untitled2

August 31, 2024

1. Getting Familiar with Numpy: Explore NumPy's core functionalities, including creating arrays, performing basic operations, and understanding array properties.

[2]:
```
pip install numpy
```

```
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 24.0 -> 24.2
[notice] To update, run: python.exe -m pip install --upgrade pip

Requirement already satisfied: numpy in
c:\users\sivasai\appdata\local\programs\python\python312\lib\site-packages
(2.1.0)
```

[26]:
```
# first we need to create an array
import numpy as np
# creating 1d array
array=np.array([1,2,3,4,5,6,7,8,9])
# creating 2d array
array_2d=np.array([[11,12,13],[14,15,16],[17,18,19]])
#creating array with zeros
c = np.zeros((2, 3))
print("Array of zeros:", c)
print(array)
print(array_2d)
```

```
Array of zeros: [[0. 0. 0.]
 [0. 0. 0.]]
[1 2 3 4 5 6 7 8 9]
[[11 12 13]
 [14 15 16]
 [17 18 19]]
```

[27]:
```
# Basic arithmetic operations
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
sum= a+b
print(sum)
sub= a-b
```

```
print(sub)
multi=a*b
print(multi)
div=a/b
print(div)
power=b**a
print(power)
```

```
[11 22 33 44]
[ -9 -18 -27 -36]
[ 10  40  90 160]
[0.1 0.1 0.1 0.1]
[      10      400    27000 2560000]
```

[28]:
```
#basic array properties
shape= array_2d.shape
ndim=array_2d.ndim
size=array_2d.size
print(shape)
print(ndim)
print(size)
print(type(array_2d))
print(type(array))
```

```
(3, 3)
2
9
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

2. Data Manipulation: Write a Python program to demonstrate data manipulation using Numpy. Focus on array creation, indexing, slicing, reshaping, and applying mathematical operations.

[29]:
```
#array creation
print(array)
print(array_2d)
```

```
[1 2 3 4 5 6 7 8 9]
[[11 12 13]
 [14 15 16]
 [17 18 19]]
```

[30]:
```
#indexing
#accessing an element
element=array[3]
element1=array_2d[2,2]
print(element,element1)
```

```
#modifying elements in array
array[0]=34
print("moddified\n",array)
array_2d[2,1]=45
print("moddified\n",array_2d)
```

```
4 19
moddified
 [34  2  3  4  5  6  7  8  9]
moddified
 [[11 12 13]
 [14 15 16]
 [17 45 19]]
```

[31]:
```
#slicing a array
slice=array[1:6]
slice1=array_2d[0:2,1:3]
print(slice)
print(slice1)
#moddifing the elements in array
array[1:3]=[77,88]
print(array)
```

```
[2 3 4 5 6]
[[12 13]
 [15 16]]
[34 77 88  4  5  6  7  8  9]
```

[36]:
```
#reshaping arrays
reshaped_array = array.reshape(3,3)
print(reshaped_array)
#flattening a 2d array into 1d array
flattened_array=array_2d.flatten()
print(flattened_array)
# Transposing a 2D array
transposed_array = array_2d.T
print(transposed_array)
# Resizing an array
resized_array = np.resize(array_2d, (2, 4))
```

```
[[34 77 88]
 [ 4  5  6]
 [ 7  8  9]]
[11 12 13 14 15 16 17 45 19]
[[11 14 17]
 [12 15 45]
 [13 16 19]]
```

5. Applying Mathematical Operations

[37]:
```python
# Applying a mathematical function square root
sqrt_array =np.sqrt(array)
print(sqrt_array)
# Sum of all elements in a 2D array
sum_elements =np.sum(array_2d)
print(sum_elements)
# Mean of each column in a 2D array
mean=np.mean(array_2d, axis=0)
print(mean)
# Applying sin function to each element in a 1D array
sin_array = np.sin(array)
print(sin_array)
# Matrix multiplication of two 2D arrays
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])
matrix_product = np.dot(matrix_a, matrix_b)
print(matrix_product)
```

```
[5.83095189 8.77496439 9.38083152 2.         2.23606798 2.44948974
 2.64575131 2.82842712 3.        ]
162
[14. 24. 16.]
[ 0.52908269  0.99952016  0.0353983  -0.7568025  -0.95892427 -0.2794155
  0.6569866   0.98935825  0.41211849]
[[19 22]
 [43 50]]
```

3.Data Aggregation: o Use Numpy functions to compute summary statistics like mean, median, standard deviation, and sum. Practice grouping data and performing aggregations. Computing Summary Statistics

[40]:
```python
# Mean: Average of the array
mean_value = np.mean(array)
print(mean_value)
# Median: Middle value in the sorted array
median_value = np.median(array)
print(median_value)
# Standard Deviation
std_devia = np.std(array)
print(std_devia)
# Sum: Total of all elements in the array
suming = np.sum(array)
print(suming)
# Variance
variance_value = np.var(array)
print( variance_value)
```

```
# Minimum
min_value = np.min(array)
print(min_value)
# Maximum
max_value = np.max(array)
print( max_value)
```

```
26.444444444444443
8.0
31.280491297332098
238
978.4691358024692
4
88
```

[41]:

```
scores = np.array([
    [85, 90, 78],
    [88, 76, 92],
    [94, 85, 89],
    [70, 82, 88],
    [68, 79, 95]
])

# Grouping students based on their average score
average_scores = np.mean(scores, axis=1)
print("Average Scores per Student:", average_scores)

# Group 1: Students with an average score above 85
group_1 = scores[average_scores > 85]
print("\nGroup 1 (Average Score > 85):\n", group_1)

# Group 2: Students with an average score of 85 or below
group_2 = scores[average_scores <= 85]
print("\nGroup 2 (Average Score <= 85):\n", group_2)

# Aggregating data within groups

# Mean score of Group 1 students across all subjects
mean_group_1 = np.mean(group_1, axis=0)
print("\nMean Scores of Group 1 (Across Subjects):", mean_group_1)

# Mean score of Group 2 students across all subjects
mean_group_2 = np.mean(group_2, axis=0)
print("Mean Scores of Group 2 (Across Subjects):", mean_group_2)

# Total sum of scores in each subject for all students
total_scores_per_subject = np.sum(scores, axis=0)
```

```
print("\nTotal Scores Per Subject:", total_scores_per_subject)

# Maximum score in each subject across all students
max_scores_per_subject = np.max(scores, axis=0)
print("Maximum Scores Per Subject:", max_scores_per_subject)

# Standard deviation of scores in each subject
std_dev_per_subject = np.std(scores, axis=0)
print("Standard Deviation Per Subject:", std_dev_per_subject)
```

```
Average Scores per Student: [84.33333333 85.33333333 89.33333333 80.
80.66666667]

Group 1 (Average Score > 85):
 [[88 76 92]
 [94 85 89]]

Group 2 (Average Score <= 85):
 [[85 90 78]
 [70 82 88]
 [68 79 95]]

Mean Scores of Group 1 (Across Subjects): [91.  80.5 90.5]
Mean Scores of Group 2 (Across Subjects): [74.33333333 83.66666667 87.        ]

Total Scores Per Subject: [405 412 442]
Maximum Scores Per Subject: [94 90 95]
Standard Deviation Per Subject: [10.23718711  4.84148737  5.74804315]
```

[42]:
```
subject1_scores = np.array([85, 90, 78, 88, 92, 94, 76, 85, 89, 70])
subject2_scores = np.array([80, 88, 74, 85, 90, 95, 72, 82, 88, 68])
# Correlation coefficient
correlation = np.corrcoef(subject1_scores, subject2_scores)[0, 1]
print("Correlation between Subject 1 and Subject 2 Scores:", correlation)
# Covariance matrix
covariance_matrix = np.cov(subject1_scores, subject2_scores)
print("\nCovariance Matrix:\n", covariance_matrix)
```

```
Correlation between Subject 1 and Subject 2 Scores: 0.985062212208179

Covariance Matrix:
 [[59.34444444 65.84444444]
 [65.84444444 75.28888889]]
```

[44]:
```
# Percentiles of Subject 1 Scores
percentile_25 = np.percentile(subject1_scores, 25)
percentile_50 = np.percentile(subject1_scores, 50)  # Median
```

```
percentile_75 = np.percentile(subject1_scores, 75)
print("\n25th Percentile of Subject 1 Scores:", percentile_25)
print("50th Percentile (Median) of Subject 1 Scores:", percentile_50)
print("75th Percentile of Subject 1 Scores:", percentile_75)
```

```
25th Percentile of Subject 1 Scores: 79.75
50th Percentile (Median) of Subject 1 Scores: 86.5
75th Percentile of Subject 1 Scores: 89.75
```

[45]:
```
# Simulating a large dataset (1 million random scores)
large_dataset = np.random.normal(loc=50, scale=10, size=1000000)
# Calculating summary statistics on the large dataset
mean_large = np.mean(large_dataset)
std_large = np.std(large_dataset)
median_large = np.median(large_dataset)
percentile_90_large = np.percentile(large_dataset, 90)
print("\nLarge Dataset Analysis:")
print("Mean:", mean_large)
print("Standard Deviation:", std_large)
print("Median:", median_large)
print("90th Percentile:", percentile_90_large)
```

```
Large Dataset Analysis:
Mean: 50.0140109288424
Standard Deviation: 10.004257156087506
Median: 50.011722021062525
90th Percentile: 62.82934079592566
```

Application in Data Science: o Conclude your program by explaining how the use of Numpy in your program can help a data science professional. Discuss the advantages of using Numpy over traditional Python data structures for numerical computations. o Provide real-world examples where NumPy's capabilities are crucial, such as in machine learning, financial analysis, and scientific research.

[ ]:
```
Application of NumPy in Data Science
1. How NumPy Benefits Data Science Professionals
a. Performance and Efficiency: NumPy is optimized for performance, primarily␣
↪because it is implemented in C and Fortran. This allows it to perform␣
↪operations significantly faster than traditional Python lists and loops.␣
↪Operations that involve large datasets, such as matrix multiplications,␣
↪element-wise operations, and linear algebra, are highly efficient with NumPy.
↪ For instance, element-wise operations on arrays are typically performed in␣
↪a fraction of the time compared to using Python loops.
```

```
b. Memory Efficiency: NumPy arrays are more memory-efficient than Python lists.
 They store elements in contiguous blocks of memory, leading to reduced
 memory overhead. This is particularly important when working with large
 datasets, as it allows for better utilization of system resources.

c. Broad Range of Mathematical Functions: NumPy offers a vast library of
 mathematical functions, including those for linear algebra, random number
 generation, Fourier transforms, and more. These functions are highly
 optimized and can be applied directly to arrays, enabling quick computations.
  This is essential in data science for tasks such as statistical analysis,
 data transformation, and simulation.

d. Broadcasting: Broadcasting is a powerful feature in NumPy that allows for
 vectorized operations between arrays of different shapes. This capability
 simplifies code, reducing the need for explicit loops, which not only makes
 the code more readable but also improves performance. Broadcasting is
 particularly useful in machine learning when performing operations on
 batches of data.

e. Integration with Other Libraries: NumPy is the foundational package for many
 other data science libraries like Pandas, SciPy, TensorFlow, and
 Scikit-learn. Its array objects are often the standard input format for
 these libraries. This integration makes it easier to transition between
 different stages of a data science project, from data manipulation to model
 building and evaluation.

f. Data Manipulation and Transformation: NumPy provides powerful tools for data
 manipulation, such as reshaping, slicing, and indexing arrays. This allows
 data scientists to easily clean, filter, and transform data, which are
 crucial steps in preparing data for analysis or machine learning models.

g. Simplicity and Consistency: NumPy's API is simple and consistent, making it
 easier for data scientists to write, understand, and maintain code. This
 simplicity extends to complex mathematical operations, which can often be
 performed with a single function call.

2. Advantages of NumPy over Traditional Python Data Structures
a. Speed and Performance: While Python lists are versatile, they are not
 optimized for numerical operations. NumPy arrays, in contrast, are designed
 for efficient numerical computation. Operations that would take considerable
 time using Python lists (due to the overhead of Python's interpreted nature)
 are executed in a fraction of the time with NumPy arrays.
```

```
b. Type Consistency and Safety: NumPy arrays are homogeneous; all elements must be of the same data type. This type consistency not only ensures safer operations (by preventing operations on incompatible data types) but also optimizes performance by enabling the use of low-level optimizations that are not possible with Python lists, which can hold elements of varying types.

c. Vectorization: Traditional Python loops are slow due to the overhead of interpreting each iteration. NumPy supports vectorization, which allows for batch operations on data without the need for explicit loops. This not only speeds up computations but also simplifies code by reducing the need for loop constructs.

d. Built-in Functions for Complex Operations: NumPy provides a rich set of functions for performing complex operations, such as Fourier transforms, statistical analysis, linear algebra, and more. These operations would either be impossible or highly inefficient to implement using basic Python data structures.

3. Real-World Examples of NumPy’s Capabilities
a. Machine Learning:

Data Preprocessing: In machine learning, data preprocessing often involves normalizing, scaling, or transforming features. NumPy is frequently used for these tasks because it can handle large datasets efficiently.
Batch Operations: Many machine learning algorithms, particularly those involving neural networks, operate on batches of data. NumPy's ability to perform fast, batch-wise operations on large matrices makes it ideal for these applications.
Gradient Descent: In optimization algorithms like gradient descent, where derivatives and matrix multiplications are computed repeatedly, NumPy’s efficient linear algebra capabilities are crucial.
b. Financial Analysis:

Time Series Analysis: Financial data often comes in the form of time series, where each data point is associated with a timestamp. NumPy’s efficient array operations are ideal for performing calculations across these time series, such as moving averages, correlations, and volatility measures.
Risk Management: Financial analysts use NumPy to calculate portfolio risk metrics such as Value at Risk (VaR) and Expected Shortfall. These calculations often involve large covariance matrices and require the efficient linear algebra routines provided by NumPy.
Monte Carlo Simulations: In finance, Monte Carlo simulations are used to model the probability of different outcomes in a process that cannot easily be predicted. NumPy is commonly used to generate random samples and perform simulations due to its efficient random number generation and array operations.
```

```
c. Scientific Research:

Simulation and Modeling: In fields like physics, chemistry, and biology,
  ↪researchers use NumPy to simulate real-world processes. For example, in
  ↪computational physics, NumPy is used to solve differential equations and
  ↪simulate physical systems.
Data Analysis: Large datasets generated by scientific experiments, such as
  ↪those from particle accelerators or space telescopes, are often analyzed
  ↪using NumPy. Its ability to handle large arrays and perform statistical
  ↪analysis efficiently makes it a critical tool in scientific research.
Image Processing: NumPy is also used in image processing, where images are
  ↪represented as large multi-dimensional arrays. Operations like filtering,
  ↪transformation, and feature extraction are performed efficiently using NumPy.

Conclusion
NumPy is an essential tool in the toolkit of any data science professional. Its
  ↪performance, efficiency, and extensive functionality make it superior to
  ↪traditional Python data structures for numerical computations. Whether in
  ↪machine learning, financial analysis, or scientific research, NumPy's
  ↪capabilities enable data scientists to handle complex, large-scale data
  ↪efficiently and effectively. Its role as the backbone of many other data
  ↪science libraries further cements its importance in the field.
```