Numpy_Tutorial

July 1, 2022

Refer to https://numpy.org/doc/stable/user/basics.types.html for data types in NumPy

[1]:
```
import numpy as np
Array_a  = np.array([10,12,14,16,18,20])
print(Array_a)
print(Array_a.dtype)
print(Array_a.dtype.itemsize)
```

```
[10 12 14 16 18 20]
int32
4
```

[2]:
```
Array_b= np.array(['Apple','Banana','Cherry'])
print(Array_b)
print(Array_b.dtype)
print(Array_b.dtype.itemsize)
```

```
['Apple' 'Banana' 'Cherry']
<U6
24
```

[3]:
```
Array_c = np.array(["2022-10-29", "2010-02-27", "2011-11-18"])
print(Array_c)
print(Array_c.dtype)
print(Array_c.dtype.itemsize)
```

```
['2022-10-29' '2010-02-27' '2011-11-18']
<U10
40
```

[4]:
```
Array_d = Array_c.astype("M")
print(Array_d.dtype)
print(Array_d.dtype.itemsize)
```

```
datetime64[D]
8
```

Refer to https://www.geeksforgeeks.org/change-data-type-of-given-numpy-array/ for astype()

[5]:
```python
Array_e = np.array(["2010-10-04", "2007-05-06", "2022-11-04"], dtype = "M")
print(Array_e)
print(Array_e.dtype)
print(Array_e.dtype.itemsize)
```

```
['2010-10-04' '2007-05-06' '2022-11-04']
datetime64[D]
8
```

[6]:
```python
# astype() is used to change the datatype of array

arr = np.array([5,10,15,20,25])
print(arr)
print(arr.dtype)

# change the datatype to 'float64'
arr = arr.astype('float64')

# Print the array after changing the data type
print(arr)

# Also print the data type
print(arr.dtype)
```

```
[ 5 10 15 20 25]
int32
[ 5. 10. 15. 20. 25.]
float64
```

[7]:
```python
Array_f = np.array(["2000-10-04", "2018-04-06", "2020-11-04"], dtype = "M")
print(Array_f)
print(Array_f.dtype)
print(Array_f.dtype.itemsize)
```

```
['2000-10-04' '2018-04-06' '2020-11-04']
datetime64[D]
8
```

[8]:
```python
nums_list = [10,12,14,16,20]
nums_array = np.array(nums_list)
type(nums_array)
```

[8]:
```
numpy.ndarray
```

[9]:
```python
row1 = [10,12,13]
row2 = [45,32,16]
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
```

```
nums_2d.shape
```

[9]:
```
(3, 3)
```

[10]:
```
nums_arr = np.arange(5,11)
print(nums_arr)
```

```
[ 5  6  7  8  9 10]
```

[11]:
```
nums_arr = np.arange(5,12,2)
print(nums_arr)
```

```
[ 5  7  9 11]
```

[12]:
```
ones_array = np.ones(6)
print(ones_array)
```

```
[1. 1. 1. 1. 1. 1.]
```

[13]:
```
# We can create a two-dimensional array of all ones by passing the number of␣
↪rows and columns as the
# first and second parameters of the ones() method

ones_array = np.ones((5,4))
print(ones_array)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[14]:
```
# The zeros() method can be used to create a NumPy array of all zeros.

zeros_array = np.zeros(6)
print(zeros_array)
```

```
[0. 0. 0. 0. 0. 0.]
```

[15]:
```
# We can create a two-dimensional array of all zeros by passing the number of␣
↪rows and columns as the first
# and second parameters of the zeros() method

zeros_array = np.zeros((5,4))
print(zeros_array)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
```

```
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

[16]:
```python
# The eye() method is used to create an identity matrix in the form of a␣
↪twodimensional
# NumPy array. An identity matrix contains 1s along the diagonal,
# while the rest of the elements are 0 in the array.

eyes_array = np.eye(5)
print(eyes_array)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

[17]:
```python
# The random.rand() function from the NumPy module can be used to create
# a NumPy array with uniform distribution.
import numpy as np
uniform_random = np.random.rand(4, 5)
print(uniform_random)
```

```
[[0.15648032 0.65871046 0.2615175  0.44739539 0.10675402]
 [0.66749578 0.27059451 0.65178849 0.26888006 0.14073759]
 [0.45542265 0.24530804 0.2579981  0.47324362 0.35213343]
 [0.53269472 0.34971592 0.92585562 0.23398064 0.09969918]]
```

[18]:
```python
# The random.randn() function from the NumPy module can be used to create
# a NumPy array with normal distribution, as shown in the following example.

normal_random = np.random.randn(4, 5)
print(uniform_random)
```

```
[[0.15648032 0.65871046 0.2615175  0.44739539 0.10675402]
 [0.66749578 0.27059451 0.65178849 0.26888006 0.14073759]
 [0.45542265 0.24530804 0.2579981  0.47324362 0.35213343]
 [0.53269472 0.34971592 0.92585562 0.23398064 0.09969918]]
```

[19]:
```python
# Finally, the random.randint() function from the NumPy module can be used
# to create a NumPy array with random integers between a certain range. The
# first parameter to the randint() function specifies the lower bound, the
# second parameter specifies the upper bound, and the last parameter specifies
# the number of random integers to generate between the range. The following
# example generates five random integers between 5 and 50.

integer_random = np.random.randint(10, 50, 5)
```

```python
print(integer_random)
```

```
[22 30 37 15 45]
```

[20]:
```python
# Depending on the dimensions, there are various ways to display the NumPy
# arrays. The simplest way to print a NumPy array is to pass the array to the␣
↪print
# method, as you have already seen in the previous section. An example is
# given below:

my_array = np.array([10,12,14,16,20,25])
print(my_array)
```

```
[10 12 14 16 20 25]
```

[21]:
```python
# You can also use loops to display items in a NumPy array. It is a good idea to
# know the dimensions of a NumPy array before printing the array on the
# console. To see the dimensions of a NumPy array, you can use the ndim
# attribute, which prints the number of dimensions for a NumPy array. To see
# the shape of your NumPy array, you can use the shape attribute.

print(my_array.ndim)
print(my_array.shape)
```

```
1
(6,)
```

[22]:
```python
# To print items in a one-dimensional NumPy array, you can use a single
# for each loop, as shown below:

for item in my_array:
    print(item)
```

```
10
12
14
16
20
25
```

[23]:
```python
# The following script creates a two-dimensional NumPy array with four rows
# and five columns. The array contains random integers between 1 and 10. The
# array is then printed on the console.

integer_random = np.random.randint(1,11, size=(4, 5))
print(integer_random)
```

```
[[8 4 8 9 5]
```

 [3 6 2 7 7]
 [1 7 7 4 4]
 [8 1 6 5 5]]

[24]:
```python
# Let's now try to see the number of dimensions and shape of our NumPy
# array.

print(integer_random.ndim)
print(integer_random.shape)
```

```
2
(4, 5)
```

[25]:
```python
# To traverse through items in a two-dimensional NumPy array, you need two
# for each loops: one for each row and the other for each column in the row.
# Let's first use one for loop to print items in our one-dimensional NumPy␣
↪array.

my_array = np.array([10,12,14,16,20,25])
for item in my_array:
    print(item)
```

```
10
12
14
16
20
25
```

[26]:
```python
# To traverse through all the items in the two-dimensional array, you can use
# the nested foreach loop, as follows:

for rows in integer_random:
    for column in rows:
        print(column)
```

```
8
4
8
9
5
3
6
2
7
7
1
7
```

```
7
4
4
8
1
6
5
5
```

[27]:
```python
# To add the items into a NumPy array, you can use the append() method from
# the NumPy module. First, you need to pass the original array and the item
# that you want to append to the array to the append() method. The append()
# method returns a new array that contains newly added items appended to the
# end of the original array. The following script adds a text item "Yellow"
# to an existing array with three items.

my_array = np.array(["Red", "Green", "Orange"])
print(my_array)
extended = np.append(my_array, "Yellow")
print(extended)
```

```
['Red' 'Green' 'Orange']
['Red' 'Green' 'Orange' 'Yellow']
```

[28]:
```python
# In addition to adding one item at a time, you can also append an array of
# items to an existing array. The method remains similar to appending a single
# item. You just have to pass the existing array and the new array to the
# append() method, which returns a concatenated array where items from the
# new array are appended at the end of the original array.

my_array = np.array(["Red", "Green", "Orange"])
print(my_array)
extended = np.append(my_array, ["Yellow", "Pink"])
print(extended)
```

```
['Red' 'Green' 'Orange']
['Red' 'Green' 'Orange' 'Yellow' 'Pink']
```

[29]:
```python
# To add items in a two-dimensional NumPy array, you have to specify
# whether you want to add the new item as a row or as a column. To do so, you
# can take the help of the axis attribute of the append method.
# Let's first create a 3 x 3 array of all zeros.
import numpy as np
zeros_array = np.zeros((3,3))
print(zeros_array)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
```

```
 [0. 0. 0.]]
```

[30]:
```python
# To add a new row in the above 3 x 3 array, you need to pass the original array
# to the new array in the form of a row vector and the axis attribute to the
# append() method. To add a new array in the form of a row, you need to set 0
# as the value for the axis attribute. Here is an example script.

zeros_array = np.zeros((3,3))
print(zeros_array)
print("Extended Array")
extended = np.append(zeros_array, [[1, 2, 3]], axis = 0)
print(extended)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
Extended Array
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [1. 2. 3.]]
```

[31]:
```python
# To append a new array as a column in the existing 2-D array, you need to set
# the value of the axis attribute to 1.

zeros_array = np.zeros((3,3))
print(zeros_array)
print("Extended Array")
extended = np.append(zeros_array, [[1],[2],[3]], axis = 1)
print(extended)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
Extended Array
[[0. 0. 0. 1.]
 [0. 0. 0. 2.]
 [0. 0. 0. 3.]]
```

[32]:
```python
# To delete an item from an array, you may use the delete() method. You need
# to pass the existing array and the index of the item to be deleted to the
# delete() method. The following script deletes an item at index 1 (second item)
# from the my_array array.

my_array = np.array(["Red", "Green", "Orange"])
print(my_array)
print("After deletion")
updated_array = np.delete(my_array, 1)
```

```
print(updated_array)
```

```
['Red' 'Green' 'Orange']
After deletion
['Red' 'Orange']
```

[33]:
```
# If you want to delete multiple items from an array, you can pass the item
# indexes in the form of a list to the delete() method. For example, the
# following script deletes the items at index 1 and 2 from the NumPy array
# named my_array.

my_array = np.array(["Apple", "Banana", "Cherry"])
print(my_array)
print("After deletion")
updated_array = np.delete(my_array, [1,2])
print(updated_array)
```

```
['Apple' 'Banana' 'Cherry']
After deletion
['Apple']
```

[34]:
```
# You can delete a row or column from a 2-D array using the delete method.
# However, just as you did with the append() method for adding items, you
# need to specify whether you want to delete a row or column using the axis
# attribute.
# The following script creates an integer array with four rows and five
# columns. Next, the delete() method is used to delete the row at index 1
# (second row). Notice here that to delete the array, the value of the axis
# attribute is set to 0.
# Format: np.delete(Array_name,index,axis) where axis = 0 for rows and axis = 1␣
 ↪for columns

integer_random = np.random.randint(1,11, size=(4, 5))
print(integer_random)
print("\n After deletion \n")
updated_array = np.delete(integer_random, 1, axis = 0)
print(updated_array)
```

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

 After deletion

[[1 7 5 7 8]
 [6 4 1 9 8]
 [7 1 7 7 4]]
```

[35]:
```
# Finally, to delete a column, you can set the value of the axis attribute to␣
↪1, as shown below:

integer_random = np.random.randint(1,11, size=(4, 5))
print(integer_random)
print("\n After deletion \n")
updated_array = np.delete(integer_random, 1, axis = 1)
print(updated_array)
```

```
[[ 3  6  3  7  3]
 [ 6  8  3  6  4]
 [ 4  5  9  4  5]
 [10 10  5  4  7]]

 After deletion

[[ 3  3  7  3]
 [ 6  3  6  4]
 [ 4  9  4  5]
 [10  5  4  7]]
```

[36]:
```
# You can sort NumPy arrays of various types. Numeric arrays are sorted by
# default in ascending order of numbers. On the other hand, text arrays are
# sorted alphabetically.
# To sort an array in NumPy, you may call the np.sort() function and pass it to
# your NumPy array. The following script shows how to sort a NumPy array of
# 10 random integers between 1 and 20.

print("unsorted array")

my_array = np.random.randint(1,20,10)
print(my_array)
print("\nsorted array \n")
sorted_array = np.sort(my_array)
print(sorted_array)
```

```
unsorted array
[ 9 19 11 17  4  4 19  5  8  2]

sorted array

[ 2  4  4  5  8  9 11 17 19 19]
```

[37]:
```
# As mentioned earlier, text arrays are sorted in alphabetical order. Here is an
# example of how you can sort a text array with the NumPy sort() method.

print("unsorted array")
```

```
my_array = np.array(["Red", "Green", "Blue", "Yellow"])
print(my_array)
print("\nsorted array \n")
sorted_array = np.sort(my_array)
print(sorted_array)
```

```
unsorted array
['Red' 'Green' 'Blue' 'Yellow']

sorted array

['Blue' 'Green' 'Red' 'Yellow']
```

[38]:
```
# Finally, Boolean arrays are sorted in a way that all the False values appear
# first in an array. Here is an example of how you can sort the Boolean arrays
# in NumPy.

import numpy as np
print("unsorted array")
my_array = np.array([False, True, True, False, False, True, False, True])
print(my_array)
print("\nSorted array")
sorted_array = np.sort(my_array)
print(sorted_array)
```

```
unsorted array
[False  True  True False False  True False  True]

Sorted array
[False False False False  True  True  True  True]
```

[39]:
```
# NumPy also allows you to sort two-dimensional arrays. In two-dimensional
# arrays, each item itself is an array. The sort() function sorts an item in␣
↪each
# individual array in a two-dimensional array.
# The script below creates a two-dimensional array of shape (4,6) containing
# random integers between 1 to 20. The array is then sorted via the np.sort()
# method.

print("unsorted array")
my_array = np.random.randint(1,20, size = (4,6))
print(my_array)
print("\nSorted array\n")
sorted_array = np.sort(my_array)
print(sorted_array)
```

```
unsorted array
[[ 6  7 11 17  3 17]
```

```
 [19 13 17 15  8  5]
 [15 12  6  2 19  4]
 [ 2  3 12 13 12 12]]

Sorted array

[[ 3  6  7 11 17 17]
 [ 5  8 13 15 17 19]
 [ 2  4  6 12 15 19]
 [ 2  3 12 12 12 13]]
```

[40]:
```
# You can also sort an array in descending order. To do so, you can first sort␣
↪an
# array in ascending order via the sort() method. Next, you can pass the sorted
# array to the flipud() method, which reverses the sorted array and returns the
# array sorted in descending order. Here is an example of how you can sort an
# array in descending order.
import numpy as np
print("unsorted array")
my_array = np.random.randint(1,20,10)
print(my_array)
print("\nsorted array")
sorted_array = np.sort(my_array)
reverse_sorted = np.flipud(sorted_array)
print(reverse_sorted)
```

```
unsorted array
[ 4  5  2  4 14 19 12  1 19 16]

sorted array
[19 19 16 14 12  5  4  4  2  1]
```

[41]:
```
# You can also modify the shape of a NumPy array. To do so, you can use the
# reshape() method and pass it the new shape for your NumPy array.
# In this section, you will see how to reshape a NumPy array from lower to
# higher dimensions and vice versa.
# The following script defines a one-dimensional array of 10 random integers
# between 1 and 20. The reshape() function reshapes the array into the shape
# (2,5).


print("one-dimensional array")
one_d_array = np.random.randint(1,20,10)
print(one_d_array)
print("\ntwo-dimensional array\n")
two_d_array = one_d_array.reshape(2,5)
print(two_d_array)
```

```
one-dimensional array
[19 14  8  6 12 15  7 12 10 12]

two-dimensional array

[[19 14  8  6 12]
 [15  7 12 10 12]]
```

[42]:

```
# It is important to mention that the product of the rows and columns of the
# original array must match the value of the product of rows and columns of
# the reshaped array. For instance, the shape of the original array in the last
# script was (10,) with product 10. The product of the rows and columns in the
# reshaped array was also 10 (2 x 5)
# You can also call the reshape() function directly from the NumPy module and
# pass it the array to be reshaped as the first argument and the shape tuple as
# the second argument. Here is an example which converts an array of shape
# (10,) to (2,5).


print("one-dimensional array")
one_d_array = np.random.randint(1,20,10)
print(one_d_array)
print("\ntwo-dimensional array")
two_d_array = np.reshape(one_d_array,(2,5))
print(two_d_array)
```

```
one-dimensional array
[ 5 14  6 10 17 12 11  5 13 17]

two-dimensional array
[[ 5 14  6 10 17]
 [12 11  5 13 17]]
```

[43]:

```
# Let's see another example of reshaping a NumPy array from lower to higher
# dimensions. The following script defines a NumPy array of shape (4,6). The␣
↪original ar
# is then reshaped to a three-dimensional array of shape (3, 4, 2). Notice here
# again that the product of dimensions of the original array (4 x 6) and the
# reshaped array (3 x 4 x 2) is the same, i.e., 24.

print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\nthree-dimensional array")
three_d_array = np.reshape(two_d_array,(3,4,2))
print(three_d_array)
```

```
two-dimensional array
```

```
[[14 15 17  1  1 16]
 [ 7 10 18 13 18 16]
 [ 7  3  7 17 16 11]
 [15 11  2  2  1  8]]

three-dimensional array
[[[14 15]
  [17  1]
  [ 1 16]
  [ 7 10]]

 [[18 13]
  [18 16]
  [ 7  3]
  [ 7 17]]

 [[16 11]
  [15 11]
  [ 2  2]
  [ 1  8]]]
```

[44]:

```python
# Let's try to reshape a NumPy array in a way that the product of dimensions
# does not match. In the script below, the shape of the original array is (4,6).
# Next, you try to reshape this array to the shape (1,4,2). In this case, since
# the product of dimensions of the original and the reshaped array don't match,
# you will see an error in the output.

print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\nthree-dimensional array")
three_d_array = np.reshape(two_d_array,(1,4,2))
print(three_d_array)
```

```
two-dimensional array
[[ 9 13  9 15 10  9]
 [13  6 17  1  3 16]
 [ 2 16 18  2 16 19]
 [12  6  4  4 13  3]]

three-dimensional array
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6004/636047444.py in <module>
      9 print(two_d_array)
     10 print("\nthree-dimensional array")
---> 11 three_d_array = np.reshape(two_d_array,(1,4,2))
```

```
     12 print(three_d_array)

<__array_function__ internals> in reshape(*args, **kwargs)

~\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in reshape(a, newshape,␣
↪order)
    297            [5, 6]])
    298     """
--> 299     return _wrapfunc(a, 'reshape', newshape, order=order)
    300
    301

~\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in _wrapfunc(obj,␣
↪method, *args, **kwds)
     56
     57     try:
---> 58         return bound(*args, **kwds)
     59     except TypeError:
     60         # A TypeError occurs if the object does have such a method in its

ValueError: cannot reshape array of size 24 into shape (1,4,2)
```

[45]:
```
# Let's now see a few examples of reshaping NumPy arrays from higher to
# lower dimensions. In the script below, the original array is of shape (4,6)
# while the new array is of shape (24). The reshaping, in this case, will be␣
↪successful
# since the product of dimensions for original and reshaped arrays is the same.

print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\none-dimensional array")
one_d_array = two_d_array.reshape(24)
print(one_d_array)
```

```
two-dimensional array
[[ 6 17  2 15 13 14]
 [15  7  9  4  1 12]
 [12 15  1  1 11  6]
 [ 1  7 15 13  2  7]]

one-dimensional array
[ 6 17  2 15 13 14 15  7  9  4  1 12 12 15  1  1 11  6  1  7 15 13  2  7]
```

[46]:
```
# Finally, to convert an array of any dimensions to a flat, one-dimensional
# array, you will need to pass -1 as the argument for the reshaped function, as
```

```
# shown in the script below, which converts a two-dimensional array to a one␣
↪dimensional
# array.

print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\none-dimensional array")
one_d_array = two_d_array.reshape(-1)
print(one_d_array)
```

```
two-dimensional array
[[15 16  7 10 14 14]
 [10 16 19  7 14 12]
 [16  2  4 13  3 15]
 [ 3 13  5  8  6  1]]

one-dimensional array
[15 16  7 10 14 14 10 16 19  7 14 12 16  2  4 13  3 15  3 13  5  8  6  1]
```

[47]:

```
# Similarly, the following script converts a three-dimensional array to a one␣
↪dimensiona
# array.

print("two-dimensional array")
three_d_array = np.random.randint(1,20, size = (4,2,6))
print(three_d_array)
print("\non-dimensional array")
one_d_array = three_d_array .reshape(-1)
print(one_d_array)
```

```
two-dimensional array
[[[18  1 19  5  6  5]
  [ 7  6  9  9 13 15]]

 [[ 8 13 17  2 13 12]
  [12  2 18  1  7  4]]

 [[ 7  9 13 10  3 10]
  [ 3  4 11  5 10  9]]

 [[19 15  6 13  1  2]
  [17  7  9  9  5  1]]]

on-dimensional array
[18  1 19  5  6  5  7  6  9  9 13 15  8 13 17  2 13 12 12  2 18  1  7  4
  7  9 13 10  3 10  3  4 11  5 10  9 19 15  6 13  1  2 17  7  9  9  5  1]
```

[48]:
```
# NumPy arrays can be indexed and sliced. Slicing an array means dividing an
# array into multiple parts. NumPy arrays are indexed just like normal lists.
# Indexes in NumPy arrays start from 0, which means that the first item of a
# NumPy array is stored at the 0th index. The following script creates a simple
# NumPy array of the first 10 positive integers.

s = np.arange(1,11)
print(s)
```

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

[49]:
```
# The item at index one can be accessed as follows:

s = np.arange(1,11)
print(s)
print(s[1])
```

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

[50]:
```
# To slice an array, you have to pass the lower index, followed by a colon and
# the upper index. The items from the lower index (inclusive) to the upper
# index (exclusive) will be filtered. The following script slices the array "s"
# from the 1st index to the 9th index. The elements from index 1 to 8 are
# printed in the output.

s = np.arange(1,11)
print(s)
print(s[1:9])
```

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

[51]:
```
# If you specify only the upper bound, all the items from the first index to the
# upper bound are returned. Similarly, if you specify only the lower bound, all
# the items from the lower bound to the last item of the array are returned.

s = np.arange(1,11)
print(s)
print(s[:5])
print(s[5:])
```

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

[52]:
```
# Array slicing can also be applied on a two-dimensional array. To do so, you
# have to apply slicing on arrays and columns separately. A comma separates
# the rows and columns slicing. In the following script, the rows from the
# first and second indexes are returned, while all the columns are returned.
# You can see the first two complete rows in the output.

row1 = [10,12,13]
row2 = [45,32,16]
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
print(nums_2d[:2,:])
```

```
[[10 12 13]
 [45 32 16]]
```

[53]:
```
# the following script returns all the rows but only the first two
# columns.

row1 = [10,12,13]
row2 = [45,32,16]
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
print(nums_2d[:,:2])
```

```
[[10 12]
 [45 32]
 [45 32]]
```

[54]:
```
# Let's see another example of slicing. Here, we will slice the rows from row
# one to the end of rows and column one to the end of columns. (Remember,
# row and column numbers start from 0.) In the output, you will see the last
# two rows and the last two columns.

row1 = [10,12,13]
row2 = [45,32,16]
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
print(nums_2d[1:,1:])
```

```
[[32 16]
 [32 16]]
```

[55]:
```
# Broadcasting allows you to perform various operations between NumPy
# arrays of different shapes. This is best explained with the help of an
 ↪example.
# In the script below, you define two arrays: a one-dimensional array of three
# items and another scaler array that contains only one item. Next, the two
```

```
# arrays are added. Finally, in the output, you will see that the scaler value,␣
↪i.e.,
# 10 is added to all the items in the array that contains three items. This is␣
↪an
# amazing ability and is extremely useful in many scenarios, such as machine
# learning and artificial neural networks.

array1 = np.array ([14,25,31])
array2 = np.array([10])
result = array1 + array2
print(result)
```

```
[24 35 41]
```

[56]:

```
# Let's see another example of broadcasting. In the script below, you add a
# two-dimensional array with three rows and four columns to a scaler array
# with one item. In the output, you will see that the scaler value, i.e., 10,␣
↪will be
# added to all the 12 items in the first array, which contains three rows and␣
↪four
# columns.

array1 = np.random.randint(1,20, size = (3,4))
print(array1)
array2 = np.array([10])
print("after broadcasting")
result = array1 + array2
print(result)
```

```
[[17  7  5  3]
 [ 3 10 12 14]
 [ 2  3  5 12]]
after broadcasting
[[27 17 15 13]
 [13 20 22 24]
 [12 13 15 22]]
```

[57]:

```
# You can also use broadcasting to perform operations between twodimensional
# and one-dimensional arrays.
# For example, the following script creates two arrays: a two-dimensional array
# of three rows and four columns and a one-dimensional array of one row and
# four columns. If you perform addition between these two rows, you will see
# that the values in a one-dimensional array will be added to the corresponding
# columns' values in all the rows in the two-dimensional array.
# For instance, the first row in the two-dimensional array contains values 4, 6,
# 1, and 2. When you add the one-dimensional array with values 5, 10, 20, and
```

```
# 25 to it, the first row of the two-dimensional array becomes 9, 16, 21, and␣
↪27.

array1 = np.random.randint(1,20, size = (3,4))
print(array1)
array2 = np.array([5,10,20,25])
print("after broadcasting")
result = array1 + array2
print(result)
```

```
[[10  9 17  4]
 [11  3  5 11]
 [ 1 14  7 12]]
after broadcasting
[[15 19 37 29]
 [16 13 25 36]
 [ 6 24 27 37]]
```

[58]:
```
# Finally, let's see an example where an array with three rows and one column
# is added to another array with three rows and four columns. Since, in this
# case, the values of row match, therefore, in the output, for each column in␣
↪the
# two-dimensional array, the values from the one- dimensional array are added
# row-wise. For instance, the first column in the two-dimensional array
# contains the values 10, 19, and 11. When you add the one-dimensional array
# (5, 10, 20) to it, the new column value becomes 15, 29, and 31.

array1 = np.random.randint(1,20, size = (3,4))
print(array1)
array2 = np.array([[5],[10],[20]])
print("after broadcasting")
result = array1 + array2
print(result)
```

```
[[ 6 13  6  6]
 [ 9  5  8  6]
 [ 3 18 12 16]]
after broadcasting
[[11 18 11 11]
 [19 15 18 16]
 [23 38 32 36]]
```

[59]:
```
# There are two main ways to copy an array in NumPy. You can either copy
# the contents of the original array, or you can copy the reference to the
# original array into another array.
# To copy the contents of the original array into a new array, you can call the
# copy() function on the original array. Now, if you modify the contents of the
```

```
# new array, the contents of the original array are not modified.
# For instance, in the script below, in the copied array2, the item at index 1␣
↪is
# updated. However, when you print the original array, you see that the index
# one for the original array is not modified.

array1 = np.array([10,12,14,16,18,20])
array2 = array1.copy()
array2[1] = 20
print(array1)
print(array2)
```

```
[10 12 14 16 18 20]
[10 20 14 16 18 20]
```

[60]:
```
# The other method to copy an array in Python is via the view() method.
# However, with the view method, if the contents of a new array are modified,
# the original array is also modified. Here is an example:

array1 = np.array([10,12,14,16,18,20])
array2 = array1.view()
array2[1] = 20
print(array1)
print(array2)
```

```
[10 20 14 16 18 20]
[10 20 14 16 18 20]
```

[61]:
```
# You can save and load NumPy arrays to and from your local drive.
# To save a NumPy array, you need to call the save() method from the NumPy
# module and pass it the file name for your NumPy as the first argument, while
# the array object itself as the second argument. Here is an example:

array1 = np.array([10,12,14,16,18,20])
np.save("array1",array1)
```

[62]:
```
# The save() method saves a NumPy array in "NPY" file format. You can also
# save a NumPy array in the form of a text file, as shown in the following
# script:

array1 = np.array([10,12,14,16,18,20])
np.savetxt("my_array.txt", array1)
```

[63]:
```
# To load a NumPy array in the "NPY" format, you can use the load() method,
# as shown in the following example.

a = np.array(([i + j for i in range(5)
```

```
for j in range(5)]))
# a is printed.
print("a is:")
print(a)
np.save('file', a)
print("the array is saved in the file.npy")
# the array is saved in the file.npy
b = np.load('file.npy')
# the array is loaded into b
print("b is:")
print(b)
# b is printed from file.npy
print("b is printed from file.npy")
```

```
a is:
[0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8]
the array is saved in the file.npy
b is:
[0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8]
b is printed from file.npy
```

[64]:
```
# On the other hand, if your NumPy array is stored in the text form, you may
# use the loadtxt() method to load such a NumPy array.
import numpy as np
loaded_array = np.loadtxt("my_array.txt")
print(loaded_array)
```

```
[10. 12. 14. 16. 18. 20.]
```

[65]:
```
# NumPy arrays contain various functionalities for performing statistical
# operations, such as finding the mean, median, and standard deviations of
# items in a NumPy array.
# To find the mean or average of all the items in a NumPy array, you need to
#pass the array to the mean() method of the NumPy module. Here is an
# example:

my_array = np.array([2,4,8,10,12])
print(my_array)
print("mean:")
print(np.mean(my_array))
```

```
[ 2  4  8 10 12]
mean:
7.2
```

[66]:
```
# You can also find the mean in a two-dimensional NumPy array across rows
# and columns. To find the mean across columns, you need to pass 1 as the
```

```
# value for the axis parameter of the mean method. Similarly, to find the mean
# across rows, you need to pass 0 as the parameter value.
# The following script finds the mean of a two-dimensional array containing
# two rows and three columns across rows and columns.

my_array = np.random.randint(1,20, size = (2,3))
print(my_array)
print("mean:")
print(np.mean(my_array, axis = 1))
print(np.mean(my_array, axis = 0))
```

```
[[14 16 11]
 [14 17  9]]
mean:
[13.66666667 13.33333333]
[14.  16.5 10. ]
```

[67]:
```
# The median() method from the NumPy module is used to find the median
# value in a NumPy array. Here is an example.

my_array = np.array([2,4,8,10,12])
print(my_array)
print("median:")
print(np.median(my_array))
```

```
[ 2  4  8 10 12]
median:
8.0
```

[68]:
```
# Similarly, to find the median values across columns and rows in a␣
↪twodimensional
# array, you need to pass 1 and 0, respectively, as the values for
# the axis attribute of the median method.
# The following script finds the median values across rows and columns for a
# two-dimensional array with three rows and five columns.

my_array = np.random.randint(1,20, size = (3,5))
print(my_array)
print("median:")
print(np.median(my_array, axis = 1))
print(np.median(my_array, axis = 0))
```

```
[[13 17 19 10  2]
 [14 11 17  8 10]
 [17 18  5  4  4]]
median:
[13. 11.  5.]
[14. 17. 17.  8.  4.]
```

[69]:
```
# The max() function returns the maximum value from the array, while the
# min() function returns the minimum value.
# The following script returns the minimum value in a NumPy array using the
# min() method.

my_array = np.array([2,4,8,10,12])
print(my_array)
print("min value:")
print(np.amin(my_array))
```

```
[ 2  4  8 10 12]
min value:
2
```

[70]:
```
# You can get the minimum values across all rows or columns in a twodimensional
# NumPy array by passing 0 or 1 as values for the axis attribute of
# the min() method. The value of 1 for the axis attribute returns the minimum
# values across all columns, whereas a value of 0 returns the minimum values
# across all rows.

my_array = np.random.randint(1,20, size = (3,4))
print(my_array)
print("min:")
print(np.amin(my_array, axis = 1))
print(np.amin(my_array, axis = 0))
```

```
[[ 5 17 16 10]
 [10 18 13 16]
 [ 6 12 14 12]]
min:
[ 5 10  6]
[ 5 12 13 10]
```

[71]:
```
# To get the maximum value from a NumPy array, you may use the max()
# method, as shown in the script below:

my_array = np.array([2,4,8,10,12])
print(my_array)
print("max value:")
print(np.amax(my_array))
```

```
[ 2  4  8 10 12]
max value:
12
```

[72]:
```
# Like the min() method, which returns the minimum value, you can get the
# maximum values across all rows or columns in a two-dimensional NumPy
# array by passing 0 or 1 as values for the axis attribute of the max() method.
```

```python
# The value of 1 for the axis attribute returns the minimum values across all
# columns, whereas a value of 0 returns the minimum values across all rows.

my_array = np.random.randint(1,20, size = (3,4))
print(my_array)
print("max:")
print(np.amax(my_array, axis = 1))
print(np.amax(my_array, axis = 0))
```

```
[[ 5 14  6 12]
 [16  7  7  8]
 [10 17 12 13]]
max:
[14 16 17]
[16 17 12 13]
```

[73]:
```python
# The standard deviation of a NumPy array can be found via the std() method.
# Here is an example:

my_array = np.array([2,4,8,10,12])
print(my_array)
print("std value:")
print(np.std(my_array))
```

```
[ 2  4  8 10 12]
std value:
3.7094473981982814
```

[74]:
```python
# To find the standard deviation across rows and columns in a two-dimensional
# NumPy array, you need to call the std() method. The value of 1 for the axis
# attribute returns the minimum list of minimum values across all columns,
# whereas a value of 0 returns minimum values across all rows.

my_array = np.random.randint(1,20, size = (3,4))
print(my_array)
print("std-dev:")
print(np.std(my_array, axis = 1))
print(np.std(my_array, axis = 0))
```

```
[[10 15 12  1]
 [ 8  6 14  5]
 [ 2  1 18 13]]
std-dev:
[5.22015325 3.49106001 7.22841615]
[3.39934634 5.79271573 2.49443826 4.98887652]
```

[75]:
```python
# Finally, to find correlations, you can use the correlation() method and pass␣
↪it
# the two NumPy arrays between which you want to find the correlation.

a1 = np.array([1, 3, 0, 0.9, 1.2])
a2 = np.array([-1, 0.5, 0.2, 0.6, 5])
print(a1)
print(a2)
print("correlation value:")
print(np.correlate(a1, a2))
```

```
[1.  3.  0.  0.9 1.2]
[-1.   0.5  0.2  0.6  5. ]
correlation value:
[7.04]
```

[76]:
```python
# Oftentimes, you would need to find unique values within a NumPy array and
# to find the count of every unique value in a NumPy array.
# To find unique values in a NumPy array, you need to pass the array to the
# unique() method from the NumPy module. Here is an example:
import numpy as np
my_array =np.array([5,8,7,5,9,3,7,7,1,1,8,4,6,9,7,3])
unique_items = np.unique(my_array)
print(unique_items)
```

```
[1 3 4 5 6 7 8 9]
```

[77]:
```python
# To find the count of every unique item in a NumPy array, you need to pass
# the array to the unique() method and pass True as the value for the
# return_counts attribute. The unique() method in this case returns a tuple,
# which contains a list of all unique items and a corresponding list of counts␣
↪for
# each unique item. Here is an example:

my_array =np.array([5,8,7,5,9,3,7,7,1,1,8,4,6,9,7,3])
unique_items, counts = np.unique(my_array, return_counts=True)
print(unique_items)
print(counts)
```

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

[78]:
```python
# One way to get the count value against every unique item is to create an array
# of arrays where the first item is the list of unique items and the second␣
↪item is
# the list of counts, as shown in the script below:
```

```
my_array =np.array([5,8,7,5,9,3,7,7,1,1,8,4,6,9,7,3])
unique_items, counts = np.unique(my_array, return_counts=True)
print(unique_items)
print(counts)
frequencies = np.asarray((unique_items, counts))
print(frequencies)
```

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

[79]:

```
# Next, you can transpose the array that contains the list of unique items and
# their counts. In the output, you will get an array where each item is a list.␣
 ↪The
# first item in the list is the unique item itself, while the second is the␣
 ↪count of
# the item. For instance, in the output of the script below, you can see that␣
 ↪item
# 1 occurs twice in the input array, item 3 also occurs twice, and so on.

my_array =np.array([5,8,7,5,9,3,7,7,1,1,8,4,6,9,7,3])
unique_items, counts = np.unique(my_array, return_counts=True)
print(unique_items)
print(counts)
frequencies = np.asarray((unique_items, counts)).T
print(frequencies)
```

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

Refer to https://numpy.org/doc/stable/reference/generated/numpy.ndarray.T.html for more details about Transpose

[80]:

```
# There are two main methods to reverse a NumPy array: flipud() and fliplr().
# The flipud() method is used to reverse a one-dimensional NumPy array, as
# shown in the script below:

print("original")
my_array = np.random.randint(1,20,10)
```

```
print(my_array)
print("reversed")
reversed_array = np.flipud(my_array)
print(reversed_array)
```

```
original
[19  6  3  4 13  3  5 10 17 16]
reversed
[16 17 10  5  3 13  4  3  6 19]
```

[81]:
```
# On the other hand, to reverse a two-dimensional NumPy array, you can use
# the fliplr() method, as shown below:

print("original")
my_array = np.random.randint(1,20, size = (3,4))
print(my_array)
print("reversed")
reversed_array = np.fliplr(my_array)
print(reversed_array)
```

```
original
[[ 6 15  5  9]
 [17 18 12 10]
 [19  7  8  9]]
reversed
[[ 9  5 15  6]
 [10 12 18 17]
 [ 9  8  7 19]]
```

[82]:
```
# CSV files are an important source of data. The NumPy library contains
# functions that allow you to store NumPy arrays in the form of CSV files. The
# NumPy module also allows you to load CSV files into NumPy arrays.
# To save a NumPy array in the form of a CSV file, you can use the tofile()
# method and pass it your NumPy array.
# The following script stores a two-dimensional array of three rows and four
# columns to a CSV file. You can see that you need to pass a comma "," as the
# value for the sep parameter.


my_array = np.random.randint(1,20, size = (3,4))
print(my_array)
my_array.tofile('CarPrice.csv', sep = ',')
# The following NumPy array will be stored in the form of a CSV file.
# If you open the CSV file, you will seethe NumPy array you stored had
# two dimensions, it is flattened and stored as a single row in your CSV file.
# This is the default behavior of the tofile() function.
```

```
[[ 6 12 14  4]
```

 [10 4 3 7]
 [6 2 5 1]]

[83]:
```python
# To store a two-dimensional NumPy array in the form of multiple rows in a
# CSV file, you can use the savetxt() method from the NumPy module, as
# shown in the following script.

my_array = np.random.randint(1,20, size = (3,4))
print(my_array)
np.savetxt('CarPrice.csv', my_array, delimiter=',')
# open your newly created CSV file and check the difference.
```

```
[[15 11 19 17]
 [ 9  2  4 13]
 [12 12 13  3]]
```

[84]:
```python
# To load a CSV file into a NumPy array, you can use the genfromtxt() method
# and pass it the CSV file along with a comma "," as the value for the delimiter
# attribute of the genfromtxt() method. Here is an example:

loaded_array = np.genfromtxt('CarPrice.csv', delimiter=',')
print(loaded_array)
```

```
[[15. 11. 19. 17.]
 [ 9.  2.  4. 13.]
 [12. 12. 13.  3.]]
```

[85]:
```python
# NumPy arrays provide a variety of functions to perform arithmetic
# operations. Some of these functions are explained in this section.
# The sqrt() function is used to find the square roots of all the elements in a
# list, as shown below:

nums = [10,20,30,40,50]
np_sqr = np.sqrt(nums)
print(np_sqr)
```

```
[3.16227766 4.47213595 5.47722558 6.32455532 7.07106781]
```

[86]:
```python
# The log() function is used to find the logs of all the elements in a list, as
# shown below:

nums = [10,20,30,40,50]
np_log = np.log(nums)
print(np_log)
```

```
[2.30258509 2.99573227 3.40119738 3.68887945 3.91202301]
```

[87]:
```
# The exp() function takes the exponents of all the elements in a list, as shown
# below:

nums = [10,20,30,40,50]
np_exp = np.exp(nums)
print(np_exp)
```

```
[2.20264658e+04 4.85165195e+08 1.06864746e+13 2.35385267e+17
 5.18470553e+21]
```

[88]:
```
# You can find the sines and cosines of items in a list using the sine and␣
 ↪cosine
# function, respectively, as shown in the following script.

nums = [10,20,30,40,50]
np_sine = np.sin(nums)
print(np_sine)
nums = [10,20,30,40,50]
np_cos = np.cos(nums)
print(np_cos)
```

```
[-0.54402111  0.91294525 -0.98803162  0.74511316 -0.26237485]
[-0.83907153  0.40808206  0.15425145 -0.66693806  0.96496603]
```

[89]:
```
# Data science makes extensive use of linear algebra. The support for
# performing advanced linear algebra functions quickly and efficiently makes
# NumPy one of the most routinely used libraries for data science.
# To find a matrix dot product, you can use the dot() function. To find the dot
# product, the number of columns in the first matrix must match the number of
# rows in the second matrix. Here is an example.

A = np.random.randn(4,5)
B = np.random.randn(5,4)
Z = np.dot(A,B)
print(Z)
```

```
[[-1.49739985  0.98812596 -2.04313075 -1.30679268]
 [ 0.47988136  1.34806668 -1.70840157 -0.57021635]
 [-0.22899703  2.7196726  -1.43178076  1.50717677]
 [-1.39136305 -1.52688078 -0.60190022 -1.44703014]]
```

[90]:
```
# In addition to finding the dot product of two matrices, you can element-wise
# multiply two matrices. To do so, you can use the multiply() function.
# However, the dimensions of the two matrices must match.

row1 = [10,12,13]
row2 = [45,32,16]
```

```
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
multiply = np.multiply(nums_2d, nums_2d)
print(multiply)
```

```
[[ 100  144  169]
 [2025 1024  256]
 [2025 1024  256]]
```

[91]:
```
# You find the inverse of a matrix via the linalg.inv() function, as shown
# below:

row1 = [1,2,3]
row2 = [5,2,8]
row3 = [9,1,10]
nums_2d = np.array([row1, row2, row3])
inverse = np.linalg.inv(nums_2d)
print(inverse)
```

```
[[ 0.70588235 -1.          0.58823529]
 [ 1.29411765 -1.          0.41176471]
 [-0.76470588  1.         -0.47058824]]
```

[92]:
```
# Similarly, the determinant of a matrix can be found using the linalg.det()
# function, as shown below:

row1 = [1,2,3]
row2 = [5,2,8]
row3 = [9,1,10]
nums_2d = np.array([row1, row2, row3])
determinant = np.linalg.det(nums_2d)
print(determinant)
```

```
16.999999999999993
```

[93]:
```
# The trace of a matrix refers to the sum of all the elements along the diagonal
# of a matrix. To find the trace of a matrix, you can use the trace() function,␣
↪as
# shown below:

row1 = [1,2,3]
row2 = [4,5,6]
row3 = [7,8,9]
nums_2d = np.array([row1, row2, row3])
trace = np.trace(nums_2d)
print(trace)
```

```
15
```

[94]:
```
# Now that you know how to use the NumPy library to perform various linear
# algebra functions, let's try to solve a system of linear equations, which is␣
↪one
# of the most basic problems in linear algebra.
# A system of linear equations refers to a collection of equations with some
# unknown variables. Solving a system of linear equations refers to finding the
# values of the unknown variables in equations. One of the ways to solve a
# system of linear equations is via matrices.
# The following script creates the NumPy arrays A and B for our matrices A
# and B, respectively.

A = np.array([[6, 3],[2, 4]])
B = np.array([42, 32])
# Next, the script below finds the inverse of the matrix A using the inv()
# method from the linalg submodule from the NumPy module.
A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[ 0.22222222 -0.16666667]
 [-0.11111111  0.33333333]]
```

[95]:
```
# Finally, the script below takes the dot product of the inverse of matrix A␣
↪with
# matrix B (which is a vector in our case).
X = np.dot(A_inv, B)
print(X)
# The output shows that in our system of linear equations, the values of the
# unknown variables x and y are 4 and 6, respectively.
```

```
[4. 6.]
```

0.1 Numpy Hstack vs Vstack

0.1.1 Hstack is used to merge two arrays Horizontally and Vstack is used to merge two arrays vertically

[96]:
```
a=np.array([1,2,3,4])
b=np.array([5,6,7,8])
print(a)
print(b)
```

```
[1 2 3 4]
[5 6 7 8]
```

[97]:
```
c=np.hstack((a,b))
print(c)
```

```
[1 2 3 4 5 6 7 8]
```

[98]:
```
a=np.array([1,2,3,4])
b=np.array([5,6,7,8])
print(a)
print(b)
```

```
[1 2 3 4]
[5 6 7 8]
```

[99]:
```
c=np.vstack((a,b))
print(c)
```

```
[[1 2 3 4]
 [5 6 7 8]]
```

0.2 Masked Arrays

Masked arrays are arrays that may have missing or invalid enteries.

- In many circumstances, datasets can be incomplete or tained by the presence of invalid data.
- For Example, a sensor may have failed to record a data or recorded an invalid value.

0.2.1 The numpy module provides a convenient way to address this issue by introducing masked arrays

0.2.2 What is a mask ?

A mask is either

- nomask,indicating that no value of associated array is invalid.
- array of booleans that determines for each element of the associated array whether the valueis valid or not.
- When an element of the mask is False, the corresponding element of the associated array is valid and is said to be unmasked.
- When an element of the mask is True, the corresponding element of the associated array is said to be masked (invalid).

[100]:
```
import numpy as np
import numpy.ma as ma
```

[101]:
```
x = np.array([1,2,3,-1,5])

# masking fourth entry

mx =ma.masked_array(x,mask=[0,0,0,1,0])

# mean operation does not consider fourth element

mx.mean()
```

[101]: 2.75

[102]:
```
x= np.arange(10).reshape(2,5)
print(x)
np.ma.asarray(x)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

[102]:
```
masked_array(
  data=[[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]],
  mask=False,
  fill_value=999999)
```

[103]:
```
a = np.arange(4)
```

[104]:
```
a
```

[104]:
```
array([0, 1, 2, 3])
```

[105]:
```
# value 2 is masked
b=ma.masked_equal(a,2)
```

[106]:
```
b.mean() #value 2 is not considered in mean operaton
```

[106]:
```
1.3333333333333333
```

[107]:
```
b
```

[107]:
```
masked_array(data=[0, 1, --, 3],
             mask=[False, False,  True, False],
       fill_value=2)
```

[108]:
```
c= ma.masked_greater(a,2) #all the values greater than 2 is masked
```

[109]:
```
print(c)
```

```
[0 1 2 --]
```

[110]:
```
c
```

[110]:
```
masked_array(data=[0, 1, 2, --],
             mask=[False, False, False,  True],
       fill_value=999999)
```

[111]:
```
x = [0.31,1.2,0.01,0.2,-0.4,-1.1]
```

[112]:
```
d= ma.masked_inside(x,-0.3,0.3)
```

[113]:
```
d
```

[113]:
```
masked_array(data=[0.31, 1.2, --, --, -0.4, -1.1],
             mask=[False, False,  True,  True, False, False],
       fill_value=1e+20)
```

[114]:
```
d.mean() #masked values are not included in mean operations
```

[114]:
```
0.0024999999999999467
```

Refer to https://numpy.org/doc/stable/reference/maskedarray.generic.html#:~:text=A%20masked%20array%20is
for more details about masked arrays

[ ]:
```
```