

**DATA SCIENCE**

ASSIGNMENT - 1

NAME : M.VASEEKARAN

REGISTER NO : 121012012773

DEGREE : B.TECH

BRANCH : CSE –3rd YEAR

SPECIALIZATION : DATA SCIENCE

SUBJECT : APPLIED AI

SUB.CODE : XCSHD3

# 1.Python Lambda Functions

Python Lambda Functions are anonymous function means that the function is without a name. As we already know that the def keyword is used to define a normal function in Python. Similarly, the lambda keyword is used to define an anonymous function in [Python](https://www.geeksforgeeks.org/python-programming-language/).

Python Lambda Function Syntax

Syntax: lambda arguments : 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.

## **Python Lambda Function Example**

In the example, we defined a lambda function(**upper**) to convert a string to its upper case using [upper()](https://www.geeksforgeeks.org/python-string-upper/).

str1 = 'GeeksforGeeks'

upper = lambda string: string.upper()

print(upper(str1))

## Use of Lambda Function in Python

Let’s see some of the practical uses of the Python lambda function.

### Condition Checking Using Python lambda function

Here, the format\_numric calls the lambda function, and the num is passed as a parameter to perform operations.

format\_numeric = lambda num: f"{num:e}" if isinstance(num, int) else f"{num:,.2f}"

print("Int formatting:", format\_numeric(1000000))

print("float formatting:", format\_numeric(999999.789541235))

**Output:**

Int formatting: 1.000000e+06

float formatting: 999,999.79

### Difference Between Lambda functions and def defined function

def cube(y):

return y\*y\*y

lambda\_cube = lambda y: y\*y\*y

# using function defined

# using def keyword

print("Using function defined with `def` keyword, cube:", cube(5))

# using the lambda function

print("Using lambda function, cube:", lambda\_cube(5))

**Output:**

Using function defined with `def` keyword, cube: 125

Using lambda function, cube: 125

As we can see in the above example, both the **cube()** function and **lambda\_cube()** function behave the same and as intended. Let’s analyze the above example a bit more:

| **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. |

# 2. map() function & filter() function

In Python, the `map()` and `filter()` functions are built-in higher-order functions that are often used to process and manipulate iterable data structures like lists, tuples, and more. They are both functional programming tools that take a function as an argument and apply it to elements of an iterable, but they serve different purposes:

1. `map(function, iterable)`:

- The `map()` function applies a given function to each item in the iterable (e.g., a list) and returns an iterator (generally a map object in Python 3) that contains the results.

- It transforms each item in the iterable by applying the specified function to it.

- The length of the resulting map object will be the same as the length of the input iterable.

Example:

```python

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

squared = map(lambda x: x \*\* 2, numbers)

# The 'squared' variable now contains a map object.

# You can convert it to a list to see the results.

squared\_list = list(squared)

# squared\_list is now [1, 4, 9, 16, 25]

```

2. `filter(function, iterable)`:

- The `filter()` function applies a given function to each item in the iterable and returns an iterator (generally a filter object in Python 3) that contains only the items for which the function returns `True`.

- It is used for filtering elements from an iterable based on a condition specified by the function.

- The length of the resulting filter object may be less than or equal to the length of the input iterable.

Example:

```python

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

even\_numbers = filter(lambda x: x % 2 == 0, numbers)

# The 'even\_numbers' variable now contains a filter object.

# You can convert it to a list to see the filtered results.

even\_numbers\_list = list(even\_numbers)

# even\_numbers\_list is now [2, 4, 6, 8, 10]

```

Both `map()` and `filter()` can be used with functions defined using `lambda` expressions (anonymous functions) or regular named functions. They are useful for simplifying and making code more concise when working with iterable data and applying transformations or filters to elements.

# 3. Iterators and generators in python

Iterators and generators are both Python constructs that allow you to work with sequences of data, but they serve different purposes and have distinct implementations.

## Iterators:

An iterator is an object that implements the Python iterator protocol, which consists of two methods: `\_\_iter\_\_()` and `\_\_next\_\_()`. Iterators are used to represent a stream of data, and they allow you to iterate through a collection of items one at a time without loading the entire collection into memory.

- `\_\_iter\_\_()` method returns the iterator object itself. It is called when you use the `iter()` function on an iterable object.

- `\_\_next\_\_()` method returns the next item from the iterator. It raises the `StopIteration` exception when there are no more items to return.

Here's an example of an iterator:

```python

class MyIterator:

def \_\_init\_\_(self, start, end):

self.current = start

self.end = end

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

return self

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

if self.current < self.end:

self.current += 1

return self.current - 1

else:

raise StopIteration

# Using the iterator

my\_iter = MyIterator(0, 5)

for num in my\_iter:

print(num)

```

## Generators:

Generators are a more concise and user-friendly way to create iterators in Python. They use the `yield` keyword within a function to define a generator. When a function contains a `yield` statement, it becomes a generator function. When the function is called, it returns a generator object, and execution of the function is paused at the `yield` statement. The state of the function is saved, allowing it to resume execution from where it left off when the next value is requested.

Here's an example of a generator function:

```python

def my\_generator(start, end):

current = start

while current < end:

yield current

current += 1

# Using the generator

gen = my\_generator(0, 5)

for num in gen:

print(num)

```

Generators are often more memory-efficient than creating an entire list of values upfront, as they produce values on-the-fly as needed.

In summary, iterators are a lower-level mechanism for creating objects that can be iterated over, while generators provide a more user-friendly and efficient way to create iterators using the `yield` keyword within functions. Both iterators and generators are essential tools for working with sequences of data in a memory-efficient manner in Python.

# 4. Modules and Packages

In Python, modules and packages are organizational tools that help you structure and manage your code in a more organized and modular way. They allow you to group related functions, classes, and variables into reusable units, making your code more maintainable and readable.

## Modules:

A module is a single Python file that contains Python code, including functions, classes, and variables. Modules are used to organize related code into separate files to improve code organization and reusability. You can import and use functions, classes, and variables defined in a module in other Python scripts.

Here's how you create and use a module:

1. Create a Python file with a `.py` extension, e.g., `my\_module.py`.

2. Define functions, classes, or variables within the module file.

3. In another Python script, import the module using the `import` statement.

Example module (`my\_module.py`):

```python

# my\_module.py

def greet(name):

return f"Hello, {name}!"

my\_variable = 42

```

Using the module in another script:

```python

import my\_module

print(my\_module.greet("Alice")) # Output: "Hello, Alice!"

print(my\_module.my\_variable) # Output: 42

```

## Packages:

A package is a directory that contains a special file called `\_\_init\_\_.py`, which can be empty or contain initialization code. Packages allow you to organize related modules into a hierarchical structure. This hierarchical structure can have sub-packages, allowing you to create a complex and organized project directory structure.

Here's how you create and use a package:

1. Create a directory (folder) with an `\_\_init\_\_.py` file. This directory will be your package, and you can give it a meaningful name.

2. Inside the package directory, create one or more module files (`.py` files).

3. To use modules within a package, import them using dot notation.

Example package structure:

```

my\_package/

\_\_init\_\_.py

module1.py

module2.py

```

Using modules from a package:

```python

# Importing modules from the package

from my\_package import module1, module2

print(module1.some\_function())

print(module2.some\_variable)

```

Packages are particularly useful for organizing large projects into smaller, manageable components and avoiding naming conflicts. They also facilitate the distribution and sharing of code libraries through Python's packaging ecosystem, such as PyPI (Python Package Index).

In summary, modules and packages in Python are essential tools for code organization, reusability, and maintaining a structured project directory. Modules are single Python files containing code, while packages are directories containing modules and optional `\_\_init\_\_.py` files. Both modules and packages help you manage and structure your Python code effectively.

# 5.MATRIX OPERATIONS

Matrix operations in Python can be performed using various libraries, with NumPy being the most popular and widely used for numerical operations involving matrices and arrays. Here, I'll show you how to perform common matrix operations using NumPy:

## 1. Matrix Creation:

To create matrices, you can use NumPy arrays. NumPy arrays can be created from Python lists or using NumPy functions.

```python

import numpy as np

# Creating matrices from lists

matrix1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Creating matrices using NumPy functions

matrix2 = np.zeros((3, 3)) # 3x3 matrix filled with zeros

matrix3 = np.ones((3, 3)) # 3x3 matrix filled with ones

matrix4 = np.eye(3) # 3x3 identity matrix

```

## 2. Matrix Addition and Subtraction:

You can add and subtract matrices of the same shape element-wise.

```python

result\_add = matrix1 + matrix2

result\_sub = matrix1 - matrix2

```

## 3. Matrix Multiplication:

Matrix multiplication can be done using the `dot()` function or the `@` operator.

```python

result\_mul = np.dot(matrix1, matrix2)

# or

result\_mul = matrix1 @ matrix2

```

## 4. Matrix Transposition:

To find the transpose of a matrix, use the `T` attribute or the `transpose()` function.

```python

matrix\_transpose = matrix1.T

# or

matrix\_transpose = np.transpose(matrix1)

```

## 5. Matrix Inversion:

To find the inverse of a matrix, you can use `numpy.linalg.inv()`.

```python

matrix\_inverse = np.linalg.inv(matrix1)

```

## 6. Determinant of a Matrix:

To find the determinant of a matrix, you can use `numpy.linalg.det()`.

```python

determinant = np.linalg.det(matrix1)

```

## 7. Eigenvalues and Eigenvectors:

To find the eigenvalues and eigenvectors of a matrix, use `numpy.linalg.eig()`.

```python

eigenvalues, eigenvectors = np.linalg.eig(matrix1)

```

## 8. Matrix Rank:

To find the rank of a matrix, use `numpy.linalg.matrix\_rank()`.

```python

rank = np.linalg.matrix\_rank(matrix1)

```

## 9. Singular Value Decomposition (SVD):

SVD is available through `numpy.linalg.svd()`.

```python

U, S, V = np.linalg.svd(matrix1)

```

## 10. Element-wise Operations:

NumPy allows you to perform element-wise operations easily.

```python

elementwise\_product = matrix1 \* matrix2 # Element-wise multiplication

elementwise\_power = np.power(matrix1, 2) # Element-wise exponentiation

```

These are some of the fundamental matrix operations you can perform using NumPy in Python. NumPy provides a powerful and efficient framework for working with matrices and arrays, making it a standard choice for scientific and numerical computing tasks.