

NAME : AADHILAFEMINE AJ

REG NO : 121011012715

SUBJECT : APPLIED AI

SUBJECT CODE : XCSHDO3

DEPARTMENT : B.TECH(CSE)

SEMESTER : 5TH SEM

**LAMBDA FUNCTIONS:**

Lambda functions are similar to user-defined functions but without a name. They're commonly referred to as anonymous functions.

Lambda functions are efficient whenever you want to create a function that will only contain simple expressions – that is, expressions that are usually a single line of a statement.

Every anonymous function you define in Python will have 3 essential parts:

* The lambda keyword.
* The parameters (or bound variables), and
* The function body.

A lambda function can have any number of parameters, but the function body can only contain one expression. Moreover, a lambda is written in a single line of code and can also be invoked immediately.

**Syntax :**

The formal syntax to write a lambda function is as given below:

lambda p1, p2: expression

**example :**

adder = lambda x, y: x + y

print (adder (1, 2))

**The output for the above example** is 3.

### **Code Explanation for above example :**

Here, we define a variable that will hold the result returned by the lambda function.

**1.** The lambda keyword used to define an anonymous function.

**2.** x and y are the parameters that we pass to the lambda function.

**3.** This is the body of the function, which adds the 2 parameters we passed. Notice that it is a single expression. You cannot write multiple statements in the body of a lambda function.

**4.** We call the function and print the returned value.

**DEF PACKAGES AND MODULES:**

In Python, both modules and packages organize and structure the code but serve different purposes.

In simple terms, a module is a single file containing python code, whereas a package is a collection of modules that are organized in a directory hierarchy.

**modules :**

In Python, a module is a single file containing Python definitions and statements. These definitions and statements can include variables, functions, and classes and can be used to organize related functionality into a single, reusable package. Module organizes and reuses code in Python by grouping related code into a single file.

Modules can be imported and used in other Python files using the **import** statement.

Some popular modules in Python are math, random, csv, and datetime.

**Example:**

Consider a Python module math.py that contains a function to calculate the square of a number.

#math.py module

def square(i):

return x\*\*2

This module can be used be imported and used in the different files as follows:

#main.py file

import math

print(math.square(5))

**The output for the above code** is 25.

**Packages :**

Python Packages are collections of modules that provide a set of related functionalities, and these modules are organized in a directory hierarchy. In simple terms, packages in Python are a way of organizing related modules in a single namespace.

* Packages in Python are installed using a package manager like pip (a tool for installing and managing Python packages).
* Each Python package must contain a file named \_init\_.py.

**Example :**

Let there be any package (named my\_package) that contains two sub-modules (mod\_1, and mod\_2)

my\_package/

\_init\_.py

mod\_1.py

mod\_2.py

**Note:**

 init.py file is required to make python treat the dictionary as a package.

**MATRIX OPERATION:**

Matrix operation can perform some arithmetic operations like addition,subtraction and multiplication.

**MATRIX ADDITION:**

These matrices can be added if (if and only if) the order of the matrices are equal, i.e. the two matrices have the same number of rows and columns.

**Adding elements of the matrix:**

# importing numpy as np

import numpy as np

# creating first matrix

A = np.array([[1, 2], [3, 4]])

# creating second matrix

B = np.array([[4, 5], [6, 7]])

print("Printing elements of first matrix")

print(A)

print("Printing elements of second matrix")

print(B)

# adding two matrix

print("Addition of two matrix")

print(np.add(A, B))

**OUTPUT FOR MATRIX ADDITION :**

Printing elements of first matrix

[[1 2]

[3 4]]

Printing elements of second matrix

[[4 5]

[6 7]]

Addition of two matrix

[[ 5 7]

[ 9 11]]

**MATRIX SUBTRACTION:**

The subtraction of matrices is an operation where element-wise subtraction applies to the matrices of the same order, which essentially means that subtraction between two matrices can only happen when both of them have the same number of rows and columns.

**Subtracting elements of matrices:**

# importing numpy as np

import numpy as np

# creating first matrix

A = np.array([[1, 2], [3, 4]])

# creating second matrix

B = np.array([[4, 5], [6, 7]])

print("Printing elements of first matrix")

print(A)

print("Printing elements of second matrix")

print(B)

# subtracting two matrix

print("Subtraction of two matrix")

print(np.subtract(A, B))

**OUTPUT FOR MATRIX SUBTRACTION :**

Printing elements of first matrix

[[1 2]

[3 4]]

Printing elements of second matrix

[[4 5]

[6 7]]

Subtraction of two matrix

[[-3 -3]

[-3 -3]]

**MATRIX MULTIPLICATION :**

Matrix multiplication is a binary operation that uses a pair of matrices to produce another matrix. The elements within the matrix are multiplied according to elementary arithmetic.

In the multiplication of two matrices, the row elements of the first matrix are multiplied to the column elements of the second matrix.

**Matrix Multiplication using Nested Loop :**

# Program to multiply two matrices using nested loops

# 3x3 matrix

X = [[12,7,3],

[4 ,5,6],

[7 ,8,9]]

# 3x4 matrix

Y = [[5,8,1,2],

[6,7,3,0],

[4,5,9,1]]

# result is 3x4

result = [[0,0,0,0],

[0,0,0,0],

[0,0,0,0]]

# iterate through rows of X

for i in range(len(X)):

# iterate through columns of Y

for j in range(len(Y[0])):

# iterate through rows of Y

for k in range(len(Y)):

result[i][j] += X[i][k] \* Y[k][j]

for r in result:

print(r)

**OUTPUT FOR MATRIX MULTIPLICATION :**

[114, 160, 60, 27]

[74, 97, 73, 14]

[119, 157, 112, 23]

**MATRIX DIVISION** :

For matrices, there is no such thing as division. You can add, subtract, and multiply matrices, but you cannot divide them. There is a related concept, though, which is called “inversion”.but we can perform division operation in array.

Python's numpy. divide() computes the element-wise division of array elements. The elements in the first array are divided by the elements in the second array.

**SYNTAX :**

numpy.divide(arr1, arr2, out = None, where = True, casting = ‘same\_kind’, order = ‘K’, dtype = None) :

Array element from first array is divided by elements from second element (all happens element-wise). Both arr1 and arr2 must have same shape and element in arr2 must not be zero; otherwise it will raise an error.

**EXAMPLE 1 FOR (arr1 divide by arr2 elements):**

# Python program explaining

# divide() function

import numpy as np

# input\_array

arr1 = [2, 27, 2, 21, 23]

arr2 = [2, 3, 4, 5, 6]

print ("arr1 : ", arr1)

print ("arr2 : ", arr2)

# output\_array

out = np.divide(arr1, arr2)

print ("\nOutput array : \n", out)

**OUTPUT FOR THE EXAMPLE 1:**

arr1 : [2, 27, 2, 21, 23]

arr2 : [2, 3, 4, 5, 6]

Output array :

[ 1. 9. 0.5 4.2 3.83333333]

**EXAMPLE 2 FOR (elements of arr1 divided by divisor):**

# Python program explaining

# divide() function

import numpy as np

# input\_array

arr1 = [2, 27, 2, 21, 23]

divisor = 3

print ("arr1 : ", arr1)

# output\_array

out = np.divide(arr1, divisor)

print ("\nOutput array : \n", out)

**OUTPUT FOR THE EXAMPLE 2:**

arr1 : [2, 27, 2, 21, 23]

Output array :

[ 0.66666667 9. 0.66666667 7. 7.66666667]

**ITERATORS AND GENERATORS :**

**ITERATOR :**

An iterator is an object that can be iterated upon. Thus, iterators contain a countable number of values.

They are neatly implemented within for loops, comprehensions, generators, etc., but concealed from view. In simple words, an iterator is just an object that can be iterated on.

A Python iterator object must implement two specific methods, \_\_iter\_\_() or iter() and \_\_next\_\_() or next() , which are referred to collectively as the iterator protocol.

**Python iter():**

The iter() function in Python returns an iterator for the supplied object. The iter() generates a thing that can be iterated one element at a time. These items are handy when combined with loops such as for loops and while loops.

**Syntax:**

iter( object , sentinel )

iter() function takes two parameters:

Object: An object whose iterator needs to be created (lists, sets, tuples, etc.).

Sentinel (optional): Special value that represents the end of the sequence.

Python next()

The next() function returns the next item from the iterator. The next() function holds the value one at a time.

**Syntax:**

next( iterator , default )

The next() method accepts two parameters:

**Iterator :** next( ) function retrieves the next item from the iterator.

default(optional): this value is returned if the iterator is exhausted (not tired, but no next item to retrieve).

**EXAMPLE :**

class MyNumbers:

# \_\_iter\_\_() is same as iter()

def \_\_iter\_\_(self):

self.a = 1

return self

# \_\_next\_\_() is same as next()

def \_\_next\_\_(self):

# 20th is the highest value

if self.a <= 5:

x = self.a

# Manually increment

self.a += 1

# returning the iterator to the function call

return x

# Create the object of the class

myclass = MyNumbers()

# get an iterator using iter()

myiter = iter(myclass)

# printing the values using a for-in loop

for x in myiter:

print(x)

**OUTPUT:**

1

2

3

4

5

None

None

None

None

None

None

None

None

#------ goes on until control exit.

**GENERATOR :**

A special type of function which does not return a single value: it returns an iterator object with a sequence of values.

Python has a generator that allows you to create your iterator function. A generator is somewhat of a function that returns an iterator object with a succession of values rather than a single item. A yield statement, rather than a return statement, is used in a generator function.

The difference is that, although a return statement terminates a function completely, a yield statement pauses the function while storing all of its states and then continues from there on subsequent calls.

**EXAMPLE :**

# Program to print the Power of two up to the given number

def PowerTwoGen( max=0 ):

n = 1

while n < max:

yield 2 \*\* n

n += 1

a = PowerTwoGen(6)

# Printing the values stored in a

for i in a:

print(i)

**OUTPUT:**

2

4

8

16

32

**MAPS AND FILTERS :**

**MAPS :**

map() - Python's map() method applies a specified function to each item of an iterable (such as a list, tuple, or string) and then returns a new iterable containing the results.

The map() syntax is as follows: map(function, iterable)

The first argument passed to the map function is itself a function, and the second argument passed is an iterable (sequence of elements) such as a list, tuple, set, string, etc...

**Example 1 - usage of the map():**

# Using map() to square each element of the data list

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

# Map function returns the map object

squares = map(lambda x: x\*x, data)

# Iterating the elements of the squares

for i in squares:

print(i, end=" ")

# Also, we can convert the map object into a list

squares = list(map(lambda x: x\*x, data))

print(f"Squares: {squares}")

**OUTPUT :**

1, 4, 9, 16, 25

Squares: [1, 4, 9, 16, 25]

**FILTERS :**

filter() - The filter() function in Python filters elements from an iterable based on a given condition or function and returns a new iterable with the filtered elements.

The syntax for the filter() is as follows: filter(function, iterable)

Here also, the first argument passed to the filter function is itself a function, and the second argument passed is an iterable (sequence of elements) such as a list, tuple, set, string, etc.

**Example 1 - usage of the filter():**

You are given a list of integers and should filter the even numbers from the list.

# Using filter() to filter even numbers from a list

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

# The filter function filters the even numbers from the data

# and returns a filter object (an iterable)

evens = filter(lambda x: x % 2 == 0, data)

# Iterating the values of evens

for i in evens:

print(i, end=" ")

# We can convert the filter object into a list as follows:

evens = list(filter(lambda x: x % 2 == 0, data))

# Printing the evens list

print(f"Evens = {evens}")

**OUTPUT:**

2 4

Evens = [2, 4]