Week_4Lecture2026

January 31, 2026

1 Week #4 - Numpy library for Python

1.0.1 During this week, you will learn Numpy in Python. Why is Numpy important? It allows you to deal with complex mathematics. Before we can use any Numpy functions, we need to call the function in a library call. After importing Numpy we can access all the functions.

1.0.2 Topics You will learn

- Arrays
- arange
- random integers
- Uniform
- random real numbers
- normal random numbers
- Maxtrix
- shape/reshape matrix
- Arithmetics of arrays
- Adding arrays
- Array methods
- array indexing and Slicing

2 Call Numpy library

Before using Nuumpy data types you must call the Numpy library. The way to access Numpy library. When ever you refer to numpy use np.

```
import numpy as np
```

2.0.1 Run the code.

[17]:
```
import numpy as np
```

3 Why NumPy? (The Business Case)

3.0.1 In Excel, if you want to multiply a column of prices by a column of quantities, you drag a formula down. In standard Python, you would need a "for loop." NumPy allows Vectorization, which lets us perform math on entire arrays at once—just like a single formula in an Excel cell applied to a whole range.

3.0.2 Key Concept: NumPy arrays are homogeneous (all data must be the same type, usually numbers), making them much faster than standard Python lists.

3.0.3 Creating Your First Arrays

Students should practice moving from standard Python lists to NumPy.

```
import numpy as np

# **1D Array (like a single row or column in Excel) - A "Vector"**  one row and three columns
prices = np.array([19.99, 25.50, 15.00])

# **2D Array (like a full spreadsheet) - A "Matrix"**   three rows and three columns
# Rows = Products, Columns = [Price, Cost, Inventory]
data = np.array([
    [19.99, 10.00, 50],
    [25.50, 12.00, 30],
    [15.00, 5.00, 100]
])
```

3.0.4 Run the code.

[18]:
```
import numpy as np

# 1D Array (like a single row or column in Excel) - A "Vector",  one row and␣
 ↪three columns.  1 x 3 elements
prices = np.array([19.99, 25.50, 15.00])

# 2D Array (like a full spreadsheet) - A "Matrix",   three rows and three␣
 ↪columns.  3x3 = 9 elements
# Rows = Products, Columns = [Price, Cost, Inventory]

data = np.array([
    [19.99, 10.00, 50],
    [25.50, 12.00, 30],
    [15.00, 5.00, 100]
])
```

4 3. Business Math: Vectorization & Broadcasting

4.0.1 This is where the precalculus background shines. Instead of manual calculation, we use element-wise operations.

4.0.2 Scenario: You want to apply a 10% tax to all prices.

Formula: $TaxedPrice = Price \times 1.10$

```
taxed_prices = prices * 1.10  # This is "Broadcasting" a single number across an array
```

[19]:
```
taxed_prices = np.round(prices * 1.10, 2)  # This is "Broadcasting" a single number across an array
taxed_prices
```

[19]:
```
array([21.99, 28.05, 16.5 ])
```

5 Slicing and Filtering

Business data is often messy. We use slicing to extract specific values like "Price", "Cost", or "Inventory".

Indexing: data[0, 0] Gets the price of the first item.

Slicing column: data[: , i] The index i equals the column you want to access; in this case, the entire Cost column. The first column index, 0, corresponds to the price. The second column index, 1, corresponds to the cost. The third column, which equals 2, is the inventory. **1st column data[: , 0]**.

Slicing: data[j : k] The index j equals the row you want to access, in this case, the entire row. The first row has index 0. The second index k determine where you want to stop gathering rows. jth row stopping at k-1 rows **1st row data[0 : 1]**

Boolean Masking: data[: , 2] < 40 Finding all products with low inventory below 40. **low_stock = data[data[:, 2] < 40]**

```
cost = data[:, 1]
profit = prices - cost

prices  # displays prices
# in another code cell
taxed_prices  # prices with taxes
```

[20]:
```
data
```

[20]:
```
array([[ 19.99,  10.  ,  50.  ],
       [ 25.5 ,  12.  ,  30.  ],
       [ 15.  ,   5.  , 100.  ]])
```

