Lambda Functions or Lambda Expressions

“lambda” is a keyword which represents lambda expression or lambda function.
Lambda function is anonymous function; the function which does not have any name is called anonymous function.
Lambda function is single line function or single line expression. This function does not contain multiple lines.
Lambda functions in Python are small, anonymous functions defined using the lambda keyword. They can take any number of arguments but can only have one expression. Lambda functions are often used for short, simple operations where a full function definition is unnecessary.

Syntax:
lambda [parameters]:statement

lambda function does not uses return keyword for returning value.

Example:

```
a=lambda:print("lambda function")
a()
a()
a()
a()
```

Output

```
lambda function
lambda function
lambda function
lambda function
```

Note: lambda expression or function is assigned to a variable so that it can be called or invoked.

Example:

```
max2=lambda a,b:a if a>b else b
res1=max2(10,20)
print(res1)
add=lambda a,b:a+b
sub=lambda a,b:a-b
```

```
multiply=lambda a,b:a*b
div=lambda a,b:a/b
res2=add(10,5)
res3=sub(5,2)
res4=multiply(5,4)
res5=div(4,2)
print(res2,res3,res4,res5)
```

Output

```
20
15 3 20 2.0
```

Lambda functions can be used with higher order functions
A function which receives input as another function is called higher order function. Lambda functions can be used as arguments.

The following predefined functions are called higher order functions

1. filter()
2. map()
3. reduce()

filter(function, iterable)

Construct an iterator from those elements of iterable for which function is true. iterable may be either a sequence, a container which supports iteration, or an iterator. If function is None, the identity function is assumed, that is, all elements of iterable that are false are removed.

Example:

```
A=[1,2,3,4,5,6,7,8,9,10]
B=list(filter(lambda n:n%2==0,A))
C=list(filter(lambda n:n%2!=0,A))
print(A,B,C,sep="\n")
names=["naresh","kishore","kiran","ramesh","rajesh"]
names1=list(filter(lambda name:name[-1]=='h',names))
print(names,names1,sep="\n")
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 6, 8, 10]
[1, 3, 5, 7, 9]
```

['naresh', 'kishore', 'kiran', 'ramesh', 'rajesh']
['naresh', 'ramesh', 'rajesh']

map(function, iterable, *iterables)
Return an iterator that applies function to every item of iterable, yielding the results. If additional iterables arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

Example:

```
A=[1,2,3,4,5]
B=[10,20,30,40,50]
C=list(map(lambda x,y:x+y,A,B))

print(A,B,C,sep="\n")

names=["naresh","suresh","ramesh","kishore","raman"]
names1=list(map(lambda name:name.upper(),names))
print(names)
print(names1)

A=["10","20","30","40","50"]
B=list(map(int,A))
print(A,B,sep="\n")
```

Output

```
[1, 2, 3, 4, 5]
[10, 20, 30, 40, 50]
[11, 22, 33, 44, 55]
['naresh', 'suresh', 'ramesh', 'kishore', 'raman']
['NARESH', 'SURESH', 'RAMESH', 'KISHORE', 'RAMAN']
['10', '20', '30', '40', '50']
[10, 20, 30, 40, 50]
```

functools.reduce(function, iterable[, initializer])
Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value.

Example:

```
import functools
A=[1,2,3,4,5,6,7,8,9,10]
res1=functools.reduce(lambda x,y:x+y,A)
print(res1)
res2=functools.reduce(lambda x,y:x if x>y else y,A)
print(res2)
```

Output

```
55
10
```

Function Recursion OR Recursive Function

Calling function within same function is called function recursion.

Recursion in Python is a programming technique where a function calls itself within its own definition. This approach is particularly useful for solving problems that can be broken down into smaller, self-similar sub-problems. A recursive function typically has two main parts: a base case and a recursive case. The base case defines the condition under which the recursion stops, preventing the function from calling itself indefinitely. The recursive case, on the other hand, is where the function calls itself with a modified input, working towards the base case.

What is default recursion depth or base case?

If condition is not defined or base case is not defined, the function is called 1000 times (default recursion depth)

How to find default recursion depth?

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

How to modify recursion depth?

```
>>> sys.setrecursionlimit(500)
```

def fun1():
  print("inside fun1")
  fun1() # recursive case
	Output
	Inside fun1
	Inside fun1
	Inside fun1

fun1()	Inside fun1 …. 1000 times

import sys
def fun1():
  print("inside fun1")
  fun1() # recursive case
 	
sys.setrecursionlimit(500)
fun1()
	Output
	Inside fun1
	Inside fun1
	Inside fun1
	….
	
	500 times