Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 7

Hello everyone, welcome to the second lecture this week, Introduction to Python. Recall that in the earlier lecture we discussed the print statements, the assignment operation, and the escape sequences in print statements, as well as assignment operations and the two different data types, one is the integer and the other is the floating point. We also discussed the format specifiers. One was %d, which was for integers, %0.nf can be written as %0.3f, %0.4f, %0.7f, etc. There is also %(m.n).e such as %1.3e, %1.4e, %1.5e.
It is recommended to practice these commands yourself in Google Colab to ensure you understand the syntax and output.

Arithmetic Operations

We assign numbers to variables to perform arithmetic operations such as addition, subtraction, multiplication, and division. Assign variables a and b as $a = 4$, $b = 5$

Sum of the two numbers: $a + b$

Difference: $a - b$

Product: $a * b$

Ratio: $\frac{a}{b}$

Integer division is performed using:

a//b

Power operation, i.e., a raised to the power b, is written as:

$$a^{**}b$$

Note that the caret symbol ^ does not work in Python for power operations; use double star instead.

The modulo (remainder) operation is:

$$a\%b$$

Evaluating these for $a = 4$ and $b = 5$: Then,

$$a + b = 9,\ a - b = -1,$$

and

$$a * b = 20,\ \frac{a}{b} = 0.8.$$

Integer division gives:

$$a // b = 0$$

which is the integer part of the division result.

Power:

$$4^5 = 1024$$

Modulo

$$4\%5 = 4$$

Basic arithmetic operators in Python are:

$$+,\quad -,\quad *,\quad /,\quad //,\quad **,\quad \%$$

Example: Computing Average Marks

Suppose marks in five subjects are m_1, m_2, m_3, m_4, m_5:

$$72,\quad 48,\quad 63,\quad 86,\quad 91$$

The average is

$$\text{Average} = \frac{m_1+m_2+m_3+m_4+m_5}{5}$$

This computes to an average of 72.

Scalar vs. Array Operations

Processing many variables individually is inefficient. Instead, it is better to process them collectively as vectors or arrays. Scalar operations are performed on individual variables, while array or vector operations handle multiple variables simultaneously. For array operations, we use the NumPy library (Numerical Python).

Using NumPy

Import the NumPy library as:

```
import numpy as np
```

Assign a vector (one-dimensional array) to a variable v:

```
v = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Python indexing starts at 0.

$$v[0] = 1, \quad v[1] = 2, \quad \ldots, \quad v[9] = 10$$

To access the sixth element:

```
print( v [5])
#output
6
```

Comments in Python

Single-line comments use #:

```
# This is a comment
```

Multi-line comments use triple single quotes:

```
'''
This is a
multiline
comment '''
```

Array Slicing

Access elements from index 2 to 6 (7 not included):

```
print( v [2 :7 ])
#output
[3 4 5 6 7]
```

Access the last element using negative index:

```
print(v[-1])
#output
10
```

```
print(v[-4])
#output
7
```

Slicing with Steps

Access elements from index 2 to 6 with step 2:

```
print( v [2 :7 :2 ])
#output
[3 5 7]
```

Access elements using negative indices and steps:

```
print(v[-2:-8:-2])
#output
[9 7 5]
```

Accessing from an Index to End or Beginning

From index 4 to end:

```
print(v[4:])
#output
[5 6 7 8 9 10]
```

From beginning to index 4 (excluded):

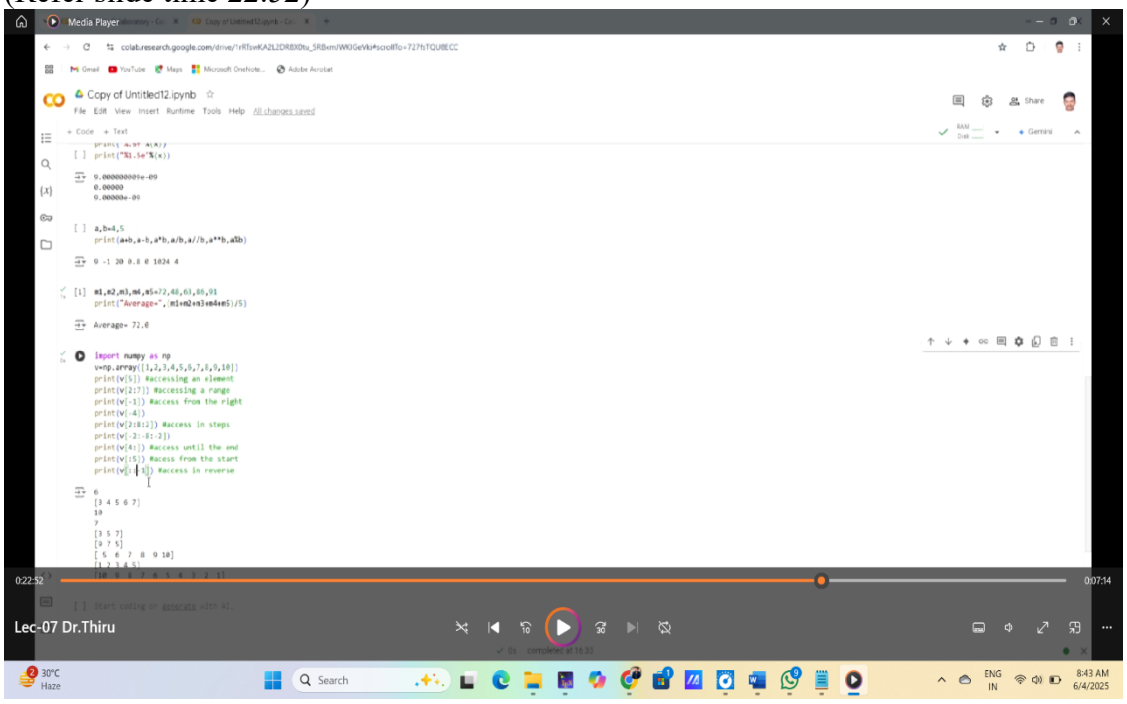
```
print(v[:5])
#output
[1 2 3 4 5]
```

Reversing an Array

Reverse the array:

```
print(v[::-1])
#output
[10 9 8 7 6 5 4  3 2 1]
```

(Refer slide time 22:52)



Two-Dimensional Arrays (Matrices)

We create a matrix using NumPy as follows:

```
import numpy as npm = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

Here, [1, 2, 3] is a one-dimensional array, [4, 5, 6] is another one- dimensional array, and [7, 8, 9] is yet another. Each of these arrays is enclosed in brackets, and these three arrays together are enclosed in another set of brackets, representing a two-dimensional array assigned to the variable m (used for matrix).

You can print m to see how it looks:

```
print(m)
#output
                [1 2 3]
                [4 5 6]
                [7 8 9]
```

You can access a specific element in the matrix using row and column indices.

For example

```
print(m[1, 2])
#output
 6
```

Here, you are accessing the first row and the second column (since indexing starts from 0). The result is:

Matrix Indexing Overview

Row indices: 0th row = [1, 2, 3], 1st row = [4, 5, 6], 2nd row =[7, 8, 9]

Column indices: 0th column = [1, 4, 7], 1st column = [2, 5, 8], 2nd column = [3, 6, 9]

Slicing Rows and Columns

You can access a range of elements using slicing. For example:

```
print(m[0:3:2, 1:3])
#output
                [2 3]
                [8 9]
```

This accesses the 0th and 2nd rows (due to step size 2) and columns 1 and 2.

Reverse Axis Access

You can also access elements in reverse. For example:

```
print(m[:-3:-1, :])

#output
                [7 8 9]
                [4 5 6]
```

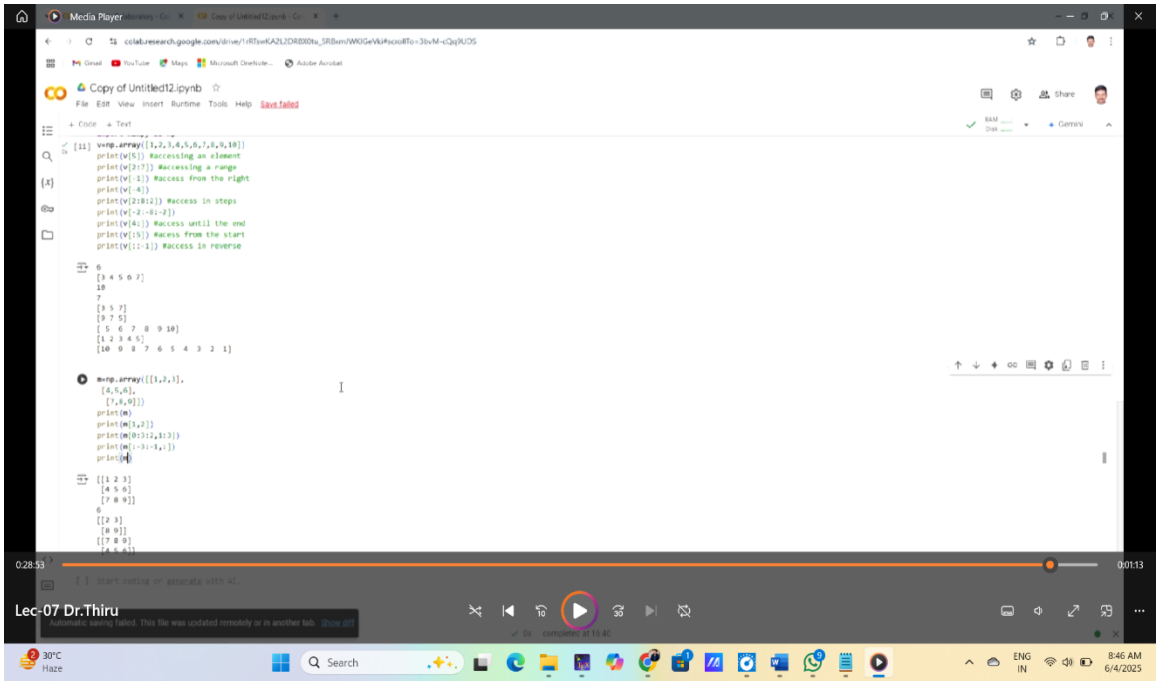
This starts from the last row and goes to the row before the last, in reverse. Columns are accessed completely using.

Reversing All Columns

To reverse all columns in each row:

```
print(m[:, ::-1])
#output
                [3 2 1]
                [6 5 4]
                [9 8 7]
```

(Refer slide time 22:52)



Conclusion

You can perform various slicing operations in two-dimensional arrays such as accessing ranges, reversing rows or columns, stepping through elements, and accessing all elements from a certain index to the end or start. These operations are very flexible and useful for array manipulations in Python.

We will continue with array operations in the next lecture. Thank you