[21]:
```
data[:, 0] # 1st column
```

[21]:
```
array([19.99, 25.5 , 15.  ])
```

[22]:
```
data[:, 1]  # 2nd column
```

[22]:
```
array([10., 12.,  5.])
```

[23]:
```
data[:, 2]  # 3rd column
```

[23]:
```
array([ 50.,  30., 100.])
```

[24]:
```
data[0 :1 ]
```

[24]:
```
array([[19.99, 10.  , 50.  ]])
```

[25]:
```
cost = data[:, 1]
profit = prices - cost

profit  # displays prices
```

[25]:
```
array([ 9.99, 13.5 , 10.  ])
```

[26]:
```
# in another code cell
taxed_prices  # prices with taxes
```

[26]:
```
array([21.99, 28.05, 16.5 ])
```

[27]:
```
data[:, 0]  # first row
```

[27]:
```
array([19.99, 25.5 , 15.  ])
```

[28]:
```
# get inventory column
data[: , 2]
```

[28]:
```
array([ 50.,  30., 100.])
```

[29]:
```
low_stock = data[data[:, 2] < 40]
low_stock
```

[29]:
```
array([[25.5, 12. , 30. ]])
```

[30]:
```
data[0:1]  # First row
```

[30]:
```
array([[19.99, 10.  , 50.  ]])
```

[31]:
```
data[1: 2]  # Second row
```

[31]:
```
array([[25.5, 12. , 30. ]])
```

[32]:
```
data[2:3]  # Third row
```

[32]:
```
array([[ 15.,   5., 100.]])
```

6 Determine the size or shape of an Array

The size of the array is the number of elements in the array. The shape of the array is the number of rows and columns.

```
print("Number of elements: ", data.size)   # number of elements
print("Number of rows and columns: ", data.shape)  # number of rows and columns
```

[33]:
```
print("Number of elements: ", data.size)   # number of elements
print("Number of rows and columns: ", data.shape)  # number of rows and columns
```

```
Number of elements:  9
Number of rows and columns:  (3, 3)
```

7 arange()

arange is similar to the range function in Python. You can create an array of values from start, to end + 1.

```
A1 = np.arange(11)
A2 = np.arange(5, 11)
```

[34]:
```
A1 = np.arange(11)
A1
```

[34]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

[35]:
```
A2 = np.arange(5, 11)
A2
```

[35]:
```
array([ 5,  6,  7,  8,  9, 10])
```

8 Random Numbers

You may want to generate random numbers from 0 to 1.

```
np.random.rand()
```

[36]:
```
np.random.rand()
```

[36]:
```
0.5270618619114266
```

9 Random Numbers (rows and columns)

You may want to generate random numbers between 0 and 1 that fits into a matrix of (row, column)

```
np.random.rand(3, 5) # (rows, columns)
```

[37]:
```
np.random.rand(3, 5)
```

[37]:
```
array([[0.79271479, 0.38396621, 0.8331135 , 0.03719885, 0.08690188],
       [0.78585578, 0.44964238, 0.202725  , 0.09680042, 0.5774812 ],
       [0.12008124, 0.24389083, 0.91794674, 0.69579966, 0.92529777]])
```

10 Random Numbers - Integers (rows, columns)

You may want to generate integers between integers, either as one value or a matrix of values.

```
np.random.randint(1, 101) # random number between 1 and 100
np.random.randint(1, 101, (5,3))
```

[38]:
```
np.random.randint(1, 101)
```

[38]:
```
22
```

[39]:
```
np.random.randint(1, 101, (5,3))
```

[39]:
```
array([[57, 85, 76],
       [18,  4,  5],
       [87, 64, 66],
       [79, 75, 72],
       [83, 77, 20]])
```

11 Random Numbers - Normal

The normal distribution is very important in Statistics, so you can generate random numbers from a Normal distribution. “‘ # the random number from a Normal distribution mean = 75 and std = 5 np.random.normal(75, 5)

[40]:
```
np.random.normal(75, 5, (4, 5))
```

[40]:
```
array([[71.21726305, 80.81937103, 73.47829239, 77.72780332, 82.00283119],
       [77.96310416, 76.25062054, 74.55816873, 87.11153051, 72.65036388],
       [73.11478165, 76.2685694 , 71.9892267 , 69.05147113, 77.54666411],
       [79.11906143, 73.78926272, 79.02315089, 74.28073896, 67.11312945]])
```

