
object from generator : 7
object from generator : 8
object from generator : 9
object from generator : 10
```

```
def my_fun(x, y):
    while x<=y:
        yield x
        x+=1
var= my_fun(5, 10)
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
```

```
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
print("object from generator :",next(var))
O/P:--
object from generator : 5
object from generator : 6
object from generator : 7
object from generator : 8
object from generator : 9
object from generator : 10
Traceback (most recent call last):
  File "E:\Python Core_Advance\generators.py", line 25, in <module>
    print("object from generator :",next(var))
StopIteration
```