Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

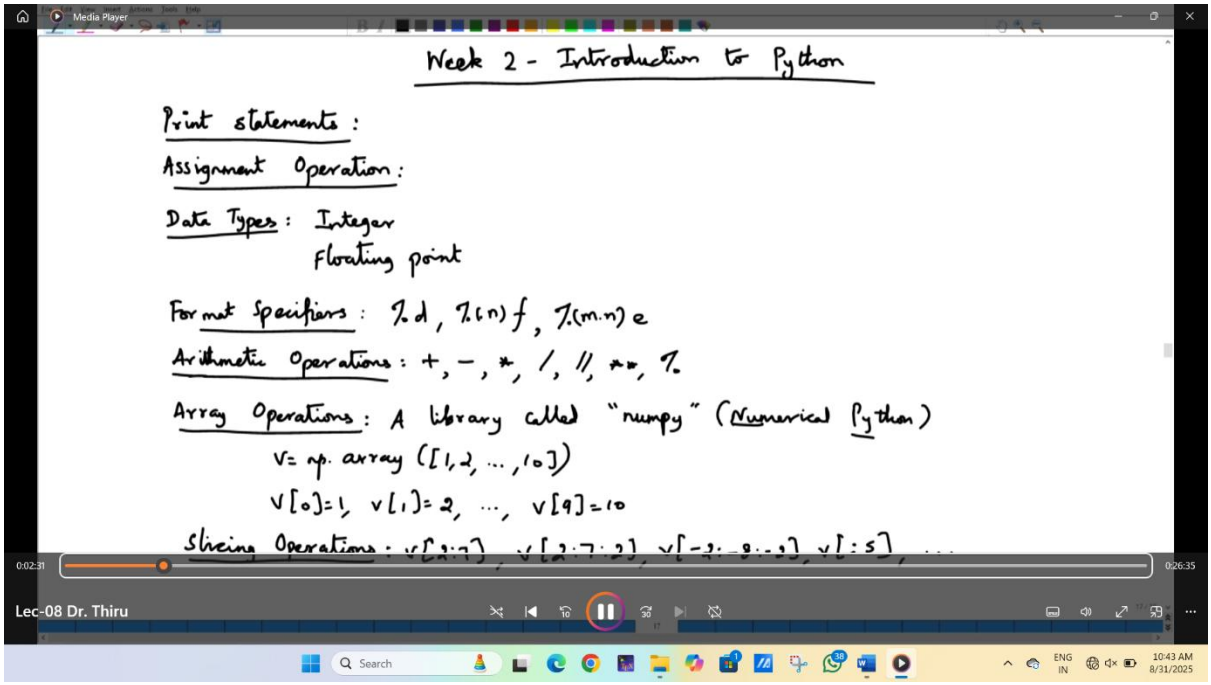
Lecture 8

Hello everyone, this is the third lecture of this particular week on the introduction to Python. You can recall that in the first two lectures, we looked at:

- Print statements, escape sequences, and format specifiers.
- The assignment operation for a scalar.
- The different data types a variable can take (e.g., integer, floating point).
- Arithmetic operations: addition (+), multiplication (*), subtraction (-), division (/), integer division (//), power operation (**), and modulo operation (%).
- Array operations using the numpy (Numerical Python) library.
- Slicing operations for accessing elements of an array (e.g., v[2:7], v[2:7:2], v[-4:-2]).

These slicing operations apply to both one-dimensional and two-dimensional arrays.

(Refer slide time 2:31)



Today, we will start with two other important operations on arrays: insertions and deletions.

1. Insertion and Deletion in 1D Arrays

First, import the NumPy library.

Code:
import numpy as np

Consider a standard array:

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

Output:
[1 2 3 4 5 6 7 8 9 10]

