Day13_Numpy_2

May 31, 2025

[1]:
```
import numpy as np
```

0.0.1 Recape

[2]:
```
 np.ones((3,2)) # (rows,columns)
```

[2]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

[3]:
```
np.zeros((3,2))
```

[3]:
```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

[4]:
```
# Generate a 3D NumPy array of shape (1, 12, 10) filled with random float␣
 ↪values between 0 and 1
# Shape breakdown:
# - 1: batch or outer dimension
# - 12: number of rows (e.g., time steps, features, etc.)
# - 10: number of values per row (e.g., features per step)
np.random.rand(1,12,10)
```

[4]:
```
array([[[0.99089099, 0.78984925, 0.75029947, 0.92998999, 0.44583764,
         0.36091074, 0.60728475, 0.69178029, 0.4044078 , 0.96735774],
        [0.08926494, 0.44438732, 0.22657739, 0.60638302, 0.26489343,
         0.3507991 , 0.9481399 , 0.25410295, 0.06328391, 0.57064479],
        [0.35191206, 0.78022962, 0.86396486, 0.38770471, 0.31291052,
         0.47421616, 0.95421921, 0.89758419, 0.478131  , 0.04750857],
        [0.49244656, 0.98164209, 0.33638084, 0.48727896, 0.02688551,
         0.72588051, 0.91699624, 0.31508812, 0.96681494, 0.38823905],
        [0.01634583, 0.18389109, 0.09403716, 0.51303832, 0.51922113,
         0.56797839, 0.12048259, 0.47057731, 0.46177379, 0.84680711],
        [0.83307271, 0.70865999, 0.8444242 , 0.54702924, 0.07104635,
         0.23185007, 0.41067411, 0.60234705, 0.790068  , 0.18696164],
        [0.1792901 , 0.18754784, 0.0099105 , 0.28928609, 0.4705313 ,
         0.82838958, 0.19563187, 0.32839344, 0.63018361, 0.80351412],
        [0.61175932, 0.38402999, 0.92540231, 0.03769851, 0.30240568,
         0.20771429, 0.00860893, 0.19633146, 0.25234107, 0.12669119],
        [0.90229687, 0.65984891, 0.67399561, 0.46218291, 0.80811059,
         0.81327216, 0.00730115, 0.34010601, 0.30824474, 0.84037013],
        [0.28633731, 0.65289412, 0.42205623, 0.01448668, 0.13717309,
         0.52729488, 0.00440527, 0.49821235, 0.67522114, 0.3125206 ],
        [0.75166741, 0.34107653, 0.97620474, 0.11294154, 0.60536899,
         0.12334869, 0.16319049, 0.43887991, 0.84763362, 0.02072149],
        [0.43820922, 0.01855302, 0.69970487, 0.94855489, 0.31270498,
         0.09805421, 0.89542786, 0.84665505, 0.13569218, 0.23719523]]])
```

```
[5]: np.random.randint(1,12,10) # you never get 12
```

```
[5]: array([ 4,  8, 10, 11,  8,  7, 11,  3,  7,  7])
```

```
[6]: np.random.randint(10,40,(10,10)) # 10 * 10 matrix with values betwuen 10 tO 40
```

```
[6]: array([[33, 31, 29, 18, 28, 21, 27, 11, 32, 14],
            [17, 15, 15, 39, 22, 21, 21, 16, 22, 18],
            [21, 13, 38, 25, 32, 33, 27, 34, 11, 29],
            [26, 36, 24, 17, 12, 11, 27, 15, 14, 35],
            [16, 21, 27, 24, 19, 35, 28, 28, 20, 33],
            [31, 28, 14, 36, 28, 38, 32, 30, 11, 37],
            [34, 29, 31, 27, 16, 35, 32, 16, 17, 39],
            [15, 24, 29, 35, 38, 22, 39, 28, 35, 20],
            [32, 34, 13, 29, 36, 22, 11, 32, 39, 31],
            [29, 15, 30, 32, 11, 32, 13, 23, 23, 15]])
```

1 np.reshape() in NumPy

1.0.1 Reshape Rules

Total elements must match:

You can only reshape an array if the total number of elements stays the same.

Example: np.arange(1,13) gives 12 elements → you can reshape to (3, 4), (4, 3), (2, 6), (6, 2), etc.

You cannot reshape it to (5, 5) because 5×5 = 25   12.

1.0.2 Multiplication Rule:

If your original array has n elements, then all new dimensions in reshape(a, b, c, …) must multiply to n.

Examples: np.arange(1, 13).reshape(2, 6) #   Valid

np.arange(1, 13).reshape(4, 3) #   Valid

np.arange(1, 13).reshape(3, 5) #   Invalid (3*5 = 15   12)

```
[7]: # Create a 1D array of numbers from 1 to 12
     np.arange(1,13) #This is 1D array
```

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

```
[8]: # Reshape it into a 3x4 2D array (3 rows, 4 columns)
     np.arange(1,13).reshape(3,4)
```

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

```
[9]: # Reshape it into a 4x3 2D array (4 rows, 3 columns)
     np.arange(1,13).reshape(4,3)
```

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

```
[10]: np.arange(1,13).reshape(5,4) # You can not 5*4 because its 20
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[10], line 1
----> 1 np.arange(1,13).reshape(5,4)

ValueError: cannot reshape array of size 12 into shape (5,4)
```

```
[11]: np.arange(1,13).reshape(6,2) #Yes because its 6*2 = 12
```

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

```
[12]: np.arange(1,13).reshape(12,1)
```

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

2 slicing

```
[13]: b = np.random.randint(10,20,(5,4))
      b
```

```
[13]: array([[17, 10, 13, 14],
             [16, 15, 14, 11],
             [17, 14, 10, 10],
             [14, 16, 19, 12],
             [16, 12, 19, 15]])
```

```
[14]: type(b)
```

```
[14]: numpy.ndarray
```

2.0.1 Tips:

: means “all”

start:end → slice rows

row, col → single value

[:, col] → full column

b[1:3] → Rows 1 & 2 Output: [[16, 15, 14, 11], [17, 14, 10, 10]]

b[1, 3] → Row 1, Column 3 (specific value) Output: 11

b[:, 2] → All rows, Column 2 Output: [13, 14, 10, 19, 19]

```
[15]: b[:]          # Returns the entire array
```

```
[15]: array([[17, 10, 13, 14],
             [16, 15, 14, 11],
             [17, 14, 10, 10],
             [14, 16, 19, 12],
             [16, 12, 19, 15]])
```

```
[16]: b[1:3]        # Returns rows 1 and 2 (index 1 to 2, as end index is excluded)
```

```
[16]: array([[16, 15, 14, 11],
             [17, 14, 10, 10]])
```

```
[17]: b[1, 3]      # Returns the element at row 1, column 3 (i.e., 2nd row, 4th column)
```

```
[17]: 11
```

```
[18]: b[2:4]       # Returns rows 2 and 3 (index 2 to 3)
```

```
[18]: array([[17, 14, 10, 10],
             [14, 16, 19, 12]])
```

```
[19]: b[1, -1]     # Returns the last element of row 1 (row index 1, column index -1)
```

```
[19]: 11
```

```
[20]: b[2:3]       # Returns only row 2 (keeps it in 2D form)
```

```
[20]: array([[17, 14, 10, 10]])
```

```
[21]: b[0:-2]      # Returns all rows from index 0 to 2 (excluding last 2 rows)
```

```
[21]: array([[17, 10, 13, 14],
             [16, 15, 14, 11],
             [17, 14, 10, 10]])
```

```
[22]: b[-5, 3]     # Returns element at row -5 (which is row 0), column 3
```

```
[22]: 14
```

3 Oprations in Numpy

```
[23]: arr2 = np.random.randint(0,100,(10,10))
```

```
[24]: arr2
```

```
[24]: array([[79, 58, 33, 52, 43, 97, 60, 30, 84, 99],
             [28, 69, 60, 19, 24, 97, 59,  1, 71, 42],
             [48, 23, 92, 40, 26, 50, 52, 60, 99, 62],
             [41, 42, 20, 45, 44, 62, 76, 58, 17, 39],
             [71, 77, 10, 43,  7, 17,  8, 11, 97, 87],
             [28, 73, 43,  2, 91,  1, 23,  1, 38, 55],
             [84,  8, 51, 87, 70, 67, 26, 13, 14, 64],
             [28,  8, 83, 26, 79, 81, 99, 10, 20, 60],
             [29, 99, 66, 46, 54,  6, 76,  0, 15, 68],
             [18,  3,  0, 26,  9,  1, 47, 65, 47, 81]])
```

[25]:

```
# arr2[::-1]
# This reverses the rows of the matrix - last row comes first, first row comes␣
 ↪last.
print(arr2[::-1])
```

```
[[18  3  0 26  9  1 47 65 47 81]
 [29 99 66 46 54  6 76  0 15 68]
 [28  8 83 26 79 81 99 10 20 60]
 [84  8 51 87 70 67 26 13 14 64]
 [28 73 43  2 91  1 23  1 38 55]
 [71 77 10 43  7 17  8 11 97 87]
 [41 42 20 45 44 62 76 58 17 39]
 [48 23 92 40 26 50 52 60 99 62]
 [28 69 60 19 24 97 59  1 71 42]
 [79 58 33 52 43 97 60 30 84 99]]
```

[26]:

```
# arr2[::-2]
# This reverses the rows and takes every 2nd row - from bottom to top, skipping␣
 ↪one row in between.
arr2[::-2]
```

[26]:

```
array([[18,  3,  0, 26,  9,  1, 47, 65, 47, 81],
       [28,  8, 83, 26, 79, 81, 99, 10, 20, 60],
       [28, 73, 43,  2, 91,  1, 23,  1, 38, 55],
       [41, 42, 20, 45, 44, 62, 76, 58, 17, 39],
       [28, 69, 60, 19, 24, 97, 59,  1, 71, 42]])
```

[27]:

```
# arr2[::-3]
# This reverses the rows and takes every 3rd row - from bottom to top, skipping␣
 ↪two rows in between.
print(arr2[::-3])
```

```
[[18  3  0 26  9  1 47 65 47 81]
 [84  8 51 87 70 67 26 13 14 64]
 [41 42 20 45 44 62 76 58 17 39]
 [79 58 33 52 43 97 60 30 84 99]]
```

[28]:

```
# arr2[0:10:3]
# This slices rows from index 0 to 9 (inclusive), taking every 3rd row - normal␣
 ↪order.
arr2[0:10:3]
```

[28]:

```
array([[79, 58, 33, 52, 43, 97, 60, 30, 84, 99],
       [41, 42, 20, 45, 44, 62, 76, 58, 17, 39],
       [84,  8, 51, 87, 70, 67, 26, 13, 14, 64],
       [18,  3,  0, 26,  9,  1, 47, 65, 47, 81]])
```

4 Numpy Array Functions

[29]:
```
arr2.max()
```

[29]:
```
99
```

[30]:
```
arr2.min()
```

[30]:
```
0
```

[31]:
```
arr2.mean()
```

[31]:
```
46.18
```

[32]:
```
arr2.mode() #You can not get output like this It dose not have mode function
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[32], line 1
----> 1 arr2.mode()

AttributeError: 'numpy.ndarray' object has no attribute 'mode'
```

[33]:
```
from numpy import *
a = median(arr2)
a
```

[33]:
```
45.5
```

5 Indexing

[34]:
```
mat = np.random.randint(0,100,(10,10))
```

[35]:
```
mat
```

[35]:
```
array([[44, 81, 78, 72, 97, 82, 92, 52, 93, 89],
       [92, 18,  3, 65, 57, 29, 30, 45, 21, 62],
       [ 3, 32, 68, 76, 81, 54, 80, 19, 21, 55],
       [27,  7, 99, 59, 52, 85,  1, 83, 62, 77],
       [43, 70, 20, 90, 80, 59, 12, 45,  8, 64],
       [56, 41, 96, 28, 99, 94, 14, 71, 74, 17],
       [53,  7, 35, 19, 72, 55, 78, 64, 69, 75],
       [91, 65, 46, 96, 56,  8, 43, 60,  6, 43],
       [38, 82, 19, 14, 17, 74, 29, 54, 36,  8],
       [16,  2, 89, 92, 31, 47, 81, 65, 84, 28]])
```

[36]:

```python
row = 5
col = 6

# Accessing a single element at row 5, column 6
print(mat[row, col])     # Output: 14
```

```
14
```

[37]:

```python
# Same as above - directly specifying the indices
print(mat[5, 6])         # Output: 14
```

```
14
```

[38]:

```python
# mat[7] returns the entire row at index 7 (row 8 in human terms)
print(mat[7])            # Output: [91 65 46 96 56  8 43 60  6 43]
```

```
[91 65 46 96 56  8 43 60  6 43]
```

[39]:

```python
# mat[:, col] returns the entire column at index 6
print(mat[:, col])       # Output: [92 30 80  1 12 14 78 43 29 81]
```

```
[92 30 80  1 12 14 78 43 29 81]
```

[40]:

```python
# mat[:, -1] returns the last column of the matrix
print(mat[:, -1])        # Output: [89 62 55 77 64 17 75 43  8 28]
```

```
[89 62 55 77 64 17 75 43  8 28]
```

[41]:

```python
# mat[row, :] returns the entire row at index 5
print(mat[row, :])       # Output: [56 41 96 28 99 94 14 71 74 17]
```

```
[56 41 96 28 99 94 14 71 74 17]
```

[42]:

```python
# mat[:, row] returns the column at index 5 (column 6 in human terms)
print(mat[:, row])       # Output: [82 29 54 85 59 94 55  8 74 47]
```

```
[82 29 54 85 59 94 55  8 74 47]
```

[43]:

```python
# mat[:row] returns all rows from index 0 to 4 (not including row 5)
print(mat[:row])         # Output: rows 0 to 4
```

```
[[44 81 78 72 97 82 92 52 93 89]
 [92 18  3 65 57 29 30 45 21 62]
 [ 3 32 68 76 81 54 80 19 21 55]
 [27  7 99 59 52 85  1 83 62 77]
 [43 70 20 90 80 59 12 45  8 64]]
```

[44]:

```python
# mat[2:6, 2:4] returns a submatrix:
# rows from index 2 to 5 (excluding row 6), columns 2 and 3
print(mat[2:6, 2:4])
# Output:
```

```
# [[68 76]
#  [99 59]
#  [20 90]
#  [96 28]]
```

```
[[68 76]
 [99 59]
 [20 90]
 [96 28]]
```

[45]:

```
# mat[2:3, 4:5] returns one element as a 1x1 submatrix - row 2, column 4
print(mat[2:3, 4:5])
# Output: [[81]]
```

```
[[81]]
```

[ ]:

```

```

6 Masking in NumPy

6.1 What is Masking?

Masking means applying a condition to a NumPy array to filter/select elements based on True/False values.

6.2 Why use it?

To easily:

Select specific elements (e.g. all > 50)

Filter arrays without writing loops

Perform element-wise operations

6.3 How does it work?

Create a condition: mat > 50 → returns a Boolean mask of the same shape as mat

Apply mask: mat[mat > 50] → returns only the values where the condition is True

[46]:

```
# Get the memory ID of the matrix (just for info)
print(id(mat))  # e.g., 140260631146816 (varies each time)
```

```
2543454478352
```

[47]:

```
# Create a Boolean mask: condition > 50
print(mat > 50)
# Output: a matrix of same shape with True/False
# Example:
# [[False  True  True  True  True  True  True  True  True  True]
#  [ True False False  True  True False False False False  True]
#    ...
# ]
```

```
[[False  True  True  True  True  True  True  True  True  True]
 [ True False False  True  True False False False False  True]
 [False False  True  True  True  True  True False False  True]
 [False False  True  True  True  True False  True  True  True]
 [False  True False  True  True  True False False False  True]
 [ True False  True False  True  True False  True  True False]
 [ True False False False  True  True  True  True  True  True]
 [ True  True False  True  True False False  True False False]
 [False  True False False False  True False  True False False]
 [False False  True  True False False  True  True  True False]]
```

[48]:

```
# Use mask to extract only values > 50
print(mat[mat > 50])
# Output: 1D array of values > 50
# [81 78 72 97 82 92 52 93 89 92 65 57 62 68 ... 54 55 89 92 81 65 84]
```

```
[81 78 72 97 82 92 52 93 89 92 65 57 62 68 76 81 54 80 55 99 59 52 85 83
 62 77 70 90 80 59 64 56 96 99 94 71 74 53 72 55 78 64 69 75 91 65 96 56
 60 82 74 54 89 92 81 65 84]
```

[50]:

```
# You can apply other conditions too:
print(mat[mat != 50])  # all elements not equal to 50
```

```
[44 81 78 72 97 82 92 52 93 89 92 18  3 65 57 29 30 45 21 62  3 32 68 76
 81 54 80 19 21 55 27  7 99 59 52 85  1 83 62 77 43 70 20 90 80 59 12 45
  8 64 56 41 96 28 99 94 14 71 74 17 53  7 35 19 72 55 78 64 69 75 91 65
 46 96 56  8 43 60  6 43 38 82 19 14 17 74 29 54 36  8 16  2 89 92 31 47
 81 65 84 28]
```

[51]:

```
print(mat[mat == 81])  # all elements equal to 81
```

```
[81 81 81]
```

[52]:

```
print(mat[mat < 30])   # elements less than 30
```

```
[18  3 29 21  3 19 21 27  7  1 20 12  8 28 14 17  7 19  8  6 19 14 17 29
  8 16  2 28]
```