UPES

UNIVERSITY WITH A PURPOSE

Python-Module3
Different types of Functions in Python

UPES
UNIVERSITY WITH A PURPOSE

Lambda Function

- In Python, an anonymous function is a function that is defined without a name.

- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.

- Hence, anonymous functions are also called **lambda functions**.

- A lambda function in python has the following syntax.
- ***lambda arguments: expression***
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.
- Lambda functions can be used wherever **function objects** are required.

Lambda Function Example

```
def doubler(x):
    return x*2

print(doubler(2))
# Prints 4
```

Lambda Function Example

```
def doubler(x):
    return x*2

print(doubler(2))
# Prints 4
```

```
doubler = lambda x: x*2

print(doubler(2))
# Prints 4
```

UPES
UNIVERSITY WITH A PURPOSE

Lambda Function Characteristics

- No Statements Allowed: A lambda function can not contain any statements in its body. Statements such as return in a lambda function will raise a SyntaxError.

- Single Expression Only: Unlike a normal function, a lambda function contains only a single expression.

- Multiple Arguments: We can send as many arguments as you like to a lambda function; just separate them with a comma , operator.

```
mul = lambda x, y: x*y
print(mul(2, 5))
```

UPES
UNIVERSITY WITH A PURPOSE

Lambda Function Characteristics

- Ways to Pass Arguments: Like a normal function, a lambda function supports all the different ways of passing arguments. This includes:
  - Positional arguments

```
add = lambda x, y, z: x+y+z
print(add(2, 3, 4))
# Prints 9
```

  - Keyword arguments

```
add = lambda x, y, z: x+y+z
print(add(2, z=3, y=4))
# Prints 9
```

Lambda Function Characteristics

- Ways to Pass Arguments: Like a normal function, a lambda function supports all the different ways of passing arguments. This includes:
  - Default argument

```
add = lambda x, y=3, z=4: x+y+z
print(add(2))
# Prints 9
```

  - Variable list of arguments (*args)

```
add = lambda *args: sum(args)
print(add(2,3,5,7))
# Prints 17
```

UPES
UNIVERSITY WITH A PURPOSE

Lambda Function Characteristics

- Ways to Pass Arguments: Like a normal function, a lambda function supports all the different ways of passing arguments. This includes:
  - Variable list of keyword arguments (**args)

```
add = lambda **kwargs: sum(kwargs.values())
print(add(x=2, y=3,z=3,f=8))
# Prints 16
```

UPES
UNIVERSITY WITH A PURPOSE

If Else with Lambda Function

- Generally if else statement is used to implement selection logic in a function.
- But as it is a statement, you cannot use it in a lambda function.
- You can use the if else ternary expression instead.

```
findMin = lambda x, y: x if x < y else y
print(findMin(2, 4))
# Prints 2
print(findMin('a', 'x'))
# Prints a
```

UPES
UNIVERSITY WITH A PURPOSE

Map Function

- The map() function expects two arguments: a function and a list.
- It takes that function and applies it on every item of the list and returns the modified list.

Map Function Example

```
def doubler(x):
    return x*2

L = [1, 2, 3, 4, 5, 6]

mod_list = map(doubler, L)

print(list(mod_list))

# Prints [2, 4, 6, 8, 10, 12]
```

Map Function Example

```
def doubler(x):
    return x*2

L = [1, 2, 3, 4, 5, 6]

mod_list = map(doubler, L)

print(list(mod_list))

# Prints [2, 4, 6, 8, 10, 12]
```

```
# Double each item of the list

L = [1, 2, 3, 4, 5, 6]

doubler = map(lambda x: x*2, L)

print(list(doubler))

# Prints [2, 4, 6, 8, 10, 12]
```

UPES
UNIVERSITY WITH A PURPOSE

Mapping Function in a Dictionary

```
dOfNames = {7 : 'sam', 8: 'john', 9: 'mathew', 10: 'riti', 11 : 'aadi',   12 : 'sachin'}
dOfNames = dict(map(lambda x: (x[0], x[1] + '_'), dOfNames.items()))
print('Modified Dictionary : ')
print(dOfNames)
```

Output:

```
Modified Dictionary :
{7: 'sam_', 8: 'john_', 9: 'mathew_', 10: 'riti_', 11: 'aadi_', 12: 'sachin_'}
```

Sorted Function

- The sorted() function sorts the elements of a given sequence(string, list, tuple) or collection(dictionary, set, frozenset) in a specific order (either ascending or descending) and returns the sorted sequence or collection as a list.
- The syntax of the sorted() function is:

  sorted(item, key=None, reverse=False)

  Item is sequence or collection

  reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

  key (Optional) - A function that serves as a key for the sort comparison. Defaults to None.
- A list also has the sort() method which performs the same way as sorted(). The only difference is that the sort() method doesn't return any value and changes the original list.