12 Converting a list to an Array

You can use a tuple to assign the mean and standard deviation. Use a list comprehension to create a list and convert the list to an array.

```
mean, std = (75, 1.5)
l3 = [np.random.normal(mean, std) for x in np.arange(25)]
L3 = np.array(l3)  # convert the list to an array
L3
```

[41]:
```
mean, std = (75, 1.5)
l3 = [np.random.normal(mean, std) for x in np.arange(25)]
L3 = np.array(l3)
L3
```

[41]:
```
array([75.19342644, 74.10090291, 76.75389595, 75.15545685, 73.33188661,
       73.70481788, 72.4157888 , 74.09252912, 77.8321144 , 76.7486797 ,
       76.83156856, 74.77752219, 73.03358406, 76.47213105, 75.54317536,
       75.11301476, 75.23188117, 73.52558889, 76.80409912, 76.05128476,
       73.36353126, 75.17488867, 74.97945775, 74.64370407, 75.98694797])
```

13 Arithmetic with NumPy Arrays

You can perform arithmetic operations with an array. You can do all the order of operations (parentheses, raise to a power, multiplication, division, addition, or subtraction).

```
import numpy as np
A = np.array([31, 32, 33, 34, 35, 36, 37, 38, 39])
B = np.array([41, 42, 43, 44, 45, 46, 47, 48, 49])
A ** 3
A * 4
A / 3
A + 7
A - 6
(A + B) ** 2 + 3
A + 3 * B
2 * A - 4 * B
```

[42]:
```
import numpy as np
A = np.array([31, 32, 33, 34, 35, 36, 37, 38, 39])
B = np.array([41, 42, 43, 44, 45, 46, 47, 48, 49])
A ** 3
```

[42]:
```
array([29791, 32768, 35937, 39304, 42875, 46656, 50653, 54872, 59319])
```

[43]:
```
A * 4
```

[43]:
```
array([124, 128, 132, 136, 140, 144, 148, 152, 156])
```

[44]:
```
A + 7
```

[44]:
```
array([38, 39, 40, 41, 42, 43, 44, 45, 46])
```

[45]:
```
A - 6
```

[45]:
```
array([25, 26, 27, 28, 29, 30, 31, 32, 33])
```

[46]:
```
(A + B) ** 2 + 3
```

[46]:
```
array([5187, 5479, 5779, 6087, 6403, 6727, 7059, 7399, 7747])
```

[47]:
```
A + 3 * B
```

[47]:
```
array([154, 158, 162, 166, 170, 174, 178, 182, 186])
```

[48]:
```
2 * A - 4 * B
```

[48]:
```
array([-102, -104, -106, -108, -110, -112, -114, -116, -118])
```

[49]:
```
G = np.array([11, 12, 13, 14, 15, 16, 17, 18, 19])
A * G
```

[49]:
```
array([341, 384, 429, 476, 525, 576, 629, 684, 741])
```

[50]:
```
A - G
```

[50]:
```
array([20, 20, 20, 20, 20, 20, 20, 20, 20])
```

[51]:
```
A/G
```

[51]:
```
array([2.81818182, 2.66666667, 2.53846154, 2.42857143, 2.33333333,
       2.25      , 2.17647059, 2.11111111, 2.05263158])
```

[52]:
```
(A + 3*G) ** 2
```

[52]:
```
array([4096, 4624, 5184, 5776, 6400, 7056, 7744, 8464, 9216])
```

14 Numpy Statistical Methods

14.0.1 Statistical methods that the numpy array uses (vectors)

- max()
- argmax()
- min()
- argmin()
- mean()