Inserting a Single Element

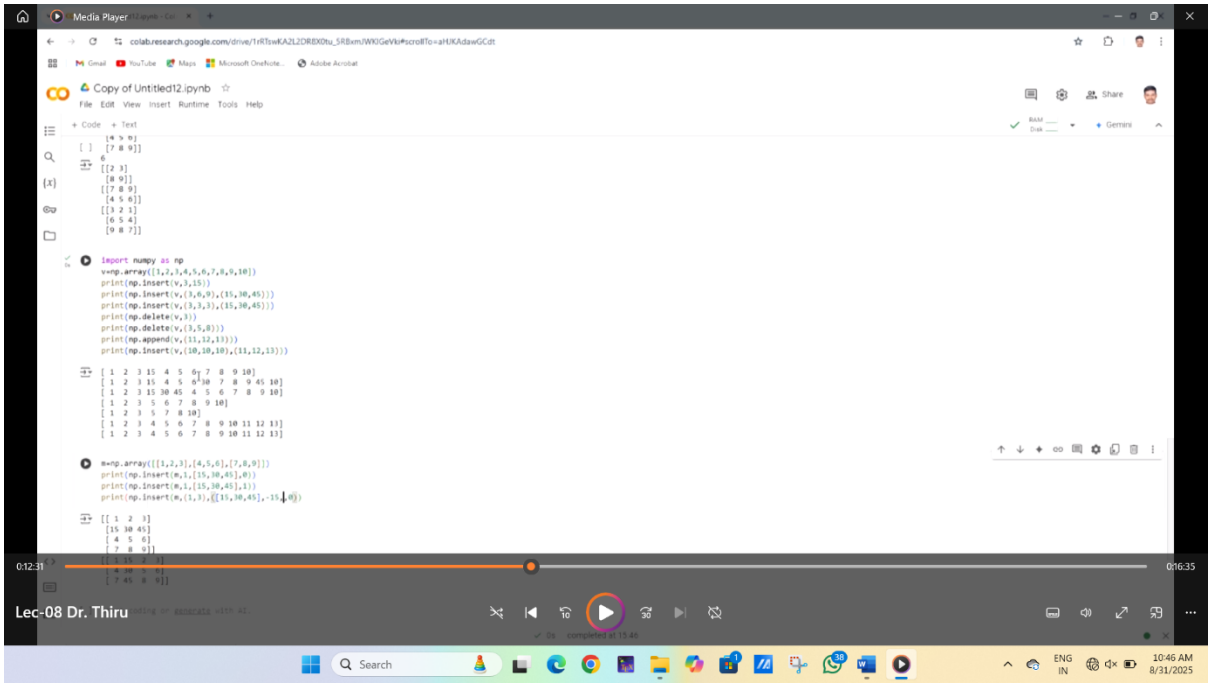
Suppose we want to insert the number 15 between 3 and 4.

Code:
new_v = np.insert(v, 3, 15)
print(new_v)

Output:
[1 2 3 15 4 5 6 7 8 9 10]

The index 3 means "insert before the element at index 3" (which is 4). Thus, the new element is placed between v[2] (value 3) and v[3] (value 4).

(Refer slide time 12:31)



Inserting Multiple Elements

Suppose we want to insert:

- 15 between 3 and 4 (before index 3)
- 30 between 6 and 7 (before index 6)
- 45 between 9 and 10 (before index 9)

Code:
```
new_v = np.insert(v, [3, 6, 9], [15, 30, 45])
print(new_v)
```

Output:
```
[ 1 2 3 15 4 5 6 30 7 8 9 45 10]
```

To insert multiple elements at the same position:

Code:
```
new_v = np.insert(v, 3, [15, 30, 45])
print(new_v)
```

Output:
```
[ 1 2 3 15 30 45 4 5 6 7 8 9 10]
```

Deleting Elements

To delete the element at index 3 (value 4):

Code:
```
new_v = np.delete(v, 3)
print(new_v)
```

Output:
```
[ 1 2 3 5 6 7 8 9 10]
```

To delete multiple elements at indices 3, 5, and 8 (values 4, 6, 9):

Code:
```
new_v = np.delete(v, [3, 5, 8])
print(new_v)
```

Output:
```
[ 1 2 3 5 7 8 10]
```

Appending Elements

The append operation inserts elements at the end of an array. To append 11, 12, 13:

Code:
```
new_v = np.append(v, [11, 12, 13])
print(new_v)
```

Output:
[1 2 3 4 5 6 7 8 9 10 11 12 13]

This is equivalent to np.insert(v, 10, [11, 12, 13]) (inserting before the non-existent index 10, which places them at the end).

2. Insertion and Deletion in 2D Arrays

Consider a 2D array (matrix):

Code:
```
m = np.array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(m)
```

Output:
```
[[1 2 3]
[4 5 6]
[7 8 9]]
```

Inserting a Row

To insert a new row [15, 30, 45] before row index 1 (i.e., between row 0 and row 1), we use the axis=0 parameter.

Code:
```
new_m = np.insert(m, 1, [15, 30, 45], axis=0)
print(new_m)
```

Output:
```
[[ 1 2 3]
[15 30 45]
[ 4 5 6]
[ 7 8 9]]
```

Inserting a Column

To insert a new column [15, 30, 45] before column index 1 (i.e., between column 0 and column 1), we use the axis=1 parameter.

Code:
```
new_m = np.insert(m, 1, [15, 30, 45], axis=1)
print(new_m)
```

Output:
```
[[ 1 15 2 3]
[ 4 30 5 6]
[ 7 45 8 9]]
```

Inserting Multiple Rows

To insert:

- Row R_1 = [15, 30, 45] before row index 1
- Row R_2 = [-15, -30, -45] before row index 3 (i.e., at the end)

Code:

```
new_m = np.insert(m, [1, 3], [[15, 30, 45], [-15, -30, -45]], axis=0)
print(new_m)
```

Output:

```
[[ 1 2 3]
[15 30 45]
[ 4 5 6]
[ 7 8 9]
[-15 -30 -45]]
```

Deleting a Column

To delete column 1 (values 2, 5, 8):

Code:

```
new_m = np.delete(m, 1, axis=1)
print(new_m)
```

Output:

```
[[1 3]
[4 6]
[7 9]]
```

Appending Rows and Columns

Appending requires the new data to be a 2D array of the correct shape.

Appending a Row:
Append a single row [15, 30, 45].

Code:

Correct: 2D array with one row

```
new_m = np.append(m, [[15, 30, 45]], axis=0)
print(new_m)
```

Output:

```
[[ 1 2 3]
[ 4 5 6]
[ 7 8 9]
[15 30 45]]
```

Appending Columns:
Append two columns. The new data must be a 2D array with the same number of rows, structured as columns.

Code:

Data for two new columns

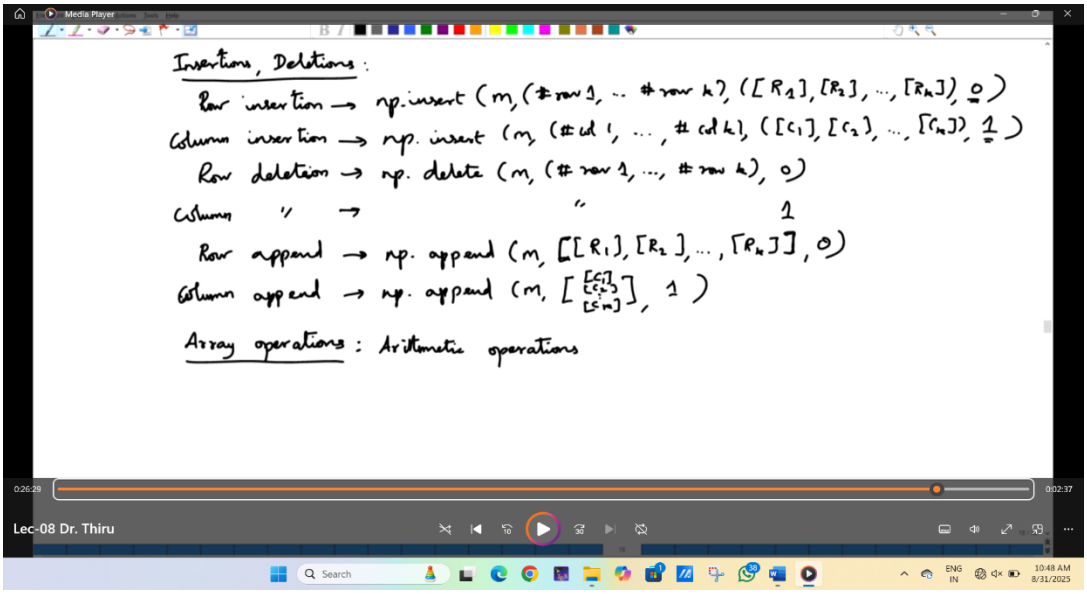
```
new_cols = np.array([[15, -15],
[30, -30],
[45, -45]]) # A 3x2 matrix

new_m = np.append(m, new_cols, axis=1)
print(new_m)
```

Output:
```
[[ 1 2 3 15 -15]
[ 4 5 6 30 -30]
[ 7 8 9 45 -45]]
```

Summary of Commands:

- **Row Insertion:** np.insert(m, [#dx$_1$, #dx$_2$, ..., #dx$_k$]), ([R$_1$], R$_2$], ..., [R$_k$], axis=0)
- **Column Insertion:** np.insert(m, [#dx$_1$, #dx$_2$, ..., #dx$_k$]), ([C$_1$], C$_2$], ..., [C$_k$], axis=1)
- **Row Deletion:** np.delete(m, [#dx$_1$, #dx$_2$, ..., #dx$_k$], axis=0)
- **Column Deletion:** np.delete(m, [#dx$_1$, #dx$_2$, ..., #dx$_k$], axis=1)
- **Row Append:** np.append(m, [[R$_1$], [R$_2$], ..., [R$_k$]], axis=0) (Must be a 2D array)
- **Column Append:** np.append(m, [[C$_1$, C$_2$, ..., C$_k$]], axis=1) (Must be a 2D array)

(Slide Refer time 26:29)



3. Array Arithmetic Operations

Consider two 1D arrays:

Code:

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

print("v + w =", v + w)
print("v - w =", v - w)
print("v * w =", v * w) # Element-wise multiplication
print("v / w =", v / w) # Element-wise division
print("v // w =", v // w) # Integer division
print("v ** w =", v ** w) # Power operation
print("v % w =", v % w) # Modulo operation
```

Sample Output:

```
v + w = [11 11 11 11 11 11 11 11 11 11]
v - w = [-9 -7 -5 -3 -1 1 3 5 7 9]
v * w = [10 18 24 28 30 30 28 24 18 10]
...
```

These operations are performed element-wise.

For 2D arrays, the same rules apply: +, -, *, /, etc., are all element-wise operations.

Matrix Multiplication

For two matrices m and n, the operator * performs element-wise multiplication.

Code:

```
m = np.array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
n = m.copy() # n is identical to m

print("Element-wise multiplication (m * n):")
print(m * n)
```

Output:

```
[[ 1 4 9]
[16 25 36]
[49 64 81]]
```

To perform true matrix multiplication, use the @ operator or np.matmul.

Code:

```
print("Matrix multiplication (m @ n):")
print(m @ n)
```

Output:

```
[[ 30 36 42]
[ 66 81 96]
[102 126 150]]
```

Remember: * is element-wise, @ is for matrix multiplication.

4. Matrix Properties

Shape of an Array

Use .shape to find the dimensions.

Code:

```
print("Shape of m (3x3):", m.shape)
print("Shape of v (10,):", v.shape)
```

Output:

```
Shape of m (3x3): (3, 3)
Shape of v (10,): (10,)
```

Number of Dimensions

Use .ndim to find the number of axes.

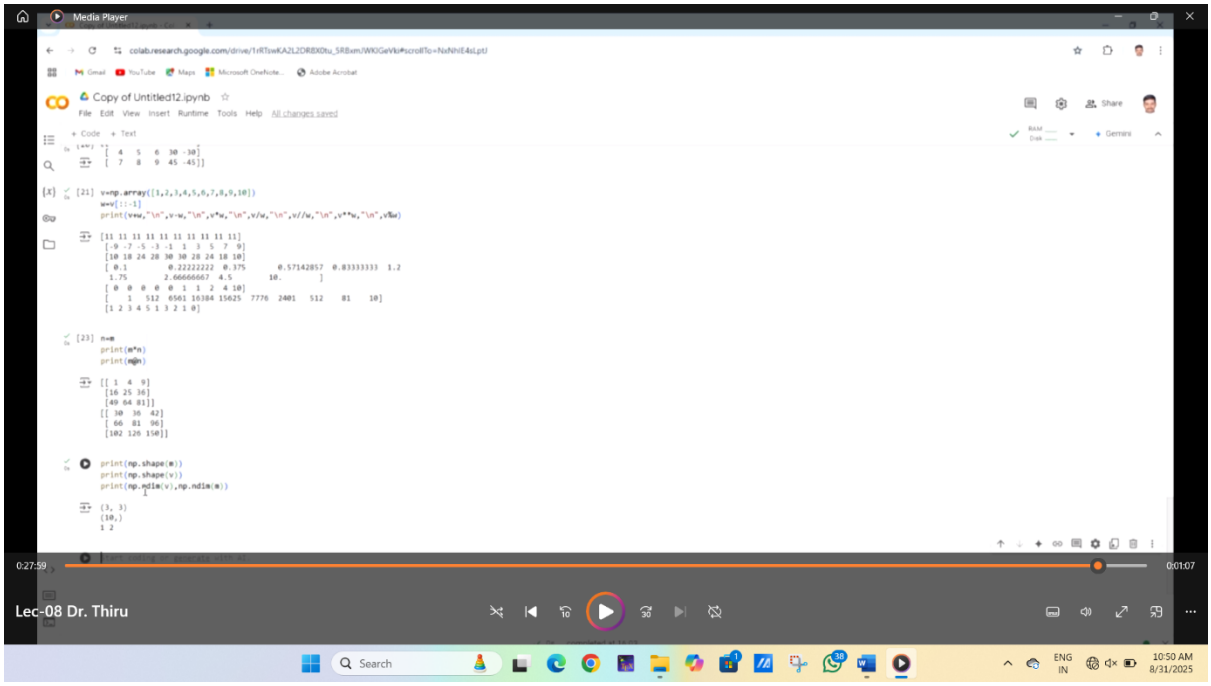
Code:

```
print("Number of dimensions of v:", v.ndim)
print("Number of dimensions of m:", m.ndim)
```

Output:

```
Number of dimensions of v: 1
Number of dimensions of m: 2
```

(Refer slide time 27:59)



We will continue with more array operations in the next lecture.

Thank you.