Numpy

October 6, 2024

NUMPY

[ ]:
```
# 1. What is a Python library? Why do we use Python libraries?
A Python library is a collection of modules and packages that provide pre-written code to help you perform common tasks without having to write the code from scratch. Libraries are essentially a set of functions, classes, and variables that you can use to accomplish specific tasks in your programs.

Why We Use Python Libraries:
Code Reusability: Libraries allow you to reuse code, which saves time and effort. Instead of writing common functions or algorithms yourself, you can leverage existing libraries.

Efficiency: Many libraries are optimized for performance, providing efficient implementations of algorithms and data structures. This can lead to faster development and execution times.

Simplification: Libraries simplify complex tasks. For example, libraries like NumPy make it easy to perform mathematical operations on large datasets, while Pandas provides convenient data manipulation tools.

Community Support: Popular libraries often have large communities around them. This means you can find extensive documentation, tutorials, and support, making it easier to learn and troubleshoot.

Functionality: Libraries extend the functionality of Python. For instance, libraries like Matplotlib for plotting, Requests for handling HTTP requests, and Flask for web development provide tools that help you accomplish specific tasks effectively.

Standardization: Using established libraries can promote coding standards and best practices, as they often follow community conventions and are regularly maintained and updated.
```

[ ]:
```
# 2. What is the difference between Numpy array and List?
```

```
The primary differences between a NumPy array and a Python list revolve around
↪performance, functionality, and usability. Here are the key distinctions:

1. Data Type Uniformity:
NumPy Array: All elements in a NumPy array must be of the same data type (e.g.,
↪all integers, all floats). This uniformity allows for more efficient storage
↪and computation.
Python List: A Python list can contain elements of different data types (e.g.,
↪integers, strings, floats, and even other lists).
2. Performance:
NumPy Array: NumPy arrays are implemented in C, making operations on them
↪significantly faster than operations on lists, especially for large datasets.
↪ NumPy uses contiguous blocks of memory, which helps improve cache
↪efficiency.
Python List: Lists are slower for mathematical operations and manipulations,
↪particularly as they grow larger.
3. Functionality:
NumPy Array: NumPy provides a wide range of mathematical functions, operations,
↪and capabilities (e.g., element-wise operations, linear algebra, Fourier
↪transforms). You can perform complex mathematical operations directly on
↪NumPy arrays without needing explicit loops.
Python List: Lists do not have built-in mathematical capabilities. You would
↪need to use loops or list comprehensions to perform operations on list
↪elements.
4. Memory Usage:
NumPy Array: NumPy arrays generally require less memory than Python lists due
↪to their fixed data types and more efficient storage.
Python List: Lists use more memory because they need to accommodate a variety
↪of object types and overhead for managing dynamic resizing.
5. Dimensionality:
NumPy Array: NumPy supports multi-dimensional arrays (e.g., 2D matrices, 3D
↪tensors) natively, which makes it easy to work with matrices and
↪higher-dimensional data.
Python List: While you can create lists of lists to represent multi-dimensional
↪data, it's less efficient and more cumbersome to manipulate than using NumPy
↪arrays.
6. Ease of Use:
NumPy Array: For mathematical operations and scientific computing, NumPy
↪provides a more straightforward and readable syntax (e.g., broadcasting).
Python List: Lists are more versatile in general programming but less suited
↪for numerical computations.
```

[1]:
```
# 3. Find the shape, size and dimension of the following array?
#[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np
array = np.array([[1, 2, 3, 4],
```

```
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
shape = array.shape
size = array.size
dimension = array.ndim

print("Shape:", shape)
print("Size:", size)
print("Dimension:", dimension)
```

```
Shape: (3, 4)
Size: 12
Dimension: 2
```

[3]:
```
# 4. Write python code to access the first row of the following array?
#[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np

array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
first_row = array[0]
print("First row:", first_row)  # Output: [1 2 3 4]
import numpy as np

# Create the array
array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])

# Access the first row
first_row = array[0]
print("First row:", first_row)  # Output: [1 2 3 4]
```

```
First row: [1 2 3 4]
First row: [1 2 3 4]
```

[9]:
```
# 5. How do you access the element at the third row and fourth column from the
 ↪given numpy array?
# [[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np

# Create the array
array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
element = array[2, 3]
print("Element at third row and fourth column:", element)
```

```python
import numpy as np

array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
```

```
Element at third row and fourth column: 12
```

[11]:
```python
# 6. Write code to extract all odd-indexed elements from the given numpy array?
#[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np

array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])

odd_indexed_elements = array[:, 1::2]
print("Odd-indexed elements:", odd_indexed_elements)
import numpy as np


array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])



odd_indexed_elements = array[:, 1::2]
print("Odd-indexed elements:", odd_indexed_elements)
```

```
Odd-indexed elements: [[ 2  4]
 [ 6  8]
 [10 12]]
Odd-indexed elements: [[ 2  4]
 [ 6  8]
 [10 12]]
```

[13]:
```python
# 7. How can you generate a random 3x3 matrix with values between 0 and 1?
import numpy as np


random_matrix = np.random.rand(3, 3)

print("Random 3x3 Matrix:")
print(random_matrix)
import numpy as np


random_matrix = np.random.rand(3, 3)
```

```python
print("Random 3x3 Matrix:")
print(random_matrix)
```

```
Random 3x3 Matrix:
[[0.57447707 0.04737315 0.31126281]
 [0.9844504  0.40955814 0.09746316]
 [0.10507747 0.34260093 0.82367233]]
Random 3x3 Matrix:
[[0.1599551  0.07843301 0.09366908]
 [0.78458889 0.71385586 0.4672639 ]
 [0.25780027 0.41131434 0.96947026]]
```

[15]:

```python
# 8. Describe the difference between np.random.rand and np.random.randn?
import numpy as np
uniform_random = np.random.rand(3, 3)
print(uniform_random)
normal_random = np.random.randn(3, 3)
print(normal_random)
import numpy as np


uniform_matrix = np.random.rand(3, 3)
print("Uniform Distribution (0 to 1):")
print(uniform_matrix)


normal_matrix = np.random.randn(3, 3)
print("\nNormal Distribution (mean 0, std 1):")
print(normal_matrix)
```

```
[[0.62802156 0.68873975 0.43072044]
 [0.69646103 0.1917385  0.65270834]
 [0.05799091 0.29072541 0.9240298 ]]
[[ 0.37305004  1.17691414  0.40956892]
 [-0.65080277 -1.05447098 -1.3801399 ]
 [-0.88560932 -0.52062262  0.00211561]]
Uniform Distribution (0 to 1):
[[0.81312497 0.51367893 0.00338047]
 [0.51988071 0.14995828 0.62085018]
 [0.03412232 0.62192402 0.2150642 ]]

Normal Distribution (mean 0, std 1):
[[ 0.43202701 -0.13111872 -0.04456481]
 [ 0.27759948 -0.97296793  0.67579719]
 [ 0.30488375  0.40437231 -1.38831672]]
```

[17]:
```
# 9. Write code to increase the dimension of the following array?
#[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np

array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])


expanded_array = np.expand_dims(array, axis=0)
print("Array after increasing dimension using expand_dims:")
print(expanded_array)


reshaped_array = array.reshape(1, 3, 4)
print("\nArray after increasing dimension using reshape:")
print(reshaped_array)
```

```
Array after increasing dimension using expand_dims:
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]]

Array after increasing dimension using reshape:
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]]
```

[1]:
```
# 10. How to transpose the following array in NumPy?
#[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np


array = np.array([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])


transposed_array_t = array.T
print("Transposed array using .T:")
print(transposed_array_t)


transposed_array_func = np.transpose(array)
print("\nTransposed array using np.transpose():")
print(transposed_array_func)
```

```
Transposed array using .T:
```

```
[[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]

Transposed array using np.transpose():
[[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]
```

[3]:
```python
# 11. Consider the following matrix:
#Matrix A: [[1, 2, 3, 4] [5, 6, 7, 8],[9, 10, 11, 12]]
#Matrix B: [[1, 2, 3, 4] [5, 6, 7, 8],[9, 10, 11, 12]]
import numpy as np
A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

B = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
```

[5]:
```python
#
index_wise_multiplication = A * B
print("Index-wise Multiplication (A * B):")
print(index_wise_multiplication)
```

```
Index-wise Multiplication (A * B):
[[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]
```

[7]:
```python
#
matrix_multiplication = A @ B.T  # B is transposed for valid multiplication
print("\nMatrix Multiplication (A @ B.T):")
print(matrix_multiplication)
```

```

Matrix Multiplication (A @ B.T):
[[ 30  70 110]
 [ 70 174 278]
 [110 278 446]]
```

[9]:
```python
#
addition = A + B
print("\nAddition (A + B):")
print(addition)
```

```
Addition (A + B):
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]]
```

[11]:
```
#
subtraction = A - B
print("\nSubtraction (A - B):")
print(subtraction)
```

```
Subtraction (A - B):
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

[13]:
```
#
division = B / A
print("\nDivision (B / A):")
print(division)
```

```
Division (B / A):
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[15]:
```
#
import numpy as np


A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

B = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

# Index-wise Multiplication (Element-wise)
index_wise_multiplication = A * B
print("Index-wise Multiplication (A * B):")
print(index_wise_multiplication)

# Matrix Multiplication
matrix_multiplication = A @ B.T  # Transposing B for multiplication
print("\nMatrix Multiplication (A @ B.T):")
```

```
print(matrix_multiplication)

# Addition
addition = A + B
print("\nAddition (A + B):")
print(addition)

# Subtraction
subtraction = A - B
print("\nSubtraction (A - B):")
print(subtraction)

# Division
division = B / A
print("\nDivision (B / A):")
print(division)
```

```
Index-wise Multiplication (A * B):
[[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]

Matrix Multiplication (A @ B.T):
[[ 30  70 110]
 [ 70 174 278]
 [110 278 446]]

Addition (A + B):
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]]

Subtraction (A - B):
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]

Division (B / A):
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[17]:
```
# 12 . Which function in Numpy can be used to swap the byte order of an array?
import numpy as np


array = np.array([1, 2, 3, 4], dtype='<i4')
```

```
print("Original array:", array)


swapped_array = array.byteswap()
print("Swapped byte order array:", swapped_array)
```

```
Original array: [1 2 3 4]
Swapped byte order array: [16777216 33554432 50331648 67108864]
```

[19]:
```
# 13. What is the significance of the np.linalg.inv function?
import numpy as np


A = np.array([[4, 7],
              [2, 6]])


A_inv = np.linalg.inv(A)

print("Original Matrix:")
print(A)
print("\nInverse Matrix:")
print(A_inv)


identity = np.dot(A, A_inv)
print("\nProduct of A and its inverse (should be identity):")
print(identity)
```

```
Original Matrix:
[[4 7]
 [2 6]]

Inverse Matrix:
[[ 0.6 -0.7]
 [-0.2  0.4]]

Product of A and its inverse (should be identity):
[[ 1.00000000e+00 -1.11022302e-16]
 [-1.11022302e-16  1.00000000e+00]]
```

[21]:
```
# 14. What does the np.reshape function do, and how is it used?
import numpy as np

array_1d = np.arange(12)
print("Original 1D Array:")
print(array_1d)
```

```
array_2d = np.reshape(array_1d, (3, 4))
print("\nReshaped to 2D Array (3x4):")
print(array_2d)


array_3d = np.reshape(array_1d, (2, 3, 2))
print("\nReshaped to 3D Array (2x3x2):")
print(array_3d)


array_auto = np.reshape(array_1d, (4, -1))
print("\nReshaped with -1 (4 rows, auto columns):")
print(array_auto)
```

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

Reshaped to 2D Array (3x4):
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Reshaped to 3D Array (2x3x2):
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]

Reshaped with -1 (4 rows, auto columns):
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

[23]:

```
# 15 . What is broadcasting in Numpy?
import numpy as np

vector = np.array([1, 2, 3])  # Shape (3,)


matrix = np.array([[10, 20, 30],
                   [40, 50, 60]])  # Shape (2, 3)
```

```python
result = matrix + vector

print("Matrix:")
print(matrix)

print("\nVector:")
print(vector)

print("\nResult of Broadcasting Addition (matrix + vector):")
print(result)
```

```
Matrix:
[[10 20 30]
 [40 50 60]]

Vector:
[1 2 3]

Result of Broadcasting Addition (matrix + vector):
[[11 22 33]
 [41 52 63]]
```

[ ]:

```python

```