Lambda Function (Anonymous Functions) in Python
The name of the function is the mandatory item in the function definition. But, in python, we have a keyword called ‘lambda’ with which we can define a simple function in a single line without actually naming it. Such functions are called lambda functions.
Syntax: lambda argument_list: expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming, besides other types of expressions in functions.

Example: Lambda Function in Python

```
s = lambda a: a*a
x=s(4)
print(x)
```

Output: 16

Example: Lambda Function in Python

```
add = lambda a,b: a+b
x=add(4,5)
print(x)
```

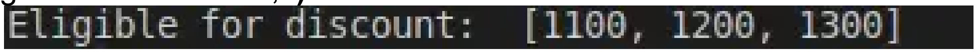
Output: 9

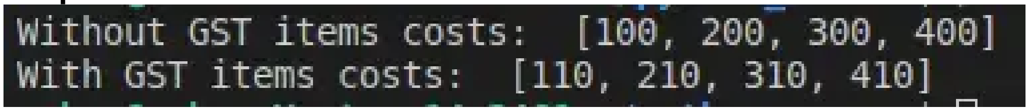
Points to Remember while working with Python Lambda Functions:

1. A lambda function can take any number of arguments but should have only one expression.
2. It takes the parameters and does some operation on them and returns the result, just the same as normal functions.
3. The biggest advantage of lambda functions is that a very concise code can be written which improves the readability of the code.
4. This can be used for only simple functions but not for complex functions.
5. The lambda function, mostly, can be used in combination with other functions such as **map(), reduce(), filter()** etc.

filter() function in Python:
This function is used to filter values from a sequence of values. The syntax is:
Syntax: filter(function, sequence)
The filter function will filter the elements in the sequence based on the condition in the function. Let’s understand it through the example.
Example: Filter Function in python

```
items_cost = [999, 888, 1100, 1200, 1300, 777]
gt_thousand = filter(lambda x : x>1000, items_cost)
x=list(gt_thousand)
```

```
print("Eligible for discount: ",x)
```
Output: 
```
Eligible for discount:  [1100, 1200, 1300]
```
In the above example, the filter applies the lambda function on all the elements in the ‘items_cost’ and returns the elements which satisfy the lambda function.

map() function in Python:
This function is used to map a particular function onto the sequence of elements. After applying, this returns a new set of values.
Syntax: map(function, sequence)
Example: Map Function in Python (Demo38.py)
```
without_gst_cost = [100, 200, 300, 400]
with_gst_cost = map(lambda x: x+10, without_gst_cost)
x=list(with_gst_cost)
print("Without GST items costs: ",without_gst_cost)
print("With GST items costs: ",x)
```
Output:

```
Without GST items costs:  [100, 200, 300, 400]
With GST items costs:  [110, 210, 310, 410]
```

reduce() function in Python:
This function reduces the sequence of elements into a single element by applying a specific condition or logic. To use the reduce function we need to import the functools module.
Syntax: reduce(function, sequence)
Example:
```
from functools import reduce
each_items_costs = [111, 222, 333, 444]
total_cost = reduce(lambda x, y: x+y, each_items_costs)
print(total_cost)
```
Output: 1110

```
import functools
lis = [1, 3, 5, 6, 2,4 ]
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

Output:
```
The maximum element of the list is : 6
```

With lambda function	**Without lambda function**
Supports single-line sometimes statements that return some value.	Supports any number of lines inside a function block
Good for performing short operations/data manipulations.	Good for any cases that require multiple lines of code.

Using the lambda function can sometime reduce the readability of code.	We can use comments and function descriptions for easy readability.

Python Lambda with Multiple Statements
Lambda functions do not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function.

```
List = [[2,3,4],[1, 4, 16, 64],[3, 6, 9, 12]]

sortList = lambda x: (sorted(i) for i in x)
secondLargest = lambda x, f : [y[len(y)-2] for y in f(x)]
res = secondLargest(List, sortList)

print(res)
```

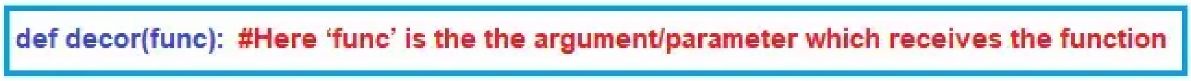
Output:

```
[3, 16, 9]
```

Decorators in Python with Examples:
Nested functions and function as first class object concepts already. Clear understanding of these concepts is important in understanding decorators. A decorator is a special function which adds some extra functionality to an existing function.

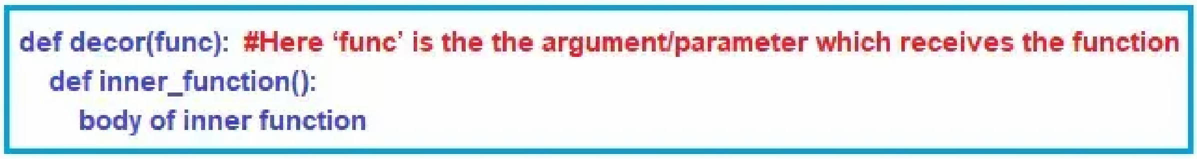
A decorator is a function that accepts a function as a parameter and returns a function.

Decorators are useful to perform some additional processing required by a function.
Steps to create decorator:
Step1: Decorator takes a function as an argument



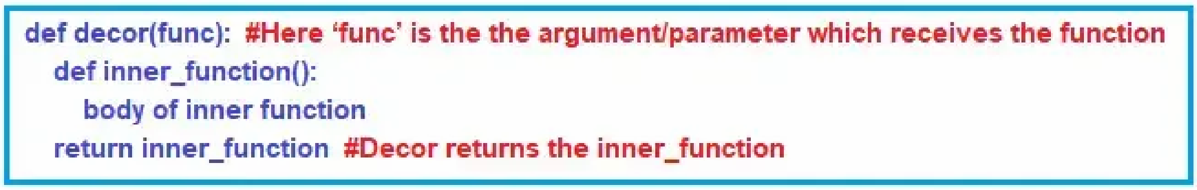
```
def decor(func):  #Here 'func' is the the argument/parameter which receives the function
```

Advertisements
Step2: Decorator body should have an inner function



```
def decor(func):  #Here 'func' is the the argument/parameter which receives the function
   def inner_function():
      body of inner function
```

Step3: Decorator should return a function

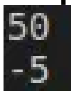


```
def decor(func):  #Here 'func' is the the argument/parameter which receives the function
   def inner_function():
      body of inner function
   return inner_function  #Decor returns the inner_function
```

Advertisements
Step4: The extra functionality which you want to add to a function can be added in the body of the inner_function.
Let's create a function which takes two arguments and prints the sum of them.
Example: Add Function

```
def add(a,b):
   res = a + b
   return res
print(add(20,30))
print(add(-10,5))
```

Output:



```
50
-5
```

Now, I wish to add some extra functionality of adding the two numbers only if they are positive. If any number is negative, then I wish to take it as 0 during adding. For adding this extra functionality let create a decorator.

We have created our decorator and now let's use it with our add function from previous example
add = decor(add)
With the above statement, we are passing the add function as parameter to the decorator function, which is returning inner_function. Now, the inner_function object or address will be overridden in the 'add' because we are capturing the returned function in it. After this, whenever we call add, the execution goes to inner_function in the decorator.
add(-10,20)
In inner_function, we are doing the extra logic for checking whether the arguments are positive or not. If not positive we are assigning them with zero. And we are

passing the processed values to the original add function which was sent to the decorator. Our final code will be

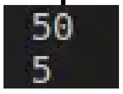
Example: Decorator Function in Python

```
def decor(func):
   def inner_function(x,y):
       if x<0:
           x = 0
       if y<0:
           y = 0
       return func(x,y)
   return inner_function

def add(a,b):
   res = a + b
   return res

add = decor(add)
print(add(20,30))
print(add(-10,5))
```

Output:



```
50
5
```

@ symbol in python:

In the above example, in order to use the decorator, we have used the **‘add = decor(add)’** line. Rather than using this we can just use the ‘@decor’ symbol on top of the function for which we want to add this extra functionality. The decorator once created can also be used for other functions as well.

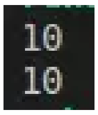
Example: Subtract Function using decorator function in Python

```
def decor(func):
   def inner_function(x,y):
       if x<0:
           x = 0
       if y<0:
           y = 0
       return func(x,y)
   return inner_function

@decor
def sub(a,b):
   res = a - b
   return res

print(sub(30,20))
print(sub(10,-5))
```

Output:



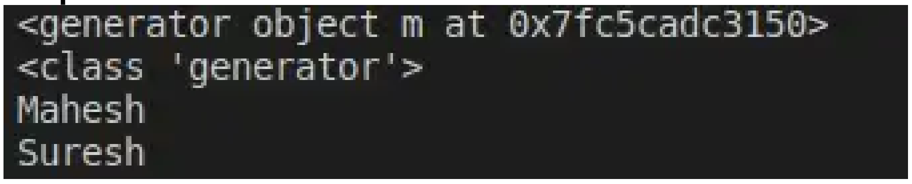
```
10
10
```

Generators in Python :
Generators are just like functions which give us a sequence of values one as an iterable (which can be iterated upon using loops). Generators contain yield statements just as functions contain return statements.
Example: Generators in Python

```
def m():
   yield 'Mahesh'
   yield 'Suresh'

g = m()
print(g)
print(type(g))
for y in g:
   print(y)
```

Output:



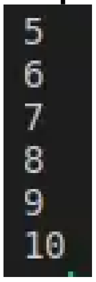
```
<generator object m at 0x7fc5cadc3150>
<class 'generator'>
Mahesh
Suresh
```

Example: Generators in Python

```
def m(x, y):
   while x<=y:
       yield x
       x+=1

g = m(5, 10)
for y in g:
   print(y)
```

Output:

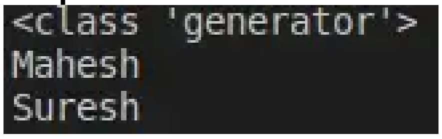


```
5
6
7
8
9
10
```

next function in Python:
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).
Example: Generators with next function in Python (Demo43.py)

```
def m():
   yield 'Mahesh'
   yield 'Suresh'
g = m()

print(type(g))
print(next(g))
print(next(g))
```
Output:



```
<class 'generator'>
Mahesh
Suresh
```

yield vs. return
The yield articulation is answerable for controlling the progression of the generator capability. By saving all states and yielding to the caller, it puts an end to the function's execution. Later it resumes execution when a progressive capability is called. In the generator function, we can make use of the multiple yield statement.
The return explanation returns a worth and ends the entire capability and just a single return proclamation can be utilized in the capability.

Difference between Generator function and Normal function

Typical capability contains just a single return explanation while generator capability can contain at least one yield proclamation.
The normal function is immediately halted and the caller is given control when the generator functions are called.
The states of the local variables are retained between calls.

What is Class Method in Python?
The @classmethod decorator is a built-in function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A class method receives the class as an implicit first argument, just like an instance method receives the instance
Syntax Python Class Method:
```
class C(object):
   @classmethod
   def fun(cls, arg1, arg2, ...):
      ....
```
fun: function that needs to be converted into a class method
returns: a class method for function.

- A class method is a method that is bound to the class and not the object of the class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.

What is the Static Method in Python?

A static method does not receive an implicit first argument. A static method is also a method that is bound to the class and not the object of the class. This method can't access or modify the class state. It is present in a class because it makes sense for the method to be present in class.

Syntax Python Static Method:

```
class C(object):
   @staticmethod
   def fun(arg1, arg2, ...):
      ...
```

returns: a static method for function fun.

Class method vs Static Method

The difference between the Class method and the static method is:

- A class method takes cls as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

When to use the class or static method?

- We generally use the class method to create factory methods. Factory methods return class objects (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

How to define a class method and a static method?

To define a class method in python, we use @classmethod decorator, and to define a static method we use @staticmethod decorator.

We use static methods to create utility functions. In the below example we use a static method to check if a person is an adult or not.

One simple Example :

class method:

Python

```
class MyClass:
   def __init__(self, value):
      self.value = value

   def get_value(self):
      return self.value
```

```
# Create an instance of MyClass
obj = MyClass(10)

# Call the get_value method on the instance
print(obj.get_value())  # Output: 10
```

Output

```
10
```

Static method:-

Python

```
class MyClass:
    def __init__(self, value):
        self.value = value

    @staticmethod
    def get_max_value(x, y):
        return max(x, y)

# Create an instance of MyClass
obj = MyClass(10)

print(MyClass.get_max_value(20, 30))

print(obj.get_max_value(20, 30))
```

Output

```
30
30
```

Below is the complete Implementation

Python3

```
# Python program to demonstrate
# use of class method and static method.
from datetime import date


class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a class method to create a Person object by birth year.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check if a Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18
```

```
person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print(person1.age)
print(person2.age)

# print the result
print(Person.isAdult(22))
```

Output:

```
21
25
True
```

Comprehension in Python

Overview:
Comprehensions are a powerful and concise way to create and manipulate lists , sets , and dictionaries . In this blog post, we'll explore what comprehensions are, how they work, and how to use them effectively in your Python code.

What are Comprehensions?
Python, comprehension is a concise way to create a new list, set, or dictionary based on an existing iterable object. Comprehensions are more concise and readable than traditional looping constructs such as for and while loops, and they can often be more efficient as well.
There are three types of comprehension in Python:

- List comprehensions
- Set comprehensions
- Dictionary comprehensions

List Comprehensions
List comprehensions are utilized to form a new list based on an existing iterable object, such as a list or a range. Here's an illustration of a basic list comprehension that makes a new list of squared values:

```
squares = [x**2 for x in range(10)]
```

```
print(squares)
```

In this case, we're employing a for loop to repeat over the range from 9, and we're utilizing the expression x**2 to produce a new value for each iteration. The resulting list contains the squared values from 0 to 81.
List comprehensions can also include conditional statements to filter the values contained in the new list. Here's an example that creates a new list of even numbers:

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

In this example, we're using the if statement to only include even values (i.e., values with a remainder of 0 when divided by 2). The resulting list contains even numbers from 0 to 8.

Set Comprehensions
Set comprehensions are similar to list comprehensions, but they're used to create a new set based on an existing iterable object. Here's an example of a simple set comprehension that makes a new set of unique values:

```
unique = {x % 3 for x in range(10)}
print(unique)
```

In this example, we're using the expression x % 3 to generate a new value for each iteration, and the resulting set contains unique values from 0 to 2. Set comprehensions can also include conditional statements to filter the values contained in the new set, just like list comprehensions.
Dictionary Comprehensions
Dictionary comprehensions create a new dictionary based on an existing iterable object. Here's an example of a simple dictionary comprehension that makes a new dictionary of key-value pairs:

```
squares = {x: x**2 for x in range(10)}
print(squares)
```

In this example, we're using the expression x**2 to generate a new value for each iteration and the value of x as the key for each key-value pair. The resulting dictionary contains the squared values from 0 to 81, with the keys being the corresponding integers.

Dictionary comprehensions can also include conditional statements to filter the key-value pairs included in the new dictionary.
Benefits of Comprehensions
The main benefits of using comprehensions in your Python code are:

1. **Conciseness:** Comprehensions allow you to write more concise code and are easier to read than traditional looping constructs.
2. **Readability:** Comprehensions are often more readable than traditional looping constructs because they clearly express the intent of the code in a single line.

3. **Efficiency:** Comprehensions are often more efficient than traditional looping constructs because they use the underlying iterator protocol and avoid creating unnecessary intermediate objects.
4. **Flexibility:** Comprehensions can be used with any iterable object, including lists, tuples, sets, and dictionaries, making them a versatile tool for data manipulation in Python.