- median()
- std()
- sum()
- count

[53]:
```
K = np.random.normal(65, 2.5, (20,1))
```

[54]:
```
K.mean().item()
```

[54]: 65.46804869234407

[55]:
```
# maximum value
L3.max().item()
```

[55]: 77.83211439514776

[56]:
```
# index of maximum value
L3.argmax().item()
```

[56]: 8

[57]:
```
# minimum of array
L3.min().item()
```

[57]: 72.41578880120036

[58]:
```
L3.argmin().item()
```

[58]: 6

[59]:
```
L3.mean().item()
```

[59]: 75.0744751311121

[60]:
```
L3.std().item()
```

[60]: 1.3710075207560533

[61]:
```
np.percentile(L3, 25).item()
```

[61]: 74.0925291150512

[62]:
```
np.percentile(L3, 75).item()
```

[62]: 76.0512847595619

[63]:
```
np.percentile(L3, 10).item()
```

[63]: 73.34454446674229

15 axis

Axises is a variable that determines whether you are applying the method to rows (axis = 1) or columns (axis = 0).

```
X = np.arange(0, 15)
X = X.reshape(3, 5)
X.mean(axis = 0)
X.sum(axis = 0)
X.mean(axis = 1)
X.sum(axis = 1)
```

[64]:
```
X = np.arange(0, 15)
X = X.reshape(3, 5)
X
```

[64]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

[65]:
```
Y = X.sum(axis = 1)
Y
```

[65]:
```
array([10, 35, 60])
```

[66]:
```
row_1_total = Y.item(0)
row_1_total
```

[66]:
```
10
```

[67]:
```
nm = X.sum(axis = 1).item(0)
nm
```

[67]:
```
10
```

[68]:
```
X.mean(axis = 0)  # average of columns
```

[68]:
```
array([5., 6., 7., 8., 9.])
```

[69]:
```
X.mean(axis = 1)  # average of rows
```

[69]:
```
array([ 2.,  7., 12.])
```

[70]:
```
X.std(axis = 0) # standard deviation of columns
```

[70]:
```
array([4.0824829, 4.0824829, 4.0824829, 4.0824829, 4.0824829])
```

[71]:
```
X.std(axis = 1) # standard deviation of rows
```

[71]:
```
array([1.41421356, 1.41421356, 1.41421356])
```

[72]:
```
np.percentile(X, [25, 75],  axis=1)
```

[72]:
```
array([[ 1.,  6., 11.],
       [ 3.,  8., 13.]])
```

[73]:
```
(X > 4).any()  # Are there any values in X greater than 4
```

[73]:
```
np.True_
```

[74]:
```
(X > -1).all()  # Are all the values in X greater than 4
```

[74]:
```
np.True_
```

[75]:
```
L4 = np.sort(L3)
L4
```

[75]:
```
array([72.4157888 , 73.03358406, 73.33188661, 73.36353126, 73.52558889,
       73.70481788, 74.09252912, 74.10090291, 74.64370407, 74.77752219,
       74.97945775, 75.11301476, 75.15545685, 75.17488867, 75.19342644,
       75.23188117, 75.54317536, 75.98694797, 76.05128476, 76.47213105,
       76.7486797 , 76.75389595, 76.80409912, 76.83156856, 77.8321144 ])
```

16 Conditional Logic

To create a new array based on the condition.

```
np.where(L4 > L4.mean(), 'higher', 'lower')
```

[76]:
```
np.where(L4 > L4.mean(), 'higher', 'lower')
```

[76]:
```
array(['lower', 'lower', 'lower', 'lower', 'lower', 'lower', 'lower',
       'lower', 'lower', 'lower', 'lower', 'higher', 'higher', 'higher',
       'higher', 'higher', 'higher', 'higher', 'higher', 'higher',
       'higher', 'higher', 'higher', 'higher', 'higher'], dtype='<U6')
```

[77]:
```
w1 = L4.mean() - 1 * L4.std()
w2 = L4.mean() - 2 * L4.std()
w3 = L4.mean() + 1 * L4.std()
w4 = L4.mean() + 2 * L4.std()
np.where(L4 < w2, "2 std below", (np.where(L4 < w1, "1 std below", (np.where(L4␣
 ↪> w4, "1 std above", "2 std above")))))
```

[77]:
```
array(['1 std below', '1 std below', '1 std below', '1 std below',
       '1 std below', '2 std above', '2 std above', '2 std above',
       '2 std above', '2 std above', '2 std above', '2 std above',
       '2 std above', '2 std above', '2 std above', '2 std above',
       '2 std above', '2 std above', '2 std above', '2 std above',
       '2 std above', '2 std above', '2 std above', '2 std above',
       '1 std above'], dtype='<U11')
```

[ ]: