# untitled1-1

August 31, 2024

## 1 Numpy

**.NumPy is a powerful library in Python used for numerical and scientific computing.**

**.Its core functionalities revolve around its central data structure, the ndarray, which is a multidimensional array object.**

**.Here's a detailed overview of NumPy's core functionalities:**

### 1.1 Import numpy

Import numpy with an alias 'np'

[166]:
```
import numpy as np
```

### 1.2 Step1:Exploring Numpy's core functionalities

**->Creating arrays**

[185]:
```
arr1 = np.array([1, 2, 3, 4]) #creating 1D-array from list
arr2= np.array((1, 2, 3, 4)) #creating 1D-array from tuple
print("arr_from_list",arr1)
print("arr_from_tuple",arr2)
```

```
arr_from_list [1 2 3 4]
arr_from_tuple [1 2 3 4]
```

[187]:
```
arr3 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # Create a 2D array (matrix)
print("2D Array:\n", arr3)
```

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

**->Performing basic operations**

[190]:
```
#Addition of two arrays
add= arr1 + arr2
print("addition:",add)
```

```
#subtraction of two arrays
sub = arr1 - arr2
print("subtraction:",sub)
#Multiplication of two arrays
mult = arr1 * arr2
print("multiplication:",mult)
#Division of two arrays
div= arr1 / arr2
print("division:",div)
```

```
addition: [2 4 6 8]
subtraction: [0 0 0 0]
multiplication: [ 1  4  9 16]
division: [1. 1. 1. 1.]
```

**->Array properties**

**1.Shape:The shape property returns a tuple representing the dimensions of the array.**

[195]:
```
array = np.array([[1, 2], [3, 4]])
print("shape:",array.shape)
```

```
shape: (2, 2)
```

**2.Number of Dimensions (ndim):The ndim property returns the number of dimensions (axes) of the array.**

[201]:
```
print("Number of dimensions:", array.ndim)
```

```
Number of dimensions: 2
```

**3.Size:The size property returns the total number of elements in the array.**

[204]:
```
print("Size:", array.size)
```

```
Size: 4
```

**4.Data Type (dtype):The dtype property returns the data type of the elements in the array.**

[207]:
```
array1 = np.array([1, 2, 3], dtype=np.float64)
print("d_type:",array1.dtype)
```

```
d_type: float64
```

**5.Item Size:The itemsize property returns the size (in bytes) of each element in the array.**

[210]:
```
print("Item size:", array.itemsize)
```

```
Item size: 4
```

### 1.3 Step2:Data Manipulation using Numpy

**->Array creation**

[110]:
```
# Create a 1D array
arr1 = np.array([10, 20, 30, 40, 50])
print("1D Array:", arr1)
```

```
1D Array: [10 20 30 40 50]
```

[112]:
```
# Create a 2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("2D Array:\n", arr2)
```

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

[114]:
```
# Create an array with a range of values
arr3= np.arange(5, 15, 2)
print("Array with a Range of Values:", arr3)
```

```
Array with a Range of Values: [ 5  7  9 11 13]
```

**->Indexing and Slicing**

[118]:
```
# Indexing in2D array
elmnt= arr2[1, 2]
print("Element at row 1, column 2:", elmnt)
```

```
Element at row 1, column 2: 6
```

[120]:
```
#Indexing in 1D array
elmnt1 = arr1[2]
print("Element at index 2 is:",elmnt1)
```

```
Element at index 2 is: 30
```

[122]:
```
# Slicing in 2D array
sliced_arr = arr2[1:, 1:]
print("Sliced Array:\n", sliced_arr)
```

```
Sliced Array:
 [[5 6]
 [8 9]]
```

[124]:
```
#Slicing in 1D array
sliced_arr2=arr1[1:2]
print("sliced Array:",sliced_arr2)
```

```
sliced Array: [20]
```

### ->Reshaping Arrays

[128]:
```python
#Reshape a 1D array into a 2D array using reshape.
reshaped_array = np.arange(12).reshape((3, 4))
print("Reshaped Array:\n", reshaped_array)
```

```
Reshaped Array:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

### ->Applying Mathematical Operations

[227]:
```python
# Arithmetic operations
print("Array1 + 5:", added_array)
```

```
Array1 + 5: [6. 7. 8.]
```

[229]:
```python
# Element-wise multiplication
print("Array2 * 2:\n", multiplied_array)
```

```
Array2 * 2:
 [[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

[233]:
```python
#Sum of elements in the 2D array
print("Sum of elements in arr np.sum(arr2):", sum_array2)
```

```
Sum of elements in arr np.sum(arr2): 45
```

[251]:
```python
# Mean of elements in the 2D array
mean_array2 = np.mean(arr2)
print("Mean of elements in Array2:", mean_array2)
```

```
Mean of elements in Array2: 2.5
```

[253]:
```python
#Transpose a 2D array using np.transpose
transpose_arr = np.transpose(arr2)
print("Transpose of Arr:\n", transpose_arr)
```

```
Transpose of Arr:
 [1 2 3 4]
```

## 1.4 Step3:Data Aggregation

[261]:
```
# Create a 2D array with random integers between 1 and 100
data = np.random.randint(1, 100, size=(5, 4))
print("Data:\n", data)
```

```
Data:
 [[33 66 10 58]
 [33 32 75 24]
 [36 76 56 29]
 [35  1  1 37]
 [54  6 39 18]]
```

**->compute Summary Statistics**

[265]:
```
#mean
mean = np.mean(data)
print("Mean of Data:", mean)
#median
median = np.median(data)
print("Median of Data:", median)
#standard deviation
std_dev = np.std(data)
print("Standard Deviation of Data:", std_dev)
#sum
sum_all = np.sum(data)
print("Sum of All Elements in Data:", sum_all)
```

```
Mean of Data: 35.95
Median of Data: 34.0
Standard Deviation of Data: 22.155078424596017
Sum of All Elements in Data: 719
```

[267]:
```
#Compute statistics along specific axis
mean_axis0 = np.mean(data, axis=0)  #axis=0 indicates Mean for each column
mean_axis1 = np.mean(data, axis=1)  # axis=1 indicates Mean for each row
print("\nMean along columns:", mean_axis0)
print("Mean along rows:", mean_axis1)
```

```

Mean along columns: [38.2 36.2 36.2 33.2]
Mean along rows: [41.75 41.   49.25 18.5  29.25]
```

**->Grouping Data and Aggregations**

[272]:
```
#Grouping data by columns
sum= np.sum(data, axis=0)  # Sum for each column
print("Sum along axis columns:", sum)
```

```
Sum along axis columns: [191 181 181 166]
```

[274]:
```
# Grouping data by rows
mean= np.mean(data, axis=1)  # Mean for each row
print("Mean along axis rows:", mean)
```

```
Mean along axis rows: [41.75 41.   49.25 18.5  29.25]
```

[278]:
```
# Variance and range
variance = np.var(data)
range= np.max(data) - np.min(data)
print("Variance is:", variance)
print("Range is:", range_values)
```

```
Variance is: 490.84749999999997
Range is: 75
```

[280]:
```
# Find maximum and minimum values for each column
max= np.max(data, axis=0)
min= np.min(data, axis=0)
print("Max along columns:", max)
print("Min along columns:", min)
```

```
Max along columns: [54 76 75 58]
Min along columns: [33  1  1 18]
```

### 1.5 Step 4:Data Analysis

[311]:
```
#arr1=np.random.uniform(1,1000,100)
#
rrr2=np.random.uniform(1,1000,100)
```

**->Correlation Coefficient:**

[320]:
```
arr1=np.random.uniform(1,1000,100)
arr2=np.random.uniform(1,1000,100)
correlation = np.corrcoef(arr1, arr2)[0][1]
print("Correlation:", correlation)
```

```
Correlation: 0.09455043404719411
```

**->Detecting outliers:**

[323]:
```
mean = np.mean(arr1)
std_dev = np.std(arr1)
outliers = arr1[np.abs(arr1 - mean) > 1 * std_dev]
print("Number of outliers in arr1:", len(outliers))
```

```
Number of outliers in arr1: 41
```

**->Calculate Percentiles:**

[330]:
```
percentiles = [25, 50, 75]
percentile_arr1 = np.percentile(arr1, percentiles)
percentile_arr2= np.percentile(arr2, percentiles)

print(f"Percentiles for arr1 (25th, 50th, 75th): {percentile_arr1}")
print(f"Percentiles for arr2 (25th, 50th, 75th): {percentile_arr2}")
```

```
Percentiles for arr1 (25th, 50th, 75th): [304.64386563 580.27898551
809.76323235]
Percentiles for arr2 (25th, 50th, 75th): [258.05304538 490.78726706
704.68296332]
```

## 1.6 Step5:Application in Data science

In conclusion, utilizing NumPy in a program offers significant benefits for data science professionals, particularly when handling numerical computations. NumPy's powerful features and optimized performance make it a preferred choice over traditional Python data structures such as lists and tuples. #### Here are some key advantages:

**Performance and Speed:** NumPy arrays are implemented in C, enabling faster execution of operations compared to Python lists, which are implemented in Python. This speed advantage is crucial when dealing with large datasets

**Memory Efficiency:** NumPy arrays consume less memory compared to Python lists. This efficiency arises from the fact that NumPy arrays allows for faster data access and processing.

**Vectorized Operations:** NumPy allows for vectorized operations, which means that operations can be applied to entire arrays at once without the need for explicit loops.

**Broad Functionality:** NumPy provides a comprehensive set of mathematical functions, including linear algebra, random number generation, and statistical operations, all optimized for use with arrays.

**Ease of Use:** The syntax and structure of NumPy are intuitive and easy to learn, allowing data scientists to quickly adopt and apply it to their projects. #### NumPy is crucial in real-world applications such as:

**Machine Learning:** For data preprocessing, implementing algorithms, and performing matrix operations essential for training models.

**Financial Analysis:** Used in portfolio optimization, time series analysis, and risk management by enabling fast and efficient numerical computations.

**Scientific Research:** Supports simulations, genomic data analysis, and image processing by providing tools for handling large datasets and performing complex calculations.

**Astronomy:** Essential for processing astronomical data and simulating cosmic events.

**Engineering:** Used in signal processing and control systems design, where efficient matrix operations and numerical computation are key.

[ ]:
```

```