numpy-assignment-solution

August 13, 2024

[1]:
```
'''Jan_24^24_Numpy_Assignment_16_Solution'''
```

[1]:
```
'Jan_24^24_Numpy_Assignment_16_Solution'
```

[2]:
```
#1.What is a Python library? Why do we use Python libraries?
'''A Python library is a collection of pre-written code and functions that␣
 ↪extend the capabilities of Python.
It provides ready-to-use tools and functionalities for specific tasks, making␣
 ↪it easier and more efficient to develop
software applications.

We use Python libraries for several reasons. Firstly, libraries save us time␣
 ↪and effort by providing pre-built solutions
for common tasks. Instead of writing code from scratch, we can simply import␣
 ↪and use the functions and classes from the
library. This speeds up development and allows us to focus on solving the␣
 ↪specific problem at hand.

Additionally, Python libraries offer a wide range of specialized␣
 ↪functionalities, such as data manipulation, scientific
computing, web development, machine learning, and more. By utilizing these␣
 ↪libraries, we can leverage the expertise and
efforts of the Python community, benefiting from their tested and optimized␣
 ↪code.

Overall, Python libraries enhance the capabilities of Python, allowing us to␣
 ↪build powerful and feature-rich applications
with less effort. They promote code reusability, collaboration, and enable us␣
 ↪to tackle complex tasks more easily.
'''
```

[2]:
```
'A Python library is a collection of pre-written code and functions that extend
the capabilities of Python. \nIt provides ready-to-use tools and functionalities
for specific tasks, making it easier and more efficient to develop\nsoftware
applications.\n\nWe use Python libraries for several reasons. Firstly, libraries
save us time and effort by providing pre-built solutions\nfor common tasks.
Instead of writing code from scratch, we can simply import and use the functions
```

```
and classes from the\nlibrary. This speeds up development and allows us to focus
on solving the specific problem at hand.\n\nAdditionally, Python libraries offer
a wide range of specialized functionalities, such as data manipulation,
scientific\ncomputing, web development, machine learning, and more. By utilizing
these libraries, we can leverage the expertise and \nefforts of the Python
community, benefiting from their tested and optimized code.\n\nOverall, Python
libraries enhance the capabilities of Python, allowing us to build powerful and
feature-rich applications\nwith less effort. They promote code reusability,
collaboration, and enable us to tackle complex tasks more easily.\n'
```

[3]:
```
#2. What is the difference between Numpy array and List?
'''A NumPy array is a data structure provided by the NumPy library in Python.␣
 ↪It is similar to a list, but with some key
differences.
Here are a few differences between a NumPy array and a list:

1. **Memory Efficiency**: NumPy arrays are more memory efficient compared to␣
 ↪lists. They store data in a contiguous block
of memory, which allows for faster access and manipulation of elements.

2. **Homogeneous Data Type**: NumPy arrays require all elements to have the␣
 ↪same data type (e.g., all integers, all
floats). This allows for more efficient storage and computation. In contrast,␣
 ↪lists can contain elements of different
data types.

3. **Vectorized Operations**: NumPy arrays support vectorized operations, which␣
 ↪means you can perform operations on
entire arrays at once, rather than iterating through each element. This makes␣
 ↪computations faster and more concise.

4. **Functionality and Performance**: NumPy provides a wide range of␣
 ↪mathematical and scientific functions that can be
applied directly to arrays. These functions are optimized for performance and␣
 ↪can be faster than using equivalent
operations on lists.

However, lists also have their advantages. Lists are more flexible and can␣
 ↪contain elements of different data types.
They can be easily modified, appended, or extended. Lists are also more␣
 ↪suitable for small-scale, general-purpose tasks.

In summary, NumPy arrays are specialized data structures that offer better␣
 ↪performance and efficiency for numerical
computations, while lists are more versatile
and suitable for general-purpose tasks.
```

```
'''
```

[3]:
```
'A NumPy array is a data structure provided by the NumPy library in Python. It
is similar to a list, but with some key\ndifferences. \nHere are a few
differences between a NumPy array and a list:\n\n1. **Memory Efficiency**: NumPy
arrays are more memory efficient compared to lists. They store data in a
contiguous block\nof memory, which allows for faster access and manipulation of
elements.\n\n2. **Homogeneous Data Type**: NumPy arrays require all elements to
have the same data type (e.g., all integers, all \nfloats). This allows for more
efficient storage and computation. In contrast, lists can contain elements of
different \ndata types.\n\n3. **Vectorized Operations**: NumPy arrays support
vectorized operations, which means you can perform operations on\nentire arrays
at once, rather than iterating through each element. This makes computations
faster and more concise.\n\n4. **Functionality and Performance**: NumPy provides
a wide range of mathematical and scientific functions that can be \napplied
directly to arrays. These functions are optimized for performance and can be
faster than using equivalent\noperations on lists.\n\nHowever, lists also have
their advantages. Lists are more flexible and can contain elements of different
data types.\nThey can be easily modified, appended, or extended. Lists are also
more suitable for small-scale, general-purpose tasks.\n\nIn summary, NumPy
arrays are specialized data structures that offer better performance and
efficiency for numerical\ncomputations, while lists are more versatile\nand
suitable for general-purpose tasks.\n'
```

[18]:
```
import numpy as np
```

[19]:
```
'''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
arr=[[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12]]
array=np.array(arr)
print("Array:",array)
print("Shape of Array:",array.shape)
print("Size of Array:",array.size)
print("Arrays is of dimension:",array.ndim)
```

```
Array: [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Shape of Array: (3, 4)
Size of Array: 12
Arrays is of dimension: 2
```

[20]:
```
'''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.asarray([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])
#Accessing the first row:
first_row=array[0]
print("First Row:",first_row)
```

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

[21]:
```
'''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
array=np.asanyarray([[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12]])
#Element from 3rd row and 4th column:
E_34 = array[2][3]
print("Element in 3rd row and 4th column:",E_34)
```

```
Element in 3rd row and 4th column: 12
```

[22]:
```
'''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.asanyarray([[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12]])
Odd_Indexed=array[1::2]
print("Odd Indexed Element:",Odd_Indexed)
```

```
Odd Indexed Element: [[5 6 7 8]]
```

[ ]:
```

```

[23]:
```
'''7. How can you generate a random 3x3 matrix with values between 0 and 1?'''
import numpy as np
mat_3x3=np.random.rand(3,3)
mat_3x3
```

[23]:
```
array([[0.83410639, 0.57462005, 0.7780523 ],
       [0.55954028, 0.27460394, 0.81220168],
       [0.15845144, 0.12478162, 0.26400989]])
```

[10]:
```
#8.Describe the difference between np.random.rand and np.random.randn?
'''
The main difference between `np.random.rand()` and `np.random.randn()` lies in␣
 ↪the distribution of the generated random
numbers.

- `np.random.rand()` generates random numbers from a uniform distribution␣
 ↪between 0 and 1. This means that each number
in the generated array has an equal chance of being any value between 0 and 1.

- On the other hand, `np.random.randn()` generates random numbers from a␣
 ↪standard normal distribution (also known as a
Gaussian or bell curve distribution) with a mean of 0 and a standard deviation␣
 ↪of 1. This means that the generated
numbers are more likely to be closer to 0, and less likely to be further away␣
 ↪from 0.

In summary, `np.random.rand()` generates random numbers from a uniform␣
 ↪distribution, while `np.random.randn()` generates
random numbers from a standard normal distribution.
'''
```

[10]:
```
'\nThe main difference between `np.random.rand()` and `np.random.randn()` lies
in the distribution of the generated random\nnumbers.\n\n- `np.random.rand()`
generates random numbers from a uniform distribution between 0 and 1. This means
that each number \nin the generated array has an equal chance of being any value
between 0 and 1.\n\n- On the other hand, `np.random.randn()` generates random
numbers from a standard normal distribution (also known as a \nGaussian or bell
curve distribution) with a mean of 0 and a standard deviation of 1. This means
that the generated \nnumbers are more likely to be closer to 0, and less likely
to be further away from 0.\n\nIn summary, `np.random.rand()` generates random
numbers from a uniform distribution, while `np.random.randn()` generates\nrandom
numbers from a standard normal distribution. \n'
```

[24]:
```
'''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
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])
#Check the dimensions:
print("Given arr has dimensions:",arr.ndim)
#Expanding the dimensions:
Expanded_arr=np.expand_dims(arr,axis=0)
print("Array with Increased Dimensions:",Expanded_arr)
print("Given arr now has dimensions:",Expanded_arr.ndim)
```

```
Given arr has dimensions: 2
Array with Increased Dimensions: [[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]]
Given arr now has dimensions: 3
```

[25]:
```
'''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
arr=np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
print("Original array:",arr)
print("Transposed_Array:",arr.T)
```

```
Original array: [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Transposed_Array: [[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]
```

[59]:
```
'''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]]
Perform the following operation using Python1
1. Index wise multiplication
2. Matix multiplication
3. Add both the matices
4. Subtact matix B from matix A
5. Divide Matix B by A
```

```
'''
import numpy as np
matrix_A = np.array([[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]])
matrix_B = np.array([[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]])

#1. Index wise multiplication:
Index_Wise_Multiplication=matrix_A*matrix_B
print("1.Index Wise Multiplication:")
print(Index_Wise_Multiplication)

#2. Matix multiplication:
try:
    Matix_multiplication=matrix_A@matrix_B
except ValueError as V:
    print("2.Since, Matrix Multiplication is not possible as Column(matrix_A)!
 ↪=Row(matrix_B)",V)
#Let's do some arrangements:
mat_B=matrix_B.reshape(4,3)
Matix_multiplication=matrix_A@mat_B
print("Multiply by Re-shaping matrix_B:")
print(Matix_multiplication)

#3. Add both the matices
Addition_Of_Matrices=matrix_A+matrix_B
print("3.Addition Of Matrices:")
print(Addition_Of_Matrices)

#4.Subtact matix B from matrix A
Subtraction_Of_Matrices=matrix_A-matrix_B
print("4.Subtraction Of Matrices:")
print(Subtraction_Of_Matrices)

#5. Diide Matix B by Matrix A
Division_Of_Matrices=matrix_B/matrix_A
print("5.Division Of Matrices:")
print(Division_Of_Matrices)
```

```
1.Index Wise Multiplication:
[[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]
2.Since, Matrix Multiplication is not possible as
Column(matrix_A)!=Row(matrix_B) matmul: Input operand 1 has a mismatch in its
core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is
different from 4)
Multiply by Re-shaping matrix_B:
[[ 70  80  90]
```

```
 [158 184 210]
 [246 288 330]]
3.Addition Of Matrices:
[[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]]
4.Subtraction Of Matrices:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
5.Division Of Matrices:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[61]:
```python
#12. Which function in Numpy can be used to swap the byte order of an array?

'''To swap the byte order of an array in NumPy, you can use the `byteswap()`␣
 ↪function.'''

import numpy as np

arr = np.array([1, 2, 3, 4], dtype=np.int32)
print("Original array:", arr)

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

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

[90]:
```python
#13. What is the significance of the np.linalg.inv function?
'''It stands for "NumPy linear algebra inverse" and it's used to find the␣
 ↪inverse of a square matrix. The inverse of a matrix
has some interesting properties, like when you multiply a matrix by its␣
 ↪inverse, you get the identity matrix. It's like
finding the "opposite" of a matrix, kind of like division in regular arithmetic.
 ↪ This function comes in handy when solving
systems of linear equations or performing transformations in linear algebra.'''
import numpy as np
matrix=np.array([[2,2],[5,9]])
inverse_of_matrix=np.linalg.inv(matrix)
print("Original matrix:")
print(matrix)
print("Inverse of matrix:")
print(inverse_of_matrix)
print("Multiplication of both:",inverse_of_matrix@matrix)
```

```
Original matrix:
[[2 2]
 [5 9]]
Inverse of matrix:
[[ 1.125 -0.25 ]
 [-0.625  0.25 ]]
Multiplication of both: [[1. 0.]
 [0. 1.]]
```

[103]:
```
#14. What does the np.reshape function do, and how is it used?
''' It allows you to change the shape or dimensions of an array without␣
 ↪changing its data. It's like rearranging the elements
of an array to fit a different shape.

For example, let's say you have a 1D array with 12 elements, and you want to␣
 ↪reshape it into a 3x4 matrix. You can use
`np.reshape()` to achieve that. It's like rearranging the elements in a␣
 ↪different way, but keeping all the original values
intact.

You can also use `np.reshape()` to flatten a multi-dimensional array into a 1D␣
 ↪array. It's like squishing all the elements
into a single row.
'''
#Example
import numpy as np
arr=np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
print("Array:",arr,"with Dimension:",arr.ndim)

print("Array after reshaping:",np.reshape(arr,(4,3,1)),"with Dimension:",np.
 ↪reshape(arr,(4,3,1)).ndim)
```

```
Array: [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]] with Dimension: 2
Array after reshaping: [[[ 1]
  [ 2]
  [ 3]]

 [[ 4]
  [ 5]
  [ 6]]

 [[ 7]
  [ 8]
```

```
 [ 9]]

[[10]
 [11]
 [12]]] with Dimension: 3
```

[104]:
```
#15. What is broadcasting in Numpy?
'''
It allows you to perform operations on arrays with different shapes, without␣
 ↪explicitly having to match their shapes. It's
like NumPy automatically adjusts the dimensions of the arrays to make the␣
 ↪operation work.

For example, let's say you have a 1D array and you want to add a scalar value␣
 ↪to each element. With broadcasting, you can
simply write the addition operation, and NumPy will automatically apply it to␣
 ↪each element of the array.

Broadcasting can also be used to perform operations between arrays with␣
 ↪different shapes. NumPy will automatically adjust the
dimensions of the arrays to make the operation work, as long as the dimensions␣
 ↪are compatible.
'''
#For Example:
import numpy as np
arr=np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
#Broadcasting, add 5 to each element in the arr
arr+5
```

[104]:
```
array([[ 6,  7,  8,  9],
       [10, 11, 12, 13],
       [14, 15, 16, 17]])
```

[ ]: