PERIYAR
MANIAMMAI
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University)
Established Under Sec. 3 of UGC Act, 1956 • NAAC Accredited
think • innovate • transform

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME	:	SIVA PRIYAN.K
REG NO	:	121012012768
SUBJECT	:	DATA SCIENCE
SUBJECT CODE	:	XCSHD03
DEPARTMENT	:	B.TECH(CSE)

LAMDA FUNCTION

In Python, a lambda function is a small, anonymous function defined using the `lambda` keyword. Lambda functions are also known as anonymous functions or lambda expressions. They can take any number of arguments but can only have one expression.

Here's the basic syntax of a lambda function:

```python
Lambda arguments: expression
```

For example, you can create a lambda function that calculates the square of a number:

```python
Square = lambda x: x**2

Result = square(5)  # This will set ‘result’ to 25
```

Lambda functions are often used in situations where you need a simple, short function for a specific task, like sorting or filtering a list. For example, sorting a list of tuples based on the second element:

```python
Data = [(1, 3), (4, 2), (2, 8)]

Sorted_data = sorted(data, key=lambda x: x[1])  # Sort by the second element

# ‘sorted_data’ will be [(4, 2), (1, 3), (2, 8)]
```

Lambda functions are concise and useful for one-off operations, but for more complex functions, it’s better to define a regular named function.

MAP AND FILTER

Map and filter are two built-in functions in Python used for processing sequences like lists, tuples, or other iterable objects.

1. **Map**: The `map` function applies a given function to each item in an iterable (e.g., a list) and returns an iterator with the results.

~~~
```python

def square(x):return x ** 2  numbers = [1, 2, 3, 4, 5]

squared_numbers = map(square, numbers)

# Convert the iterator to a list or another iterable

squared_numbers_list = list(squared_numbers)
~~~

In this example, `map` applies the `square` function to each element in the `numbers` list and returns an iterator with the squared values.

2. **Filter**: The `filter` function is used to filter elements from an iterable based on a given function (usually a lambda function) that returns `True` or `False` for each element.

~~~
```python   def is_even(x):  return x % 2 == 0 numbers = [1, 2, 3, 4, 5, 6]even_numbers = filter(is_even, numbers)

# Convert the iterator to a list or another iterable

even_numbers_list = list(even_numbers)```
~~~

In this example, `filter` keeps only the elements for which the `is_even` function returns `True`, resulting in a list of even numbers.

Both `map` and `filter` return iterators, which can be converted to lists or used directly in a `for` loop or other iterable processing operations. They are handy for applying functions to elements in a collection or selectively extracting elements that meet certain criteria.

ITERATORS AND GENERATORS

Iterators and generators are both essential concepts in Python for working with sequences of data. They allow you to iterate over items in a collection or produce values on-the-fly. However, they serve slightly different purposes and have different implementations.

1. **Iterators**:

An iterator is an object that implements two methods, `__iter__()` and `__next__()`, allowing you to loop through a collection of items.

- It maintains the state of the iteration, remembering the current position within the collection.

- You can create custom iterators by defining classes that implement these methods.
- Python has built-in functions like `iter()` and `next()` that are used to work with iterators.

Example of a custom 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:

            Result = self.current

            Self.current += 1

            Return result

        Raise StopIteration
```

```
  My_iterator = MyIterator(1, 5)

  For num in my_iterator:

     Print(num)

```

2. **Generators**:

- A generator is a function that uses the `yield` keyword to produce a sequence of values on-the-fly.

- Unlike iterators, generators don't store the entire sequence in memory, making them memory-efficient for large data sets.

- When a generator function is called, it returns an iterator automatically.

- You can use a `for` loop to iterate over the values produced by a generator.

Example of a generator function:

```python
```

```
Def my_generator(start, end):
    Current = start
    While current < end:
        Yield current
        Current += 1

For num in my_generator(1, 5):
    Print(num)
```

In this example, the `my_generator` function generates values from `start` to `end` and yields them one at a time as you iterate over it.

In summary, iterators are more general and can be used to create custom iteration behavior, while generators are a simpler and more concise way to create iterators, especially for producing values on-the-fly without precomputing the entire sequence. Generators are commonly used in Python for tasks like reading large files or generating an infinite sequence of values.

MODULES AND PACKAGES

In Python, modules and packages are essential organizational units for structuring and managing code in larger projects. They help you keep your code organized, modular, and reusable.

1. Modules:

- A module is a single Python file that contains Python code. It can include functions, classes, and variables.

- Modules allow you to organize your code into separate files, making it easier to manage and maintain.

- You can create your own modules by creating a `.py` file and placing Python code inside it.

- To use functions or classes from a module in your code, you typically import the module using the `import` statement.

Example of using a module:

```python

# mymodule.py

Def greet(name):

    Return f"Hello, {name}!"



# main.py

Import mymodule
```

```
Message = mymodule.greet("Alice")

Print(message)
```

In this example, `mymodule` is a custom module that contains the `greet` function, which is used in the `main.py` script.

***2. Packages:**

- A package is a collection of related Python modules organized in directories.

- Packages allow you to create a hierarchy of modules and sub-packages, making it suitable for organizing code in larger projects.

- To create a package, you need to create a directory with an `__init__.py` file (which can be empty) and place module files inside it.

- You can import modules from a package using dot notation, specifying the package and module names.

Example of using a package:

```
My_package/

  __init__.py

  Module1.py

  Module2.py
```

~~~~
```

```python

# module1.py

Def func1():

   Return “Function 1 from module 1”

# module2.py

Def func2():

   Return “Function 2 from module 2”

# main.py

Import my_package.module1

Import my_package.module2
~~~~

~~~
Result1 = my_package.module1.func1()

Result2 = my_package.module2.func2()

Print(result1)

Print(result2)
~~~

```In this example, `my_package` is a package containing two modules, `module1` and `module2`. Both modules are imported and used in the `main.py` script.

Modules and packages are fundamental to structuring Python code effectively, promoting code reusability, and maintaining a clean and organized project structure. They also facilitate collaboration when multiple developers work on a project.

MATRIX OPERATION

Matrix operations in Python are commonly performed using the NumPy library, which provides efficient and powerful tools for working with multi-dimensional arrays, including matrices. Below are some common matrix operations using NumPy:

1. **Matrix Creation**:

You can create matrices using NumPy arrays.

~~~
  ```python
 Import numpy as np
# Create a 2x3 matrix
   Matrix = np.array([[1, 2, 3], [4, 5, 6]])  ```
~~~

2. **Matrix Addition and Subtraction**:

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

~~~
   ```python
   Import numpy as np
Matrix1 = np.array([[1, 2], [3, 4]])
   Matrix2 = np.array([[5, 6], [7, 8]])
~~~
```

Result_add = matrix1 + matrix2

Result_subtract = matrix1 – matrix2 ```

3. **Matrix Multiplication (Dot Product)**:

You can perform matrix multiplication using the `dot` function or the `@` operator.
~~~python

Import numpy as np

Matrix1 = np.array([[1, 2], [3, 4]])

Matrix2 = np.array([[5, 6], [7, 8]])

Result_dot = np.dot(matrix1, matrix2)

# Alternatively

Result_dot_alternative = matrix1 @ matrix ```

4. **Scalar Multiplication**:

You can multiply a matrix by a scalar (single value).

```python

Import numpy as np

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

Scalar = 2

Result_scalar_mult = scalar * matrix ```

5. **Matrix Transposition**: You can transpose a matrix using the `T` attribute.

```python

Import numpy as np

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

Transposed_matrix = matrix.T``

`6. **Matrix Inversion**:

You can find the inverse of a square matrix using `np.linalg.inv()`.

```python

Import numpy as np

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

Inverse_matrix = np.linalg.inv(matrix)

~~~

These are just some of the basic matrix operations you can perform using NumPy in Python. NumPy also provides functions for more advanced linear algebra operations, eigenvalue decomposition, singular value decomposition, and more, making it a powerful tool for matrix manipulation and linear algebra computations.