2024S2 - IT2313

Programming for Data Science

NumPy for Numerical Computations (Part 1)

Introduction to NumPy Arrays

NumPy Arrays

Overview of NumPy and Arrays

https://numpy.org/

NumPy (or Numpy) is a Linear Algebra Library for Python, the reason it is so important for Data Science with Python is that almost all of the libraries in the PyData Ecosystem rely on NumPy as one of their main building blocks. Numpy is also incredibly fast, as it has bindings to C libraries.

The array object in NumPy is called ***ndarray***, it provides a lot of supporting functions that make working with ***ndarray*** very easy. To create a new NumPy array, you use the ***array()*** function of the NumPy library.

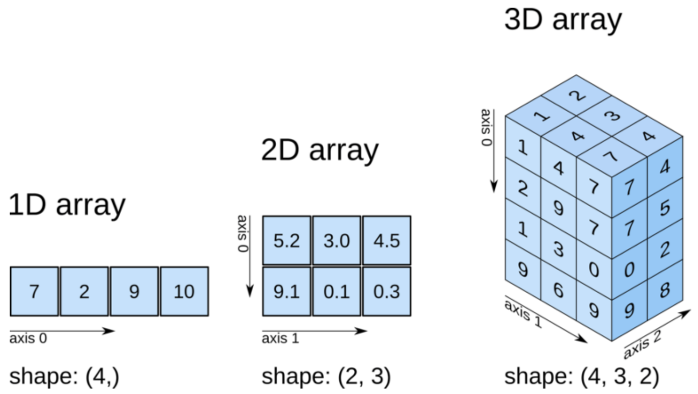
Arrays are very frequently used in data science, where speed and resources are very important.

NumPy Arrays

The key concept in NumPy is the NumPy array data type. A NumPy array may have one or more dimensions: [Create NumPy Arrays]

- One dimension arrays (1D) represent vectors.
- Two-dimensional arrays (2D) represent matrices.
- And higher dimensional arrays represent tensors

In this module, we will focus more on 2D arrays - Matrices



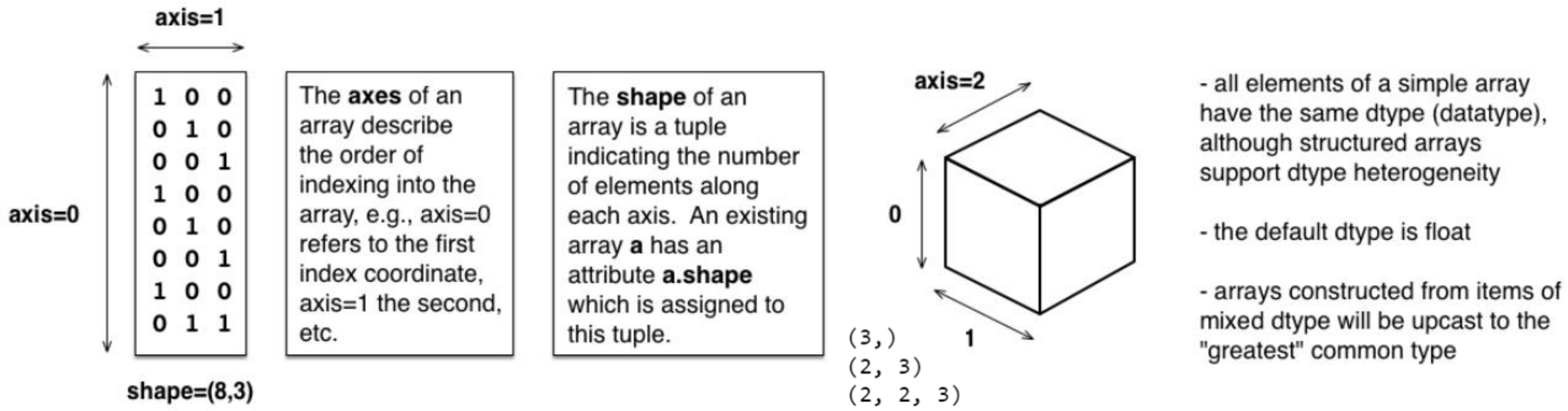
Unlike the built-in list type that can hold the elements of different types, the NumPy arrays allow only one data type for all elements. Therefore, we say that the NumPy array requires homogeneous data values.

A NumPy array can contain either integer or float numbers, but not at the same time. This restriction allows Numpy to speed up the linear algebra calculations.

Anatomy of an NumPy Array

Think of an array as a lineup of boxes, each with its own unique number. We call these numbers indices, and they start from 0 and go up sequentially. Now, each box can hold a specific piece of information, like a number, a word, or even a whole bunch of data. [Numpy Arrays]

Anatomy of an array, you're basically talking about how these boxes are organized and accessed. You can refer to each box by its index, and that makes it super easy to find, update, or remove information.



The **axes** of an array describe the order of indexing into the array, e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.

The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array **a** has an attribute **a.shape** which is assigned to this tuple.

- all elements of a simple array have the same dtype (datatype), although structured arrays support dtype heterogeneity

- the default dtype is float

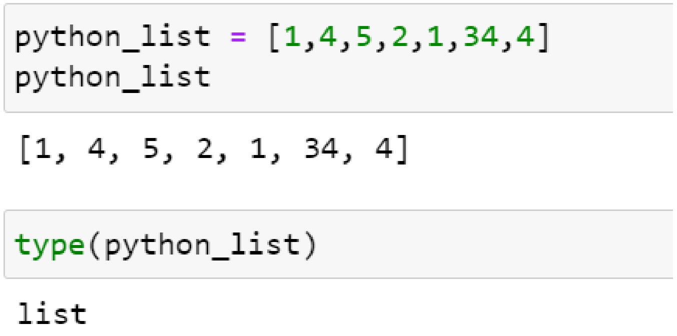
- arrays constructed from items of mixed dtype will be upcast to the "greatest" common type

Lists vs NumPy Arrays

Examples on List vs NumPy Arrays in terms on Syntax:

- Python List: Python lists have a more straightforward syntax and are part of the core Python language
- NumPy Array: NumPy introduces additional syntax for array operations, making it more specialized for numerical computing.

Python List



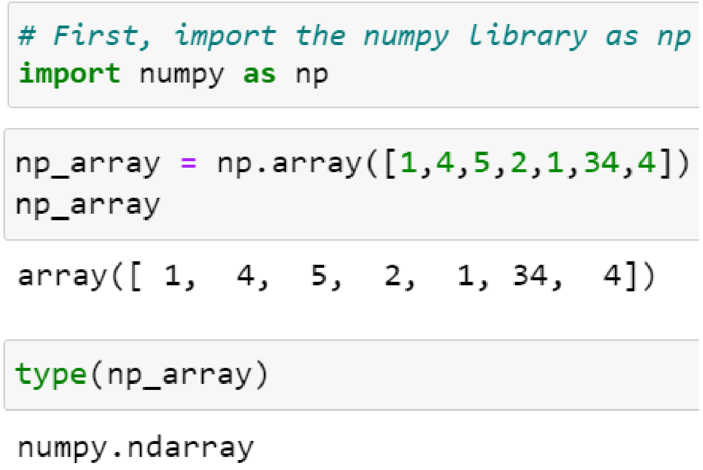
```
python_list = [1,4,5,2,1,34,4]
python_list
```

```
[1, 4, 5, 2, 1, 34, 4]
```

```
type(python_list)
```

```
list
```

NumPy Array



```
# First, import the numpy library as np
import numpy as np
```

```
np_array = np.array([1,4,5,2,1,34,4])
np_array
```

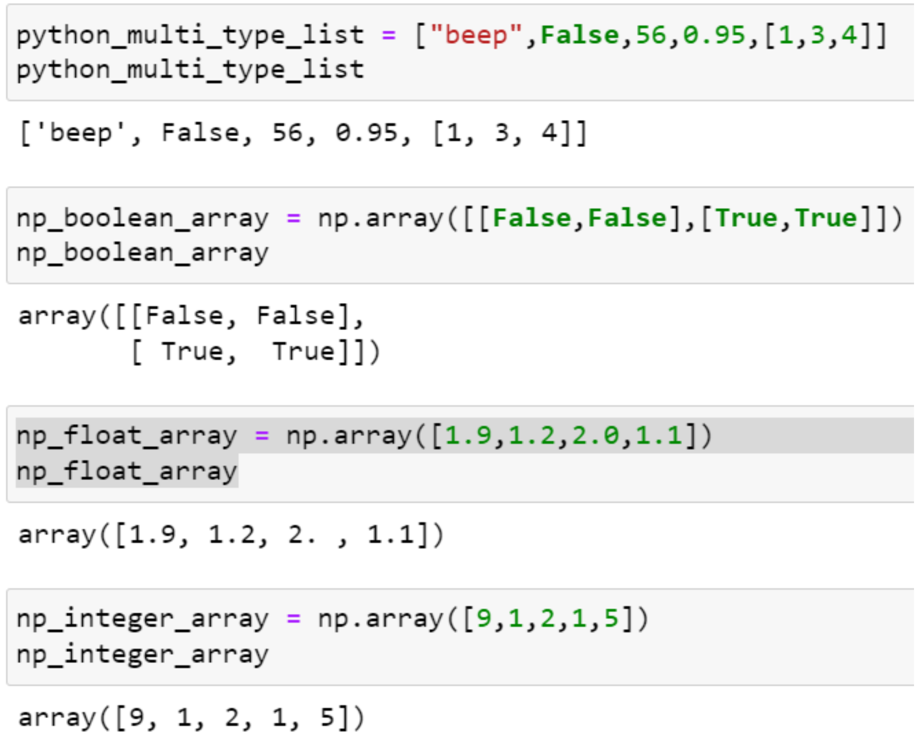
```
array([ 1,  4,  5,  2,  1, 34,  4])
```

```
type(np_array)
```

```
numpy.ndarray
```

Lists vs NumPy Arrays

Python lists can contain many different data types, NumPy arrays can contain only a single data type



```
python_multi_type_list = ["beep",False,56,0.95,[1,3,4]]
python_multi_type_list
```

```
['beep', False, 56, 0.95, [1, 3, 4]]
```

```
np_boolean_array = np.array([[False,False],[True,True]])
np_boolean_array
```

```
array([[False, False],
       [ True,  True]])
```

```
np_float_array = np.array([1.9,1.2,2.0,1.1])
np_float_array
```

```
array([1.9, 1.2, 2. , 1.1])
```

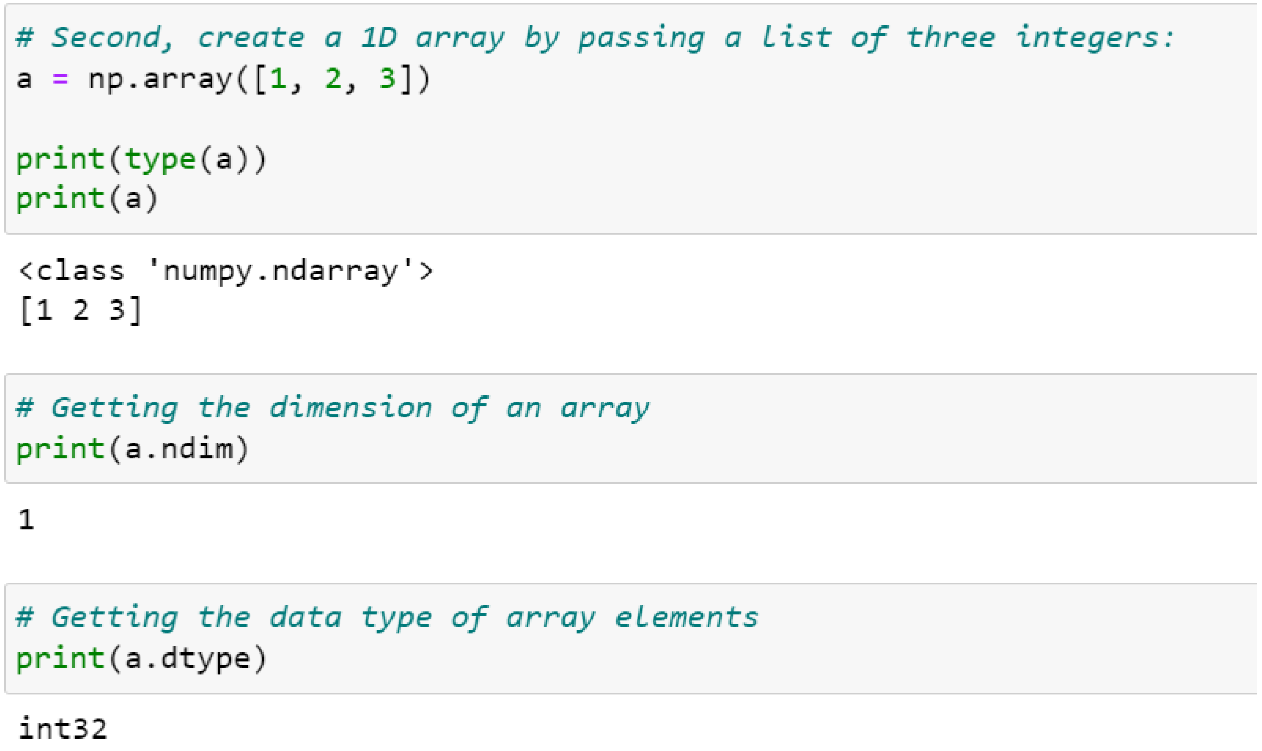
```
np_integer_array = np.array([9,1,2,1,5])
np_integer_array
```

```
array([9, 1, 2, 1, 5])
```

Creating NumPy Arrays

Creating one-dimensional arrays

The following example uses the array() function to create a one-dimensional (1-D) array:



```python
# Second, create a 1D array by passing a list of three integers:
a = np.array([1, 2, 3])

print(type(a))
print(a)
```

```
<class 'numpy.ndarray'>
[1 2 3]
```

```python
# Getting the dimension of an array
print(a.ndim)
```

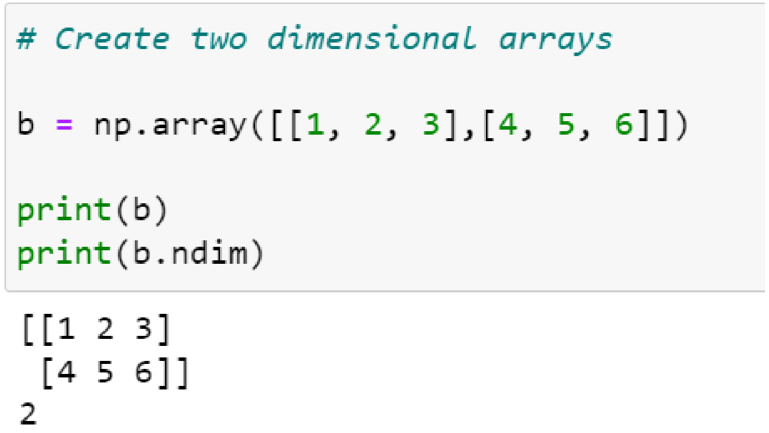
```
1
```

```python
# Getting the data type of array elements
print(a.dtype)
```

```
int32
```

Creating two-dimensional arrays

The following example uses the array() function to create a two-dimensional (2-D) array::

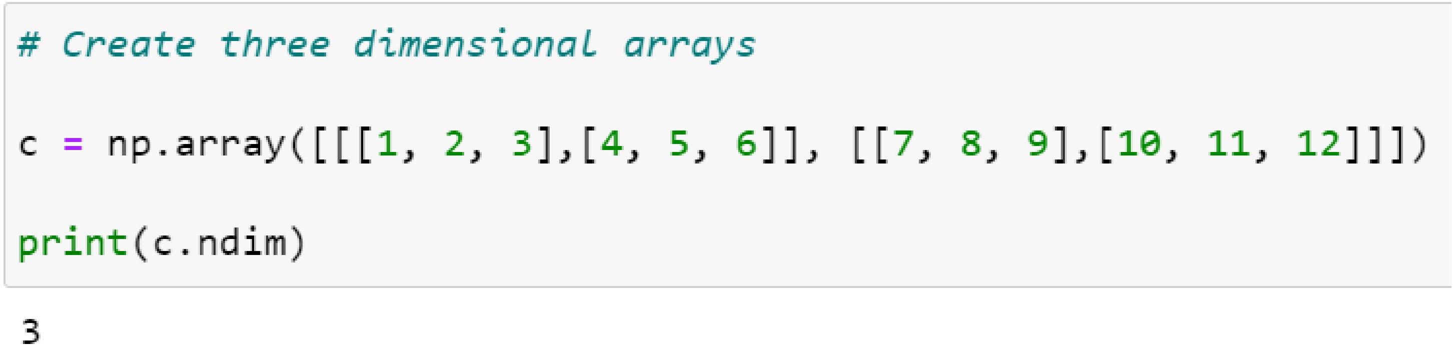


```python
# Create two dimensional arrays

b = np.array([[1, 2, 3],[4, 5, 6]])

print(b)
print(b.ndim)
```

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

Creating three-dimensional arrays

The following example uses the array() function to create a three-dimensional (3-D) array::

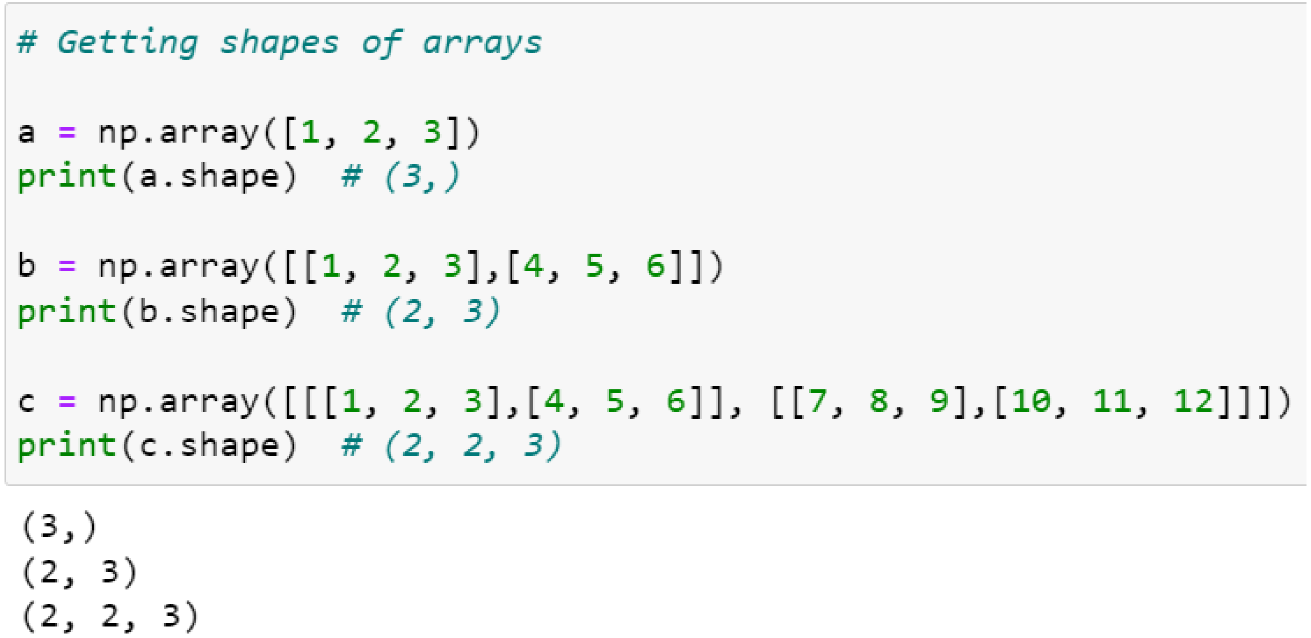


```python
# Create three dimensional arrays

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

print(c.ndim)
```

```
3
```

Creating three-dimensional arrays

Getting shapes of arrays



```
# Getting shapes of arrays

a = np.array([1, 2, 3])
print(a.shape)  # (3,)

b = np.array([[1, 2, 3],[4, 5, 6]])
print(b.shape)  # (2, 3)

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

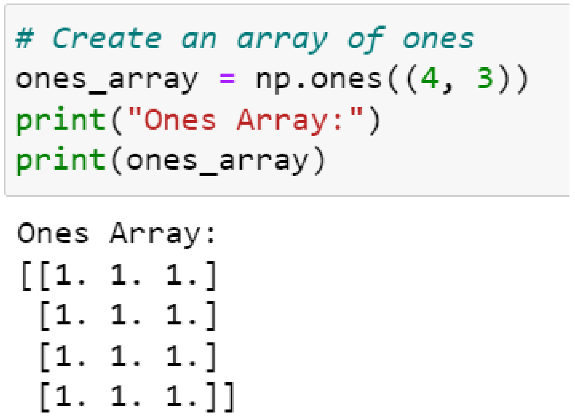
```
(3,)
(2, 3)
(2, 2, 3)
```

Creating NumPy Arrays

The following are some commands which can be use to create the NumPy arrays. Create NumPy Arrays

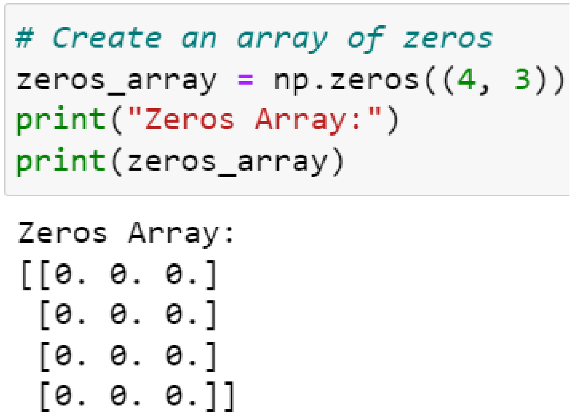
S/N	Command	Description	Examples
1	np.ones()	Create an array of ones	np.ones((4,3))
2	np.zeros()	Create an array of zeros	np.zeros((4,3))
3	np.full()	Create an full array	np.full((3,2),-1)
4	np.random.random()	Create an array with random values with uniform distribution between 0 (inclusive) and 1 (exclusive)	np.random.random((2,4))
5	np.arrange()	Uses a specified step size to generate values within the specified range	np.arange(10, 25, 5)
6	np.linspace()	Generates a specified number of evenly spaced values between a start and end point.	np.linspace(0,2,9)

Creating NumPy Arrays



```python
# Create an array of ones
ones_array = np.ones((4, 3))
print("Ones Array:")
print(ones_array)
```

```
Ones Array:
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```



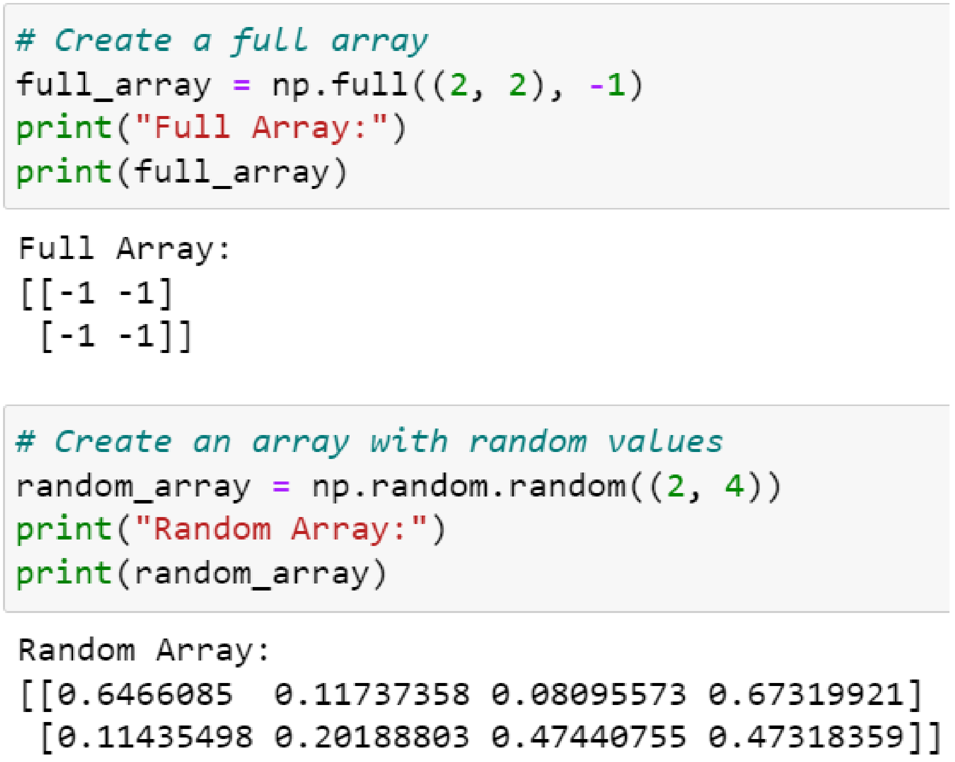
```python
# Create an array of zeros
zeros_array = np.zeros((4, 3))
print("Zeros Array:")
print(zeros_array)
```

```
Zeros Array:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

For the np.ones() and np.zeros() the only thing that you need to do in order to make arrays with ones or zeros is pass the shape of the array that you want to make.

As an option to np.ones() and np.zeros(), you can also specify the data type.

Creating NumPy Arrays



```python
# Create a full array
full_array = np.full((2, 2), -1)
print("Full Array:")
print(full_array)
```

```
Full Array:
[[-1 -1]
 [-1 -1]]
```

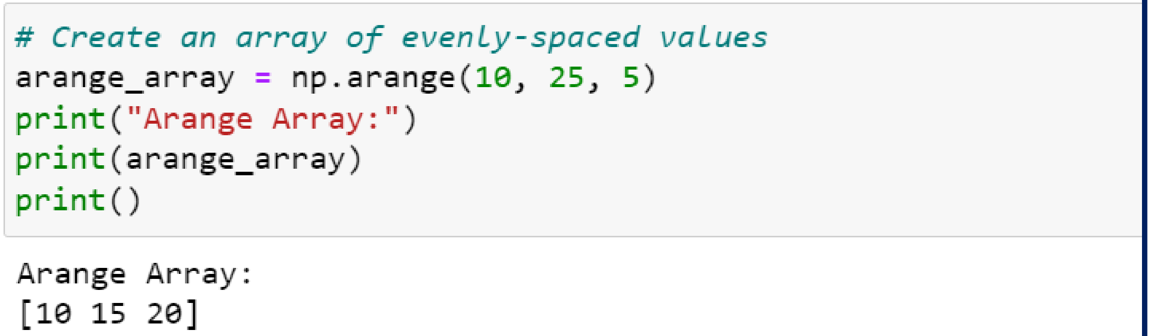
```python
# Create an array with random values
random_array = np.random.random((2, 4))
print("Random Array:")
print(random_array)
```

```
Random Array:
[[0.6466085  0.11737358 0.08095573 0.67319921]
 [0.11435498 0.20188803 0.47440755 0.47318359]]
```

Similar to the np.ones() and np.zeros(), for the np.full() and np.random.random() the only thing that you need to do in order to make arrays with ones or zeros is pass the shape of the array that you want to make.

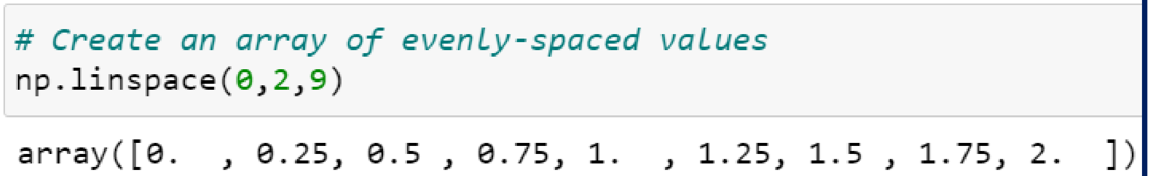
In the case of np.full(), you also have to specify the constant value that you want to insert into the array.

Creating NumPy Arrays



```
# Create an array of evenly-spaced values
arange_array = np.arange(10, 25, 5)
print("Arange Array:")
print(arange_array)
print()
```

```
Arange Array:
[10 15 20]
```



```
# Create an array of evenly-spaced values
np.linspace(0,2,9)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

np.arrange() and np.linspace() are both NumPy functions used to generate arrays of evenly spaced values.

They have some differences in terms of how they specify the range and the number of elements.

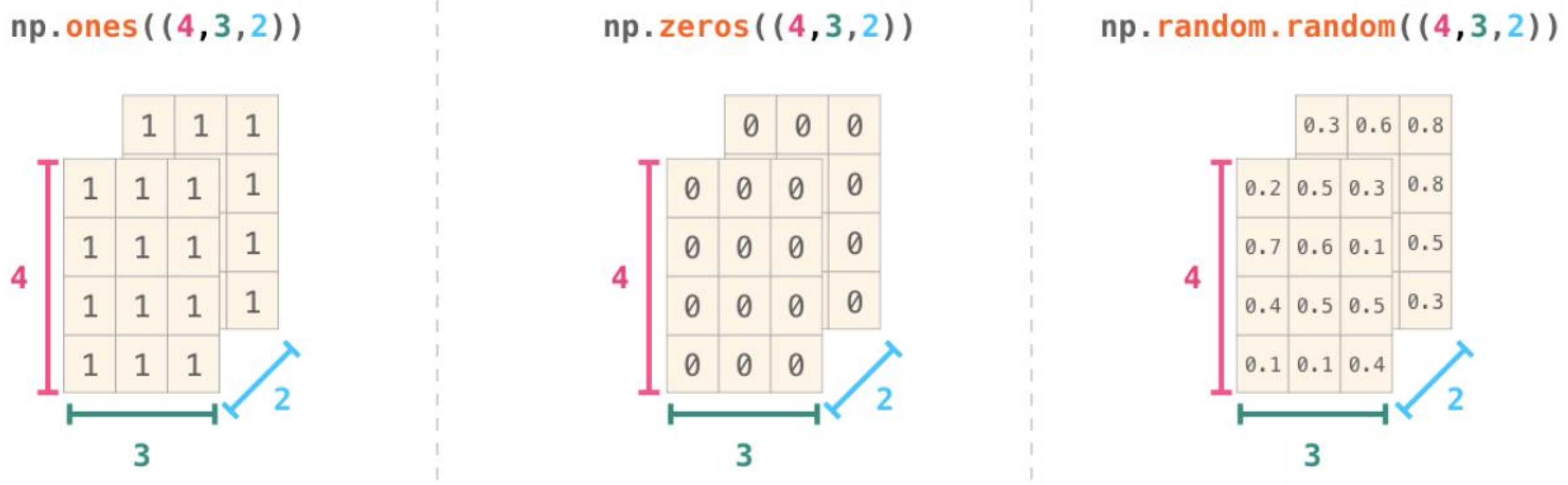
For np.arrange(), Parameters:
start: Start of the interval (default is 0).
stop: End of the interval (not inclusive).
step: Spacing between values (default is 1).

Returns an array with values in the half-open interval [start, stop) with the specified step size.

For np.linspace(), Parameters:
start: Start of the interval.
stop: End of the interval.
num: Number of evenly spaced values to generate (default is 50).

Creating NumPy Arrays – Visual Representation

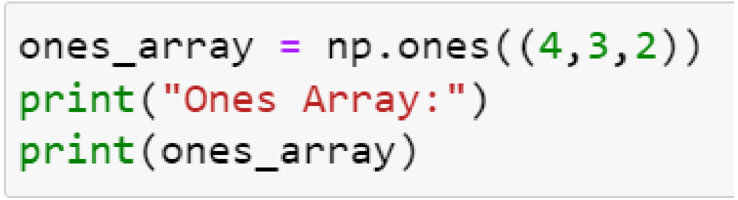
In a lot of ways, dealing with a new dimension is just adding a comma to the parameters of a NumPy function: A Virtual Intro to NumPy and Data Visualisation



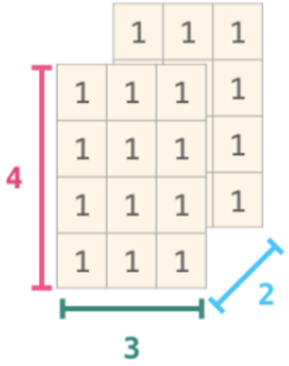
Do take note on how the Arrays are visualized and represented

Creating NumPy Arrays – Visual Representation

Keep in mind that when you print a 3-dimensional NumPy array, the text output visualizes the array differently than shown previously. NumPy's order for printing n-dimensional arrays is that the last axis is looped over the fastest, while the first is the slowest. Which means that np.ones((4,3,2)) would be printed as:



```
ones_array = np.ones((4,3,2))
print("Ones Array:")
print(ones_array)
```

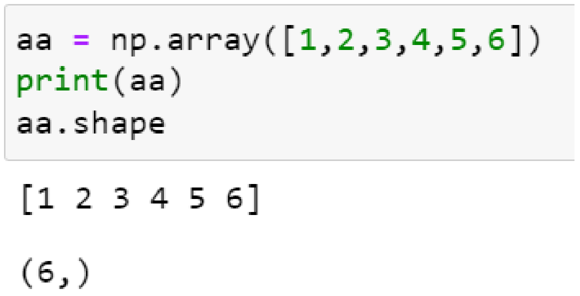


```
Ones Array:
[[[1. 1.]
  [1. 1.]
  [1. 1.]]

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

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

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

NumPy Arrays Manipulation

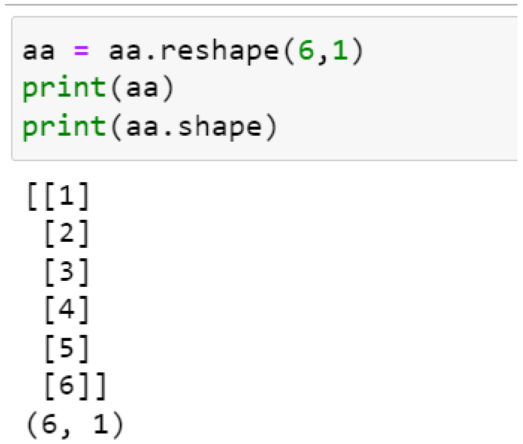
Vector Arrays

NumPy allows the reshaping of the array. The below example reshapes an 1D array into a 2D array of 6 rows with 1 element and further reshapes this array into another 2D array of 1 row with 6 elements.



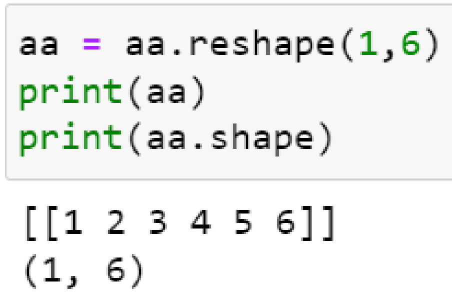
```
aa = np.array([1,2,3,4,5,6])
print(aa)
aa.shape
```

```
[1 2 3 4 5 6]

(6,)
```



```
aa = aa.reshape(6,1)
print(aa)
print(aa.shape)
```

```
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
(6, 1)
```

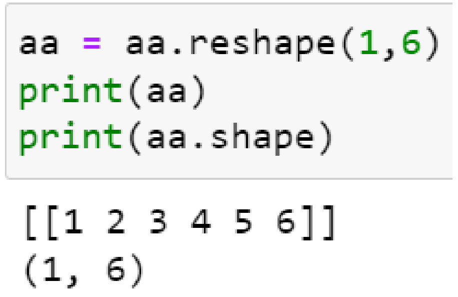


```
aa = aa.reshape(1,6)
print(aa)
print(aa.shape)
```

```
[[1 2 3 4 5 6]]
(1, 6)
```

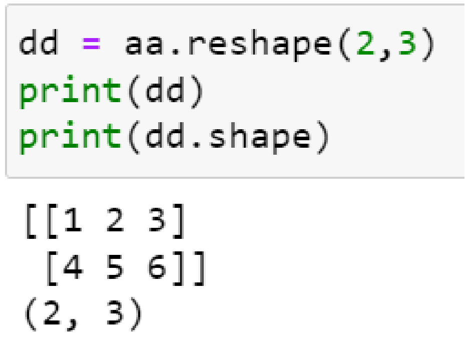
Vector Arrays

The below example reshapes the array into another 2D array of 2 rows and each with 3 elements.



```
aa = aa.reshape(1,6)
print(aa)
print(aa.shape)
```

```
[[1 2 3 4 5 6]]
(1, 6)
```



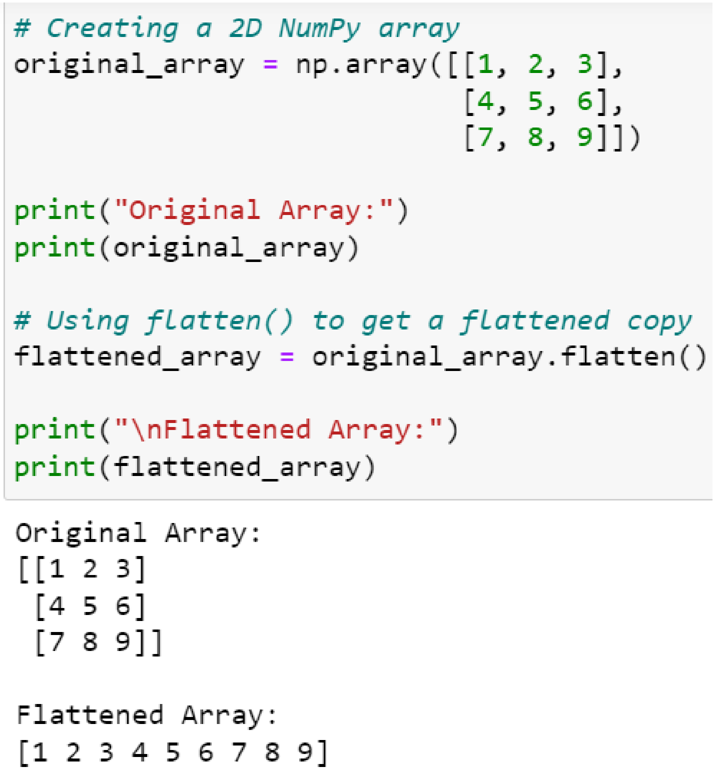
```
dd = aa.reshape(2,3)
print(dd)
print(dd.shape)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

Reshape only works if the dimensions stated are valid.

Flatten Arrays

flatten() is particularly useful for multidimensional arrays. It converts arrays with any number of dimensions into a one-dimensional array



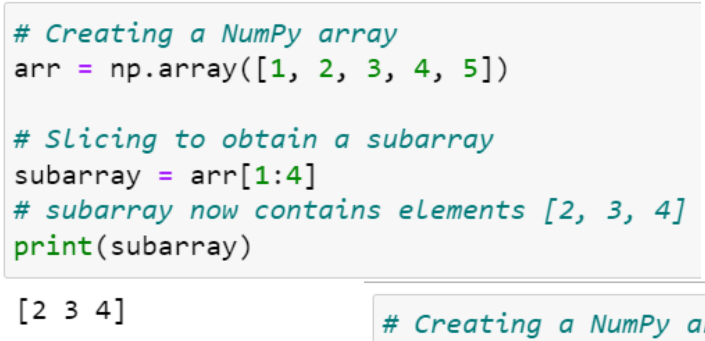
```python
# Creating a 2D NumPy array
original_array = np.array([[1, 2, 3],
                           [4, 5, 6],
                           [7, 8, 9]])

print("Original Array:")
print(original_array)

# Using flatten() to get a flattened copy
flattened_array = original_array.flatten()

print("\nFlattened Array:")
print(flattened_array)
```

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

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

Slice and Index – 1D Arrays

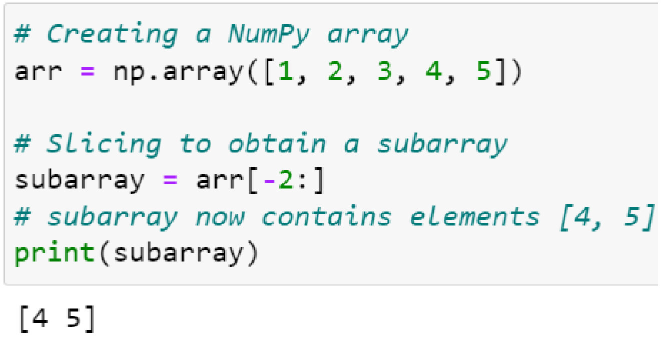
Slice

A slice in NumPy is a view or a portion of an array obtained by specifying a range of indices. To extract a contiguous subarray.



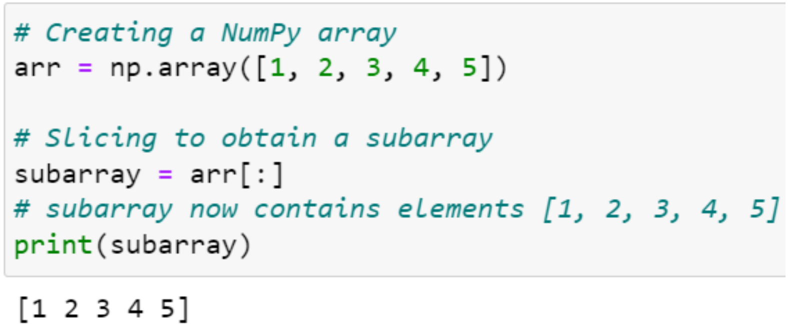
```python
# Creating a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Slicing to obtain a subarray
subarray = arr[1:4]
# subarray now contains elements [2, 3, 4]
print(subarray)
```

```
[2 3 4]
```



```python
# Creating a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Slicing to obtain a subarray
subarray = arr[-2:]
# subarray now contains elements [4, 5]
print(subarray)
```

```
[4 5]
```



```python
# Creating a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Slicing to obtain a subarray
subarray = arr[:]
# subarray now contains elements [1, 2, 3, 4, 5]
print(subarray)
```

```
[1 2 3 4 5]
```

Slice and Index – 1D Arrays

Summary of Slice Commands Usage

Let's represent the original array as [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:

Explanation of Slice Notation:

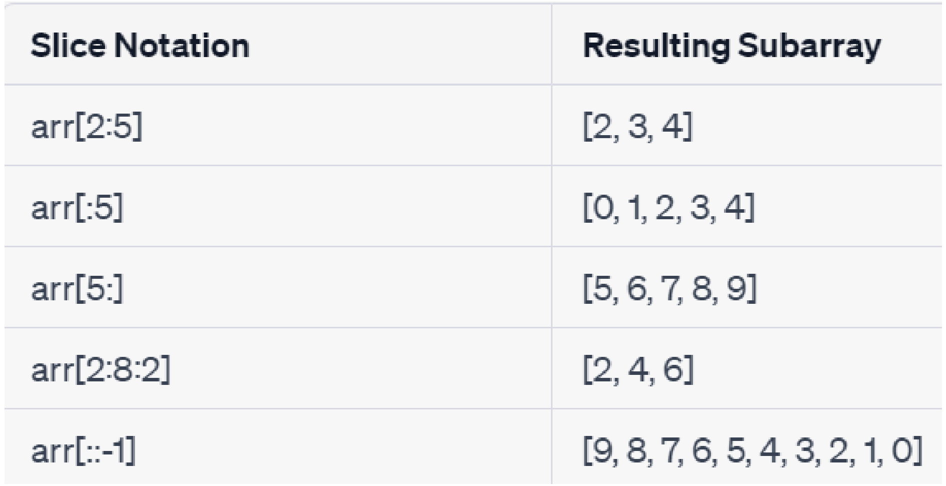
arr[2:5]: Elements from index 2 to 4 (5 excluded).

arr[:5]: Elements from the beginning up to index 4.

arr[5:]: Elements from index 5 to the end.

arr[2:8:2]: Elements from index 2 to 7 with a step of 2.

arr[::-1]: Reversed order of all elements.

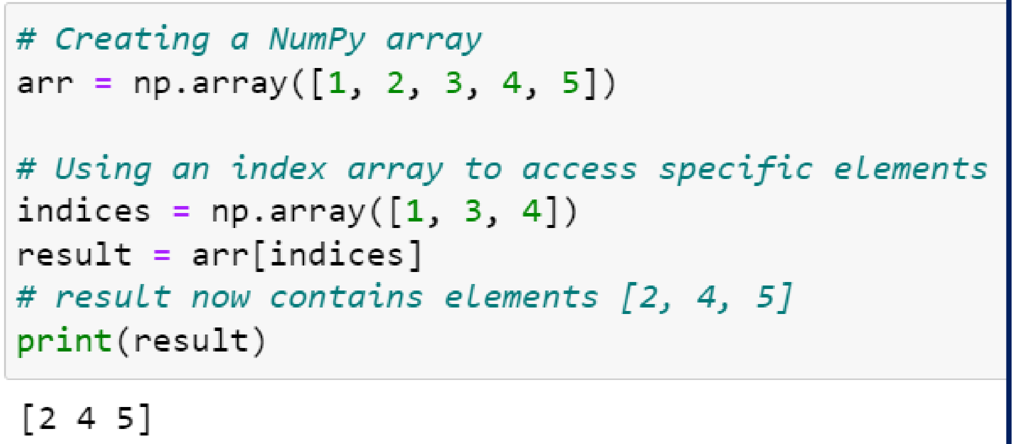


Slice Notation	Resulting Subarray
arr[2:5]	[2, 3, 4]
arr[:5]	[0, 1, 2, 3, 4]
arr[5:]	[5, 6, 7, 8, 9]
arr[2:8:2]	[2, 4, 6]
arr[::-1]	[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Slice and Index – 1D Arrays

Index

An index array in NumPy is an array of integer indices used to access specific elements in another array. It provides a way to access non-contiguous elements..



```
# Creating a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Using an index array to access specific elements
indices = np.array([1, 3, 4])
result = arr[indices]
# result now contains elements [2, 4, 5]
print(result)
```

```
[2 4 5]
```

arr = np.array([1, 2, 3, 4, 5]): Creates a 1D NumPy array named arr with elements 1 to 5.

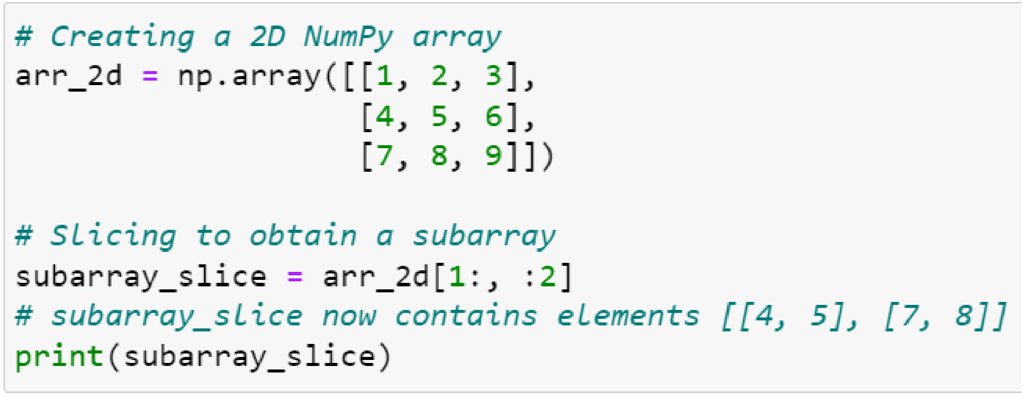
indices = np.array([1, 3, 4]): Creates a NumPy array indices containing the indices of the elements you want to access from the array arr.

result = arr[indices]: Uses the index array indices to access specific elements from the array arr.

The result is a new 1D array containing the elements at the specified indices, which are [2, 4, 5] in this case

Slice and Index – 2D Arrays

Slice



```python
# Creating a 2D NumPy array
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Slicing to obtain a subarray
subarray_slice = arr_2d[1:, :2]
# subarray_slice now contains elements [[4, 5], [7, 8]]
print(subarray_slice)
```

```
[[4 5]
 [7 8]]
```

This line uses slicing to obtain a subarray.

1: selects all rows starting from the second row (index 1) to the end.
:2 selects the first two columns (index 0 and index 1).

The resulting subarray_slice contains elements [[4, 5], [7, 8]], which are the elements in the second and third rows, and the first two columns of the original array.

Slice and Index – 2D Arrays

Using the following 2D array outputs

```python
# Creating a 2D NumPy array
random_2d = np.array([np.random.randint(10,size=5),
                      np.random.randint(10,size=5),
                      np.random.randint(10,size=5)])
print(random_2d)
```

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

```python
# Slicing to obtain a subarray
subarray_slice = random_2d[0:1, 0:3]
# subarray_slice now contains elements [[9,5,1]]
print(subarray_slice)
```

```
[[9 5 1]]
```

Using slicing to select elements from the rows with indices 0 up to (but not including) 1, and columns with indices 0 up to (but not including) 3

```python
# Slicing to obtain a subarray
subarray_slice = random_2d[0:3, 0:3]
# subarray_slice now contains elements
# [[9,5,1],[6,6,9],[0,4,1]]
print(subarray_slice)
```

```
[[9 5 1]
 [6 6 9]
 [0 4 1]]
```

To select elements from the rows with indices 0 up to (but not including) 3, and columns with indices 0 up to (but not including) 3.

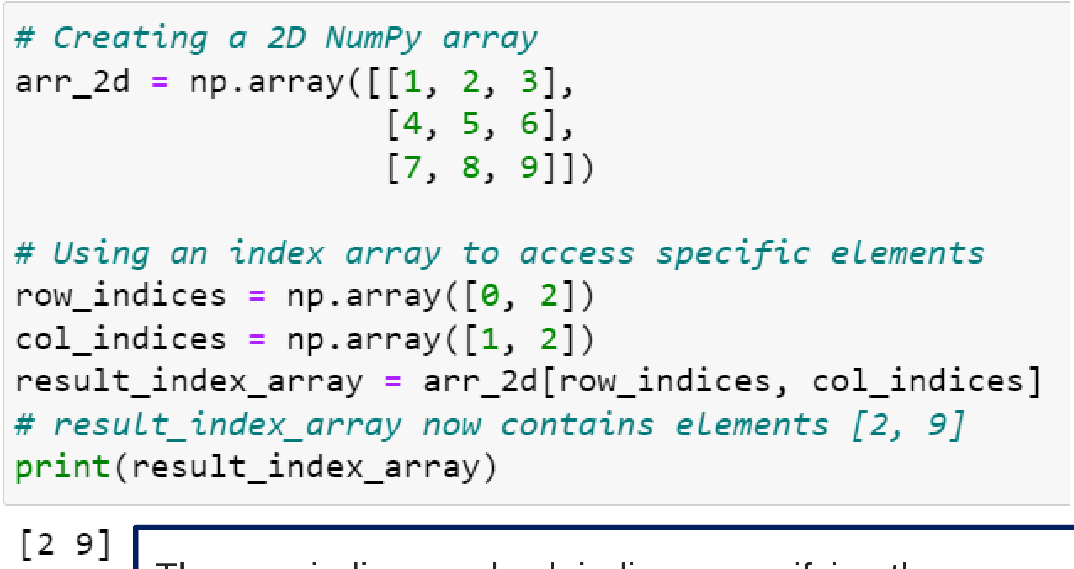
```python
# Slicing to obtain a subarray
subarray_slice = random_2d[1:3, 1:3]
# subarray_slice now contains elements
# [[6,9],[4,1]]
print(subarray_slice)
```

```
[[6 9]
 [4 1]]
```

To select elements from the rows with indices 1 up to (but not including) 3, and columns with indices 1 up to (but not including) 3

Slice and Index – 2D Arrays

Index



```python
# Creating a 2D NumPy array
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Using an index array to access specific elements
row_indices = np.array([0, 2])
col_indices = np.array([1, 2])
result_index_array = arr_2d[row_indices, col_indices]
# result_index_array now contains elements [2, 9]
print(result_index_array)
```

```
[2 9]
```

The row_indices and col_indices, specifying the row and column indices of the elements you want to access. In this case, you want to access elements at (0, 1), and (2, 2).

The resulting 1D array, which contains the elements [2, 9] obtained by using the index arrays to access specific elements from the original 2D array.

Slice and Index – 2D Arrays

Using the codes below to create a 2D array

```python
# Creating a 2D NumPy array
random_2d = np.array([np.random.randint(10,size=5),
                      np.random.randint(10,size=5),
                      np.random.randint(10,size=5)])
print(random_2d)
```

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

```python
# Using an index array to access specific elements
subarray_indices = random_2d[1]
print(subarray_indices)
```

```
[6 6 9 2 2]
```

Using array indexing to extract the second row (index 1) of the 2D array

```python
subarray_indices = random_2d[1,2]
print(subarray_indices)
```

```
9
```

The indices 1, 2 correspond to the element in the second row (index 1) and third column (index 2).

```python
subarray_indices = random_2d[2,3]
print(subarray_indices)
```

```
8
```

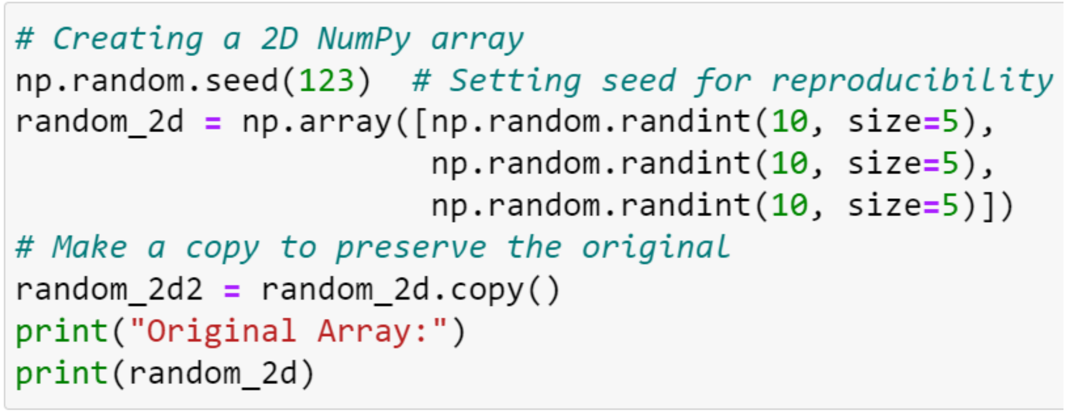
The indices 2, 3 correspond to the element in the third row (index 2) and fourth column (index 3).

Sorting the NumPy Arrays

Sorting the NumPy Arrays

We covered 3 basic sorting techniques in this course:

1. Sort the entire array
2. Sort along rows
3. Sort along columns



```
# Creating a 2D NumPy array
np.random.seed(123)  # Setting seed for reproducibility
random_2d = np.array([np.random.randint(10, size=5),
                      np.random.randint(10, size=5),
                      np.random.randint(10, size=5)])
# Make a copy to preserve the original
random_2d2 = random_2d.copy()
print("Original Array:")
print(random_2d)
```

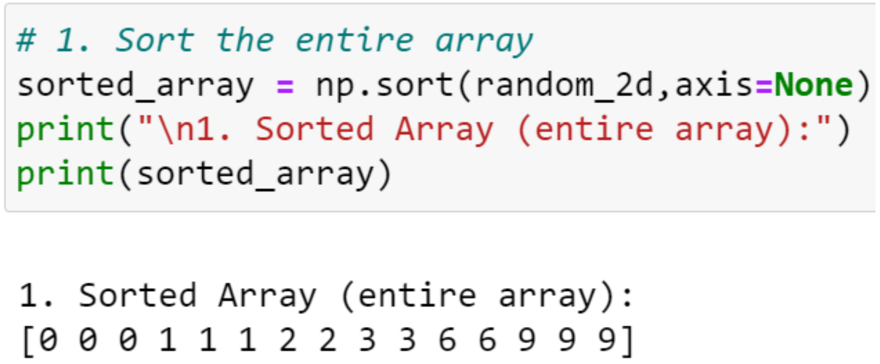
```
Original Array:
[[2 2 6 1 3]
 [9 6 1 0 1]
 [9 0 0 9 3]]
```

A two-dimensions array will be used to cover the examples. Using the codes below to create a 2D array.

We made a copy of the randomly generated array before sorting using random_2d2 = random_2d.copy()

Sorting the NumPy Arrays

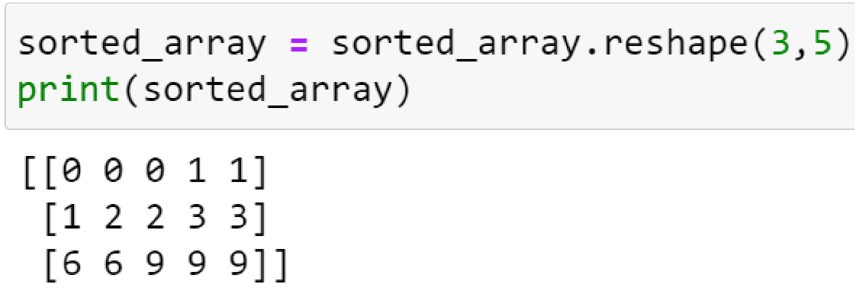
1. Sort the entire array



```
# 1. Sort the entire array
sorted_array = np.sort(random_2d,axis=None)
print("\n1. Sorted Array (entire array):")
print(sorted_array)
```

```
1. Sorted Array (entire array):
[0 0 0 1 1 1 2 2 3 3 6 6 9 9 9]
```

The axis=None argument indicates that the entire array should be flattened before sorting. This means that all the elements from the 2D array are treated as a single sequence, and they are sorted in ascending order.



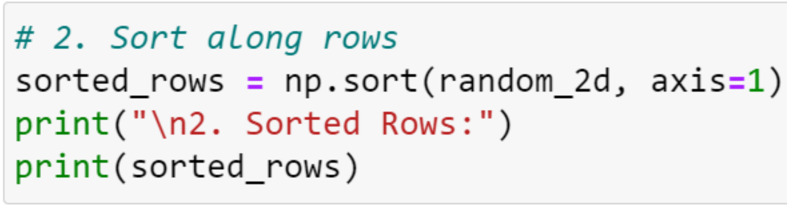
```
sorted_array = sorted_array.reshape(3,5)
print(sorted_array)
```

```
[[0 0 0 1 1]
 [1 2 2 3 3]
 [6 6 9 9 9]]
```

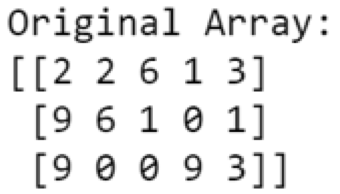
The reshape() is being used to transform back the original array shape and dimensions

Sorting the NumPy Arrays

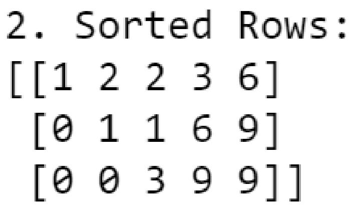
2. Sort along rows



```
# 2. Sort along rows
sorted_rows = np.sort(random_2d, axis=1)
print("\n2. Sorted Rows:")
print(sorted_rows)
```



```
Original Array:
[[2 2 6 1 3]
 [9 6 1 0 1]
 [9 0 0 9 3]]
```

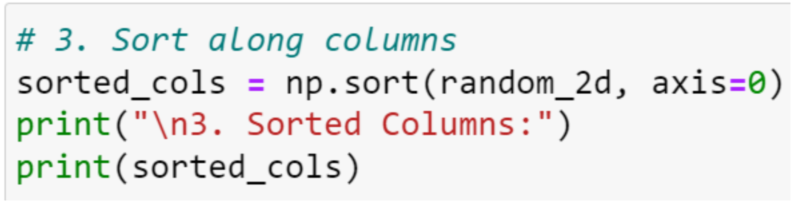


```
2. Sorted Rows:
[[1 2 2 3 6]
 [0 1 1 6 9]
 [0 0 3 9 9]]
```

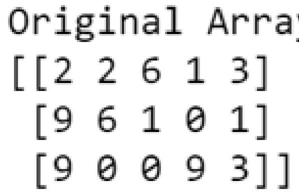
Setting axis=1 while using np.sort() sorts each row independently

Sorting the NumPy Arrays

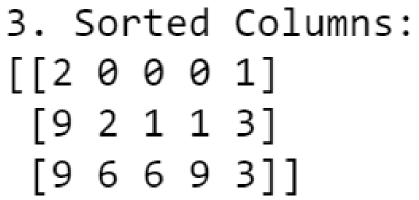
3. Sort along columns



```python
# 3. Sort along columns
sorted_cols = np.sort(random_2d, axis=0)
print("\n3. Sorted Columns:")
print(sorted_cols)
```



```
Original Array:
[[2 2 6 1 3]
 [9 6 1 0 1]
 [9 0 0 9 3]]
```

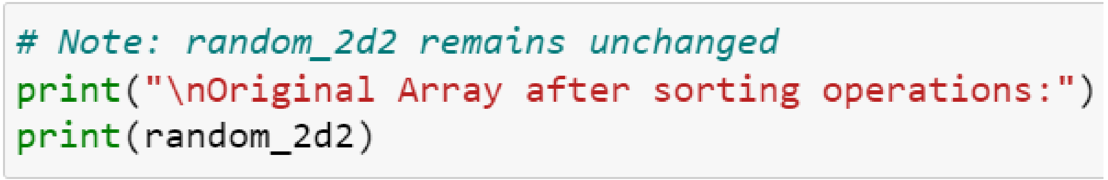


```
3. Sorted Columns:
[[2 0 0 0 1]
 [9 2 1 1 3]
 [9 6 6 9 3]]
```

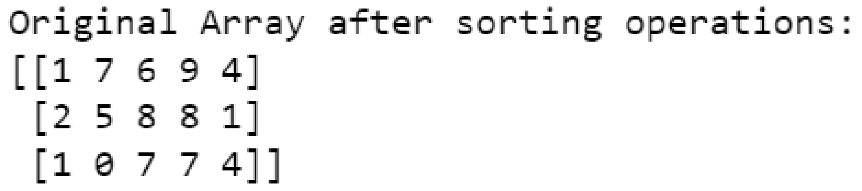
Setting axis=0 while using np.sort() sorts each column independently

Sorting the NumPy Arrays

The random_2d2 array remains unchanged, emphasizing that these operations do not modify the original array unless assigned explicitly.



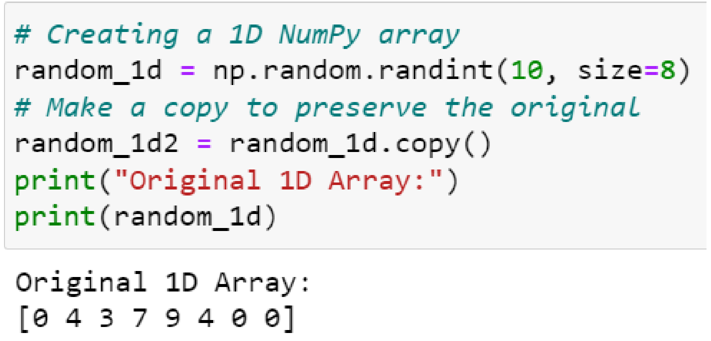
```python
# Note: random_2d2 remains unchanged
print("\nOriginal Array after sorting operations:")
print(random_2d2)
```



```
Original Array after sorting operations:
[[1 7 6 9 4]
 [2 5 8 8 1]
 [1 0 7 7 4]]
```

Filtering the NumPy Arrays

Filtering the NumPy Arrays

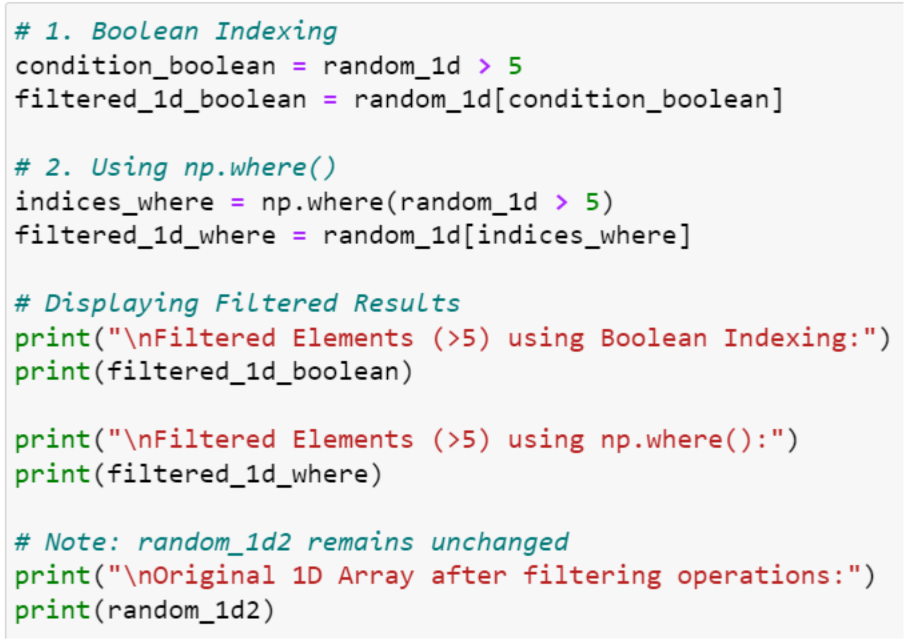


```
# Creating a 1D NumPy array
random_1d = np.random.randint(10, size=8)
# Make a copy to preserve the original
random_1d2 = random_1d.copy()
print("Original 1D Array:")
print(random_1d)
```

```
Original 1D Array:
[0 4 3 7 9 4 0 0]
```

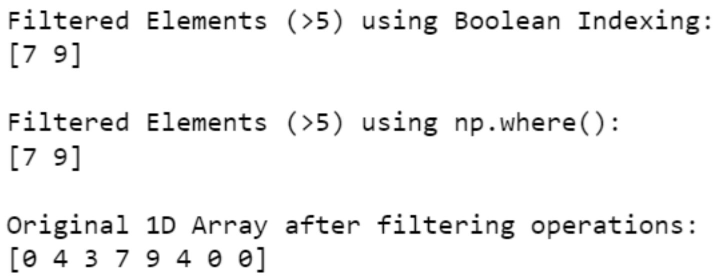
Two method on Filtering the NumPy Arrays are discussed:

- Boolean Indexing
- np.where() function



```
# 1. Boolean Indexing
condition_boolean = random_1d > 5
filtered_1d_boolean = random_1d[condition_boolean]

# 2. Using np.where()
indices_where = np.where(random_1d > 5)
filtered_1d_where = random_1d[indices_where]

# Displaying Filtered Results
print("\nFiltered Elements (>5) using Boolean Indexing:")
print(filtered_1d_boolean)

print("\nFiltered Elements (>5) using np.where():")
print(filtered_1d_where)

# Note: random_1d2 remains unchanged
print("\nOriginal 1D Array after filtering operations:")
print(random_1d2)
```

For the above randomly generated Array, let filter the elements which values are > 5



```
Filtered Elements (>5) using Boolean Indexing:
[7 9]

Filtered Elements (>5) using np.where():
[7 9]

Original 1D Array after filtering operations:
[0 4 3 7 9 4 0 0]
```

Filtering the NumPy Arrays

Let use back the same array but let check for the Even values.

```python
# Creating a 1D NumPy array
random_1d = np.random.randint(10, size=8)
# Make a copy to preserve the original
random_1d2 = random_1d.copy()
print("Original 1D Array:")
print(random_1d)
```

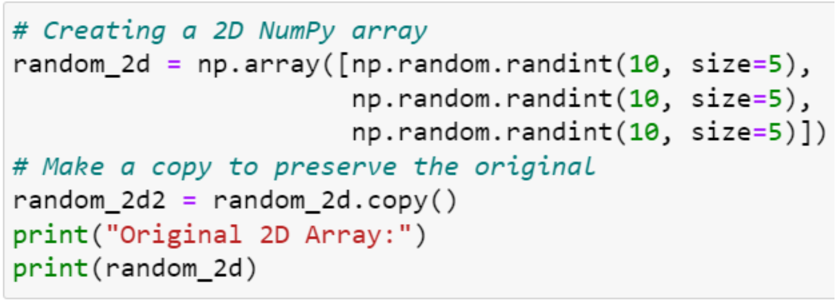
```
Original 1D Array:
[0 4 3 7 9 4 0 0]
```

```python
# 1. Boolean Indexing
condition_boolean = random_1d % 2 == 0
filtered_1d_boolean = random_1d[condition_boolean]

# 2. Using np.where()
indices_where = np.where(condition_boolean)
filtered_1d_where = random_1d[indices_where]

# Displaying Filtered Results
print("\nFiltered Even Numbering using Boolean Indexing:")
print(filtered_1d_boolean)

print("\nFiltered Even Numbers using np.where():")
print(filtered_1d_where)

# Note: random_1d2 remains unchanged
print("\nOriginal 1D Array after filtering operations:")
print(random_1d2)
```

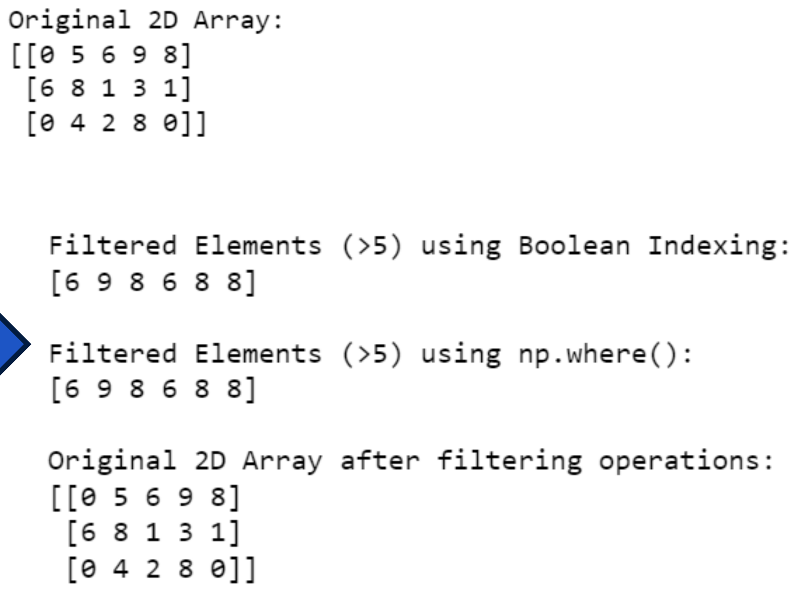
```
Filtered Even Numbering using Boolean Indexing:
[0 4 4 0 0]

Filtered Even Numbers using np.where():
[0 4 4 0 0]

Original 1D Array after filtering operations:
[0 4 3 7 9 4 0 0]
```

Filtering the NumPy Arrays

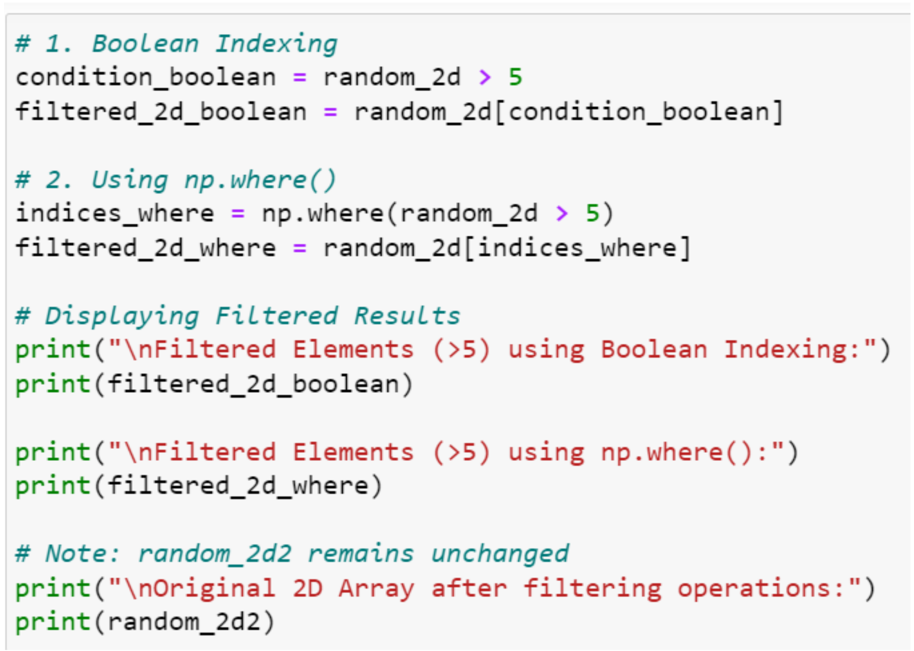
Same approach can be applied for multi-dimensions array.



```python
# Creating a 2D NumPy array
random_2d = np.array([np.random.randint(10, size=5),
                      np.random.randint(10, size=5),
                      np.random.randint(10, size=5)])
# Make a copy to preserve the original
random_2d2 = random_2d.copy()
print("Original 2D Array:")
print(random_2d)
```



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



```python
# 1. Boolean Indexing
condition_boolean = random_2d > 5
filtered_2d_boolean = random_2d[condition_boolean]

# 2. Using np.where()
indices_where = np.where(random_2d > 5)
filtered_2d_where = random_2d[indices_where]

# Displaying Filtered Results
print("\nFiltered Elements (>5) using Boolean Indexing:")
print(filtered_2d_boolean)

print("\nFiltered Elements (>5) using np.where():")
print(filtered_2d_where)

# Note: random_2d2 remains unchanged
print("\nOriginal 2D Array after filtering operations:")
print(random_2d2)
```

```
Filtered Elements (>5) using Boolean Indexing:
[6 9 8 6 8 8]

Filtered Elements (>5) using np.where():
[6 9 8 6 8 8]

Original 2D Array after filtering operations:
[[0 5 6 9 8]
 [6 8 1 3 1]
 [0 4 2 8 0]]
```

Filtering the NumPy Arrays - Examples

Below are the SIT intakes for Diploma in Information Technology (DIT) in AY2023.

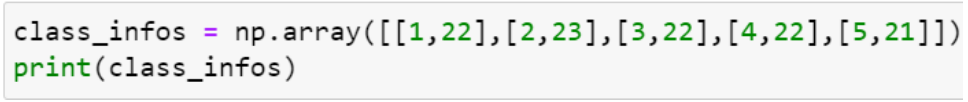
Class	1	2	3	4	5
No of Students	22	23	22	22	21

Let perform the following exercises

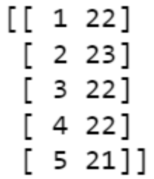
1. Create NumPy array based on table information.
2. Get the Information for Class 3
3. Get the information for Class with > 22 students
4. Determine which classes have the even number of students

Filtering the NumPy Arrays - Examples

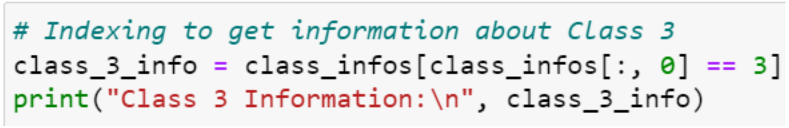
Class	1	2	3	4	5
No of Students	22	23	22	22	21



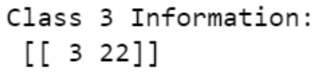
```python
class_infos = np.array([[1,22],[2,23],[3,22],[4,22],[5,21]])
print(class_infos)
```



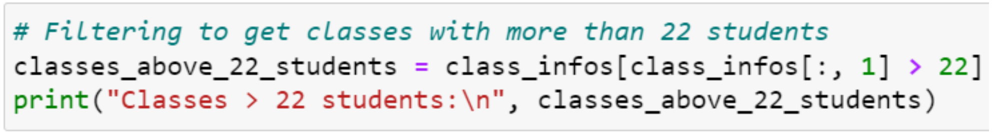
```
[[ 1 22]
 [ 2 23]
 [ 3 22]
 [ 4 22]
 [ 5 21]]
```



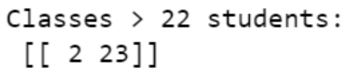
```python
# Indexing to get information about Class 3
class_3_info = class_infos[class_infos[:, 0] == 3]
print("Class 3 Information:\n", class_3_info)
```



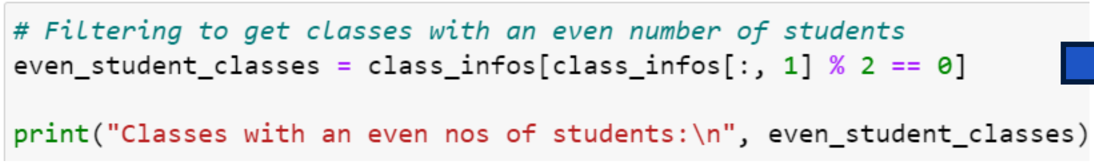
```
Class 3 Information:
 [[ 3 22]]
```



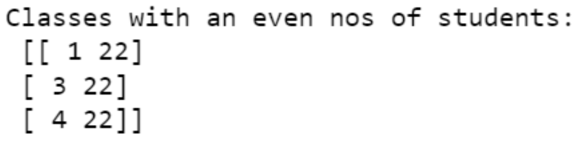
```python
# Filtering to get classes with more than 22 students
classes_above_22_students = class_infos[class_infos[:, 1] > 22]
print("Classes > 22 students:\n", classes_above_22_students)
```



```
Classes > 22 students:
 [[ 2 23]]
```



```python
# Filtering to get classes with an even number of students
even_student_classes = class_infos[class_infos[:, 1] % 2 == 0]

print("Classes with an even nos of students:\n", even_student_classes)
```



```
Classes with an even nos of students:
 [[ 1 22]
 [ 3 22]
 [ 4 22]]
```

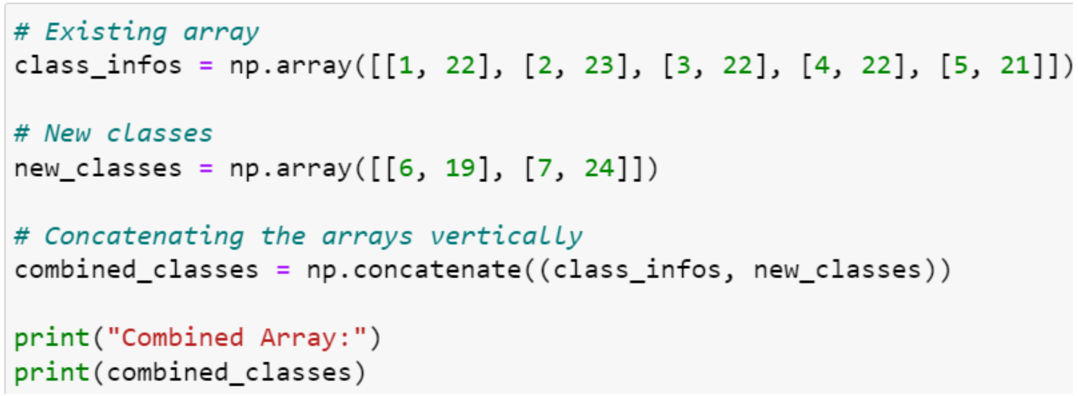
Adding and Removing NumPy Arrays

Adding NumPy Array Element - Examples

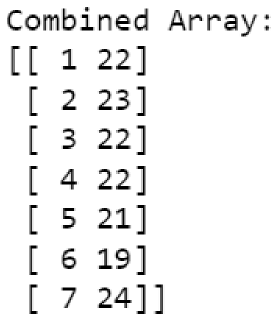
More students enrolled for this Diploma course and we need to start 2 more new classes as shown:.

Class	1	2	3	4	5	6	7
No of Students	22	23	22	22	21	19	24

concatenate() is to add in the new class information into existing array



```
# Existing array
class_infos = np.array([[1, 22], [2, 23], [3, 22], [4, 22], [5, 21]])

# New classes
new_classes = np.array([[6, 19], [7, 24]])

# Concatenating the arrays vertically
combined_classes = np.concatenate((class_infos, new_classes))

print("Combined Array:")
print(combined_classes)
```



```
Combined Array:
[[ 1 22]
 [ 2 23]
 [ 3 22]
 [ 4 22]
 [ 5 21]
 [ 6 19]
 [ 7 24]]
```

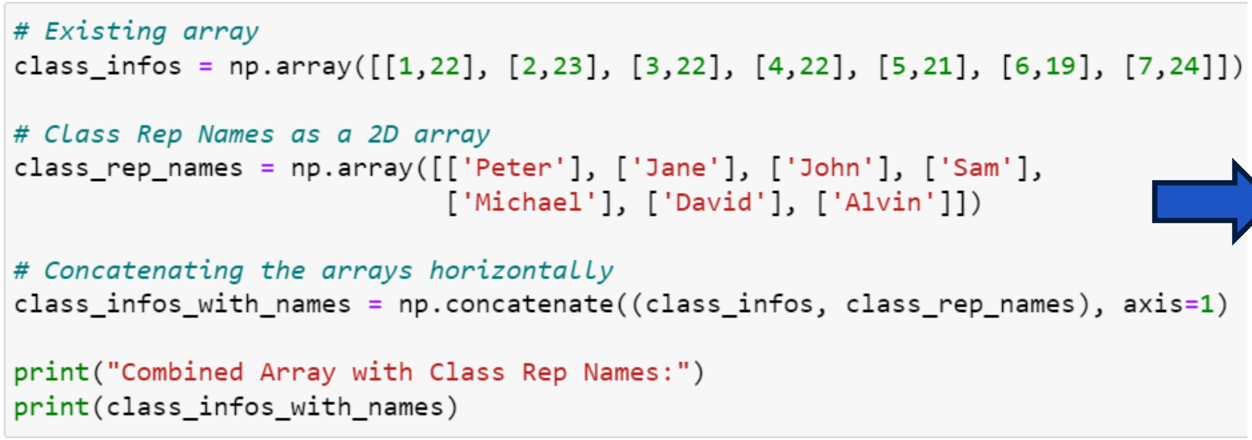
combined_classes = np.concatenate((class_id_student_nos, new_classes), axis=0) will yield the same results. The axis=0 argument specifies that the concatenation should be done along rows.

Adding NumPy Array Element - Examples

Let add in the Class Reps Name into the Array.

Class	1	2	3	4	5	6	7
Class Rep	Peter	Jane	John	Sam	Michael	David	Alvin

concatenate() is to add in the new class rep information into existing array



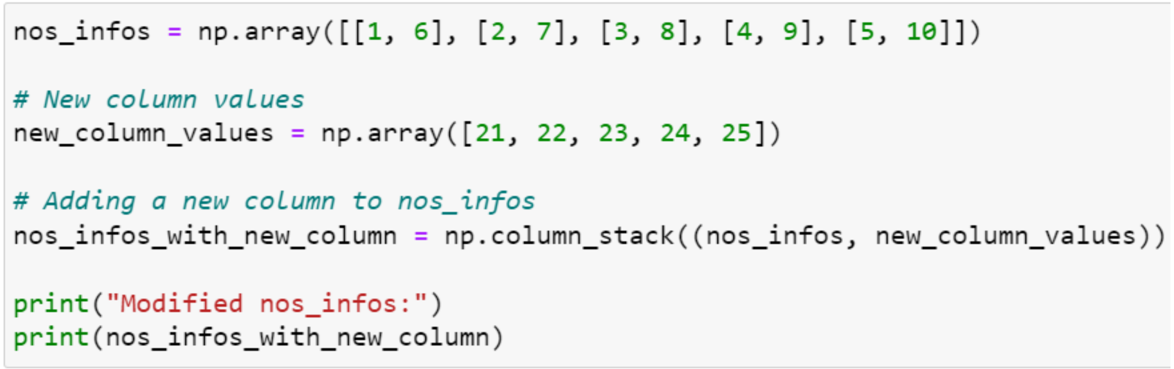
```
# Existing array
class_infos = np.array([[1,22], [2,23], [3,22], [4,22], [5,21], [6,19], [7,24]])

# Class Rep Names as a 2D array
class_rep_names = np.array([['Peter'], ['Jane'], ['John'], ['Sam'],
                            ['Michael'], ['David'], ['Alvin']])

# Concatenating the arrays horizontally
class_infos_with_names = np.concatenate((class_infos, class_rep_names), axis=1)

print("Combined Array with Class Rep Names:")
print(class_infos_with_names)
```

```
Combined Array with Class Rep Names:
[['1' '22' 'Peter']
 ['2' '23' 'Jane']
 ['3' '22' 'John']
 ['4' '22' 'Sam']
 ['5' '21' 'Michael']
 ['6' '19' 'David']
 ['7' '24' 'Alvin']]
```

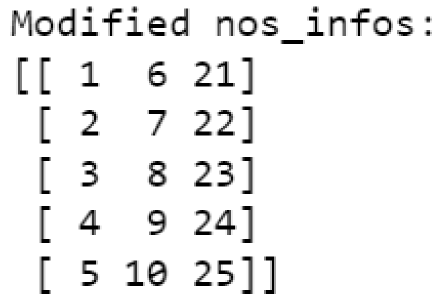
class_infos_with_names = np.concatenate((class_infos, class_rep_names), axis=1)

Adding NumPy Array Element - Examples

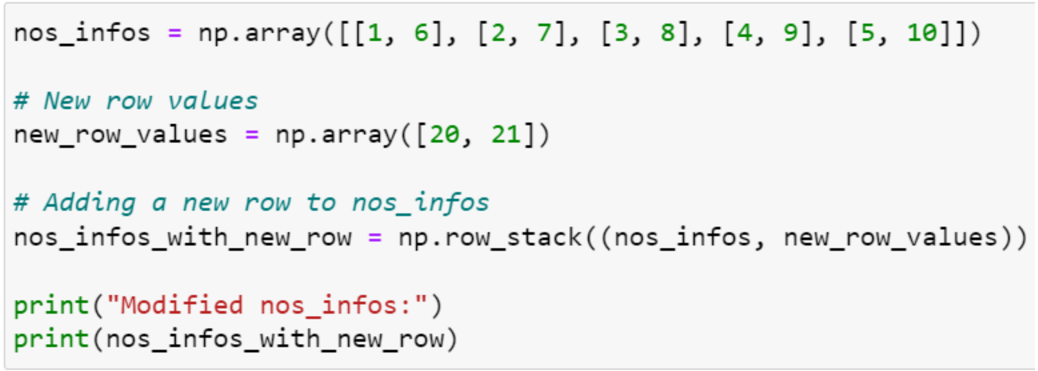
We will now explore using the ***row_stack()*** and ***column_stack()*** to add elements into the NumPy Arrays



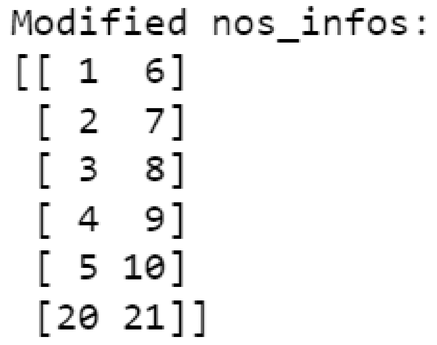
```
nos_infos = np.array([[1, 6], [2, 7], [3, 8], [4, 9], [5, 10]])

# New column values
new_column_values = np.array([21, 22, 23, 24, 25])

# Adding a new column to nos_infos
nos_infos_with_new_column = np.column_stack((nos_infos, new_column_values))

print("Modified nos_infos:")
print(nos_infos_with_new_column)
```



```
Modified nos_infos:
[[ 1  6 21]
 [ 2  7 22]
 [ 3  8 23]
 [ 4  9 24]
 [ 5 10 25]]
```



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

# New row values
new_row_values = np.array([20, 21])

# Adding a new row to nos_infos
nos_infos_with_new_row = np.row_stack((nos_infos, new_row_values))

print("Modified nos_infos:")
print(nos_infos_with_new_row)
```



```
Modified nos_infos:
[[ 1  6]
 [ 2  7]
 [ 3  8]
 [ 4  9]
 [ 5 10]
 [20 21]]
```

Removing NumPy Array Element - Examples

Let add in the Class Reps Name into the Array.

Class	1	2	3	4	5	6	7
No of Students	22	23	22	22	21	19	24
Class Rep	Peter	Jane	John	Sam	Michael	David	Alvin

delete() is used to delete the rows or columns in the existing array. The axis parameter is used to specify which is to be deleted.

```
class_infos = np.array([['1', '22', 'Peter'],
                        ['2', '23', 'Jane'],
                        ['3', '22', 'John'],
                        ['4', '22', 'Sam'],
                        ['5', '21', 'Michael'],
                        ['6', '19', 'David'],
                        ['7', '24', 'Alvin']])
```

Removing NumPy Array Element - Examples

Delete the row of the NumPy Array

```
# Deleting the 2nd row (index 1)
class_infos_modified = np.delete(class_infos, 1, axis=0)

print("Modified Class Infos:")
print(class_infos_modified)
```

```
Modified Class Infos:
[['1' '22' 'Peter']
 ['3' '22' 'John']
 ['4' '22' 'Sam']
 ['5' '21' 'Michael']
 ['6' '19' 'David']
 ['7' '24' 'Alvin']]
```

Class	1	2	3	4	5	6	7
No of Students	22	23	22	22	21	19	24
Class Rep	Peter	Jane	John	Sam	Michael	David	Alvin

Delete the column of the NumPy Array

```
# Deleting the 2nd column (index 1)
class_infos_modified = np.delete(class_infos, 1, axis=1)

print("Modified Class Infos:")
print(class_infos_modified)
```

```
Modified Class Infos:
[['1' 'Peter']
 ['2' 'Jane']
 ['3' 'John']
 ['4' 'Sam']
 ['5' 'Michael']
 ['6' 'David']
 ['7' 'Alvin']]
```

Do compare the 2 different codes

Thank You!

NYP NANYANG
THE INNOVATIVE POLYTECHNIC

www.nyp.edu.sg