Sorted Function Example

```
# List
py_list = ['e', 'a', 'u', 'o', 'i']
print(sorted(py_list))

# string
py_string = ‘Python'
print(sorted(py_string))

# vowels tuple
py_tuple = ('e', 'a', 'u', 'o', 'i')
print(sorted(py_tuple))
```

Output

```
['a', 'e', 'i', 'o', 'u']

['P', 'h', 'n', 'o', 't', 'y']

['a', 'e', 'i', 'o', 'u']
```

Sorted Function Example

```
# set
py_set = {'e', 'a', 'u', 'o', 'i'}
print(sorted(py_set, reverse=True))

# dictionary
py_dict = {'e': 1, 'a': 2, 'u': 3, 'o': 4, 'i': 5}
print(sorted(py_dict, reverse=True))
```

Output

```
['u', 'o', 'i', 'e', 'a']

['u', 'o', 'i', 'e', 'a']
```

UPES
UNIVERSITY WITH A PURPOSE

Sorted Function

- If you want your own implementation for sorting, sorted() also accepts a key function as an optional parameter.
- Based on the returned value of the key function, you can sort the given item.
- E.g sorted(item, key=len)
- Here, len() is Python's in-built function to count the length of an object.
- The list is sorted based on the length of the element, from the lowest count to highest.
- EX:

```
item=["S","SS","aaaa","cc"]
print(sorted(item, key=len))
```

UPES
UNIVERSITY WITH A PURPOSE

Key also takes user-defined functions as its value for the basis of sorting.

```
# Sort a list of integers based on
# their remainder on dividing from 7
 def func(x):
    return x % 7
L = [15, 3, 11, 7]
print ("Normal sort :", sorted(L))
print ("Sorted with key:", sorted(L, key = func))
```

Output :

```
Normal sort : [3, 7, 11, 15]
Sorted with key: [7, 15, 3, 11]
```

Sorting Dictionaries

```
dt={6:"Sahil",4:"Tarun",5:"Anurag",2:"Ankur",1:"Ravi",3:"Pankaj"}
#sorting dictionary by key values
c1=dict(sorted(dt.items(),key=lambda t:t[0]))
print(c1)
#sorting dictionary by values
c2=dict(sorted(dt.items(),key=lambda t:t[1]))
print(c2)
```

Output:

Sort a dictionary

- A dictionary in Python is a data structure which stores values as a key-value pair. We can sort this type of data by either the key or the value and this is done by using the sorted() function.
- First, we need to know how to retrieve data from a dictionary to be passed on to this function.

Retrieving data

There are three basic ways to get data from a dictionary:

- Dictionary.keys() : Returns only the keys in an arbitrary order.
- Dictionary.values() : Returns a list of values.
- Dictionary.items() : Returns all of the data as a list of key-value pairs.

UPES
UNIVERSITY WITH A PURPOSE

Depending on the data we pass on to the sorted() function, we have three different ways to sort a Dictionary in Python.

```
dict = {}
dict['1'] = 'apple'
dict['3'] = 'orange'
dict['2'] = ‘mango'
lst = dict.keys()
# Sorted by key
print("Sorted by key: ", sorted(lst))
```

Output

```
Sorted by key: ['1', '2', '3']
```

```
dict = {}
dict['1'] = 'apple'
dict['3'] = 'orange'
dict['2'] = ‘mango'
lst = dict.values()
#Sorted by value
print("Sorted by value: ", sorted(lst))
```

Output

```
Sorted by value:  ['apple', 'orange', 'pango']
```

```
dict = {}
dict['1'] = 'apple'
dict['3'] = 'orange'
dict['2'] = ‘mango'
lst = dict.items()
#Sorted by item
print("Sorted by value: ", sorted(lst))
```

Output

```
Sorted by items:  [('1', 'apple'), ('2', ‘mango'), ('3', 'orange')]
```

Filter Function

- The filter() function is similar to the map().

- It takes a function and applies it to each item in the list to create a new list with only those items that cause the function to return True.

Filter Function Example

```
def checkAge(age):
    if age > 18:
        return True
    else:
        return False

age = [5, 11, 16, 19, 24, 42]
adults = filter(checkAge, age)
print(list(adults))
# Prints [19, 24, 42]
```

Filter Function Example

```
def checkAge(age):
    if age > 18:
        return True
    else:
        return False

age = [5, 11, 16, 19, 24, 42]
adults = filter(checkAge, age)
print(list(adults))
# Prints [19, 24, 42]
```

```
# Filter the values above 18

age = [5, 11, 16, 19, 24, 42]
adults = filter(lambda x: x > 18, age)
print(list(adults))
# Prints [19, 24, 42]
```

Questions

- Use filter function that filters vowels from the list [a,e,d,f,g,t,I,c,w]
- Use filter function and prints even list and odd numbers list less than 20.

UNIVERSITY WITH A PURPOSE

Global, Local variables

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Example 1: Create a Global Variable

```
x = "global"

def foo():
    print("x inside:", x)

foo()
print("x outside:", x)
```

Output

```
x inside: global
x outside: global
```

Local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

Example 2: Accessing local variable outside the scope

```
def foo():
    y = "localmksfklds"
foo()
print(y)
```

Output

```
NameError: name 'y' is not defined
```

The output shows an error because we are trying to access a local variable y in a global scope whereas the local variable only works inside foo() or local scope.

UPES

UNIVERSITY WITH A PURPOSE

Example 3: Create a Local Variable

Normally, we declare a variable inside the function to create a local variable.

```
def foo():
    y = "local"
    print(y)

foo()
```

Output

```
local
```

Example 5: Global variable and Local variable with same name

```
x = 5 #global

def foo():
    x = 10 #local
    print("local x:", x)

foo()
print("global x:", x)
```

Output

```
local x: 10
global x: 5
```

In the above code, we used the same name x for both global variable and local variable. We get a different result when we print the same variable because the variable is declared in both scopes, i.e. the local scope inside foo() and global scope outside foo().

When we print the variable inside foo() it outputs local x: 10. This is called the local scope of the variable.

Similarly, when we print the variable outside the foo(), it outputs global x: 5. This is called the global scope of the variable.

Python Inner Functions

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

UPES
UNIVERSITY WITH A PURPOSE

```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    printer()

print_msg("Hello")
```

Output

```
Hello
```

Why use Inner Functions?

Encapsulation

A function can be created as an inner function in order to protect it from everything that is happening outside of the function.

It is important to mention that the outer function has to be called in order for the inner function to execute. If the outer function is not called, the inner function will never execute.

```
def function1(): # outer function
    print ("Hello from outer function")
    def function2(): # inner function
        print ("Hello from inner function")
    function2()
```

The code will return nothing when executed!

```
def function1(): # outer function
    print ("Hello from outer function")
    def function2(): # inner function
        print ("Hello from inner function")
    function2()
function1()
```

Closure Function

- An important point to keep in mind is that ***functions are first class objects*** in python; i.e. functions can be passed as parameters, returned from other functions and assigned to any variable.
- A *nested function*, also called an inner function, is a function defined inside another function.
- A closure is a nested function which has access to a free variable from an enclosing function that has finished its execution. Three characteristics of a Python closure are:
  - It is a nested function
  - It has access to a free variable in outer scope: When a variable is used in a function or code block where it isn’t defined then the variable is called free variable.
  - It is returned from the enclosing function
- A free variable is a variable that is not bound in the local scope.

Nested function

```
def outer(name):
    # this is the enclosing function
    def inner():
        print(name)
    inner()
# call the enclosing function
outer('Tech')
```

```python
def outer(name):
    # this is the enclosing function
    def inner():
        # this is the enclosed function
        # the inner function accessing the outer function's variable 'name'
        print(name)
    return inner

# call the enclosing function
myFunction = outer('Tech')
myFunction()
```

UPES
UNIVERSITY WITH A PURPOSE

- Python **Closures** are these **inner functions** that are **enclosed within** the **outer function**. Closures can **access variables** present in the **outer function scope**. It can **access** these **variables** even after the **outer function** has **completed** its **execution**.

- The **inner function** becomes a **closure** when we **return** the **inner function instead of calling it.**

So we have a closure in Python if-

- We have a nested function, i.e. **function within a function**
- The nested function refers to a **variable** of the **outer function**
- The enclosing function **returns** the **enclosed function**
- Read more:
- https://techvidvan.com/tutorials/closures-in-python/#:~:text=Python%20Closures%20are%20these%20inner,function%20has%20completed%20its%20execution.

THANK YOU

UNIVERSITY WITH A PURPOSE