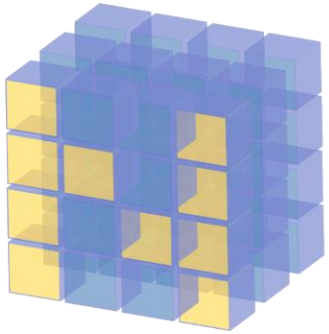


Lecture-10

NumPy in Python

Contents

- Introduction
- Creating arrays
- Array Attributes
- Array Operators
- NumPy Methods
- Shallow Copies
- Deep Copies

Introduction

- NumPy (**Numerical Python**) is a python library used for working with Python arrays.
- It offers a high-performance, richly functional n-dimensional array type called **ndarray.**
- NumPy aims to provide an array object that is up to **50x** faster that traditional Python lists.
- Working domains: Linear algebra, and Matrices.
- Over **450** Python libraries depend on NumPy.
- Popular library: Pandas, SciPy, Keras (Deep Learning Library) depend on NumPy.

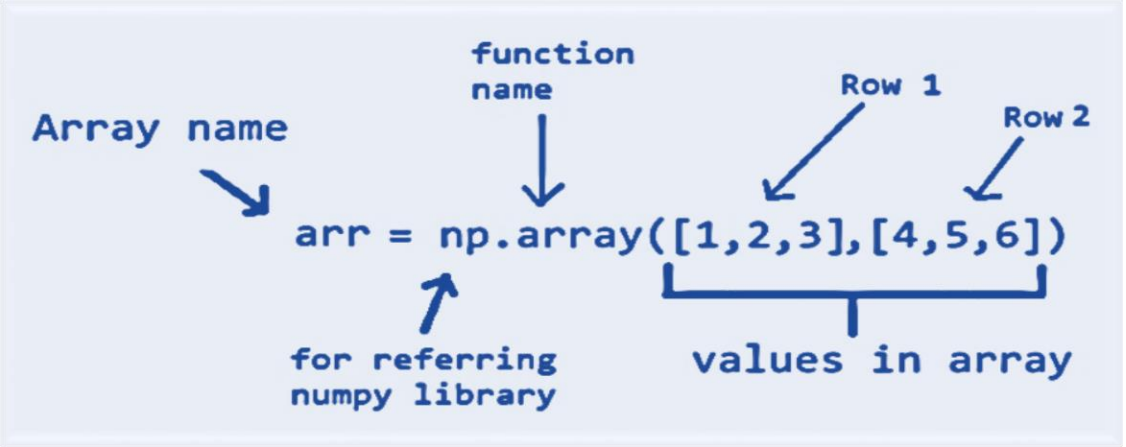
Introduction

- NumPy arrays support different data types.
- **NumPy** vs. **List**
  - NumPy is faster than lists
  - NumPy data structures take up less space than lists
  - NumPy have better runtime behavior.
  - NumPy array need to be declared but lists don’t.

Creating arrays by Importing numpy

- The NumPy documentation recommends importing the numpy module as **np**
- The numpy module provides various functions for creating arrays.

```
import numpy as np
numbers = np.array([2, 3, 5, 7, 11])
```



- The **array()** function, which receives an array as an argument.
- NumPy separates each value from the next with a **comma** and a **space** and **right-aligns** all the values.

Creating arrays by Importing numpy

```
1   import numpy as np
2   num = np.array([10,20,30])
3   num
```

```
array([10, 20, 30])
```

```
1   type(num)
```

```
numpy.ndarray
```

Arrays with Multidimensional Arguments

- The array function copies its argument's dimensions.
- NumPy auto-formats arrays, based on their number of dimensions, aligning the columns within each row.
  - 1D array example:

```
1 array_1D = np.array([2,4,5,6])
2 array_1D
```

```
array([2, 4, 5, 6])
```

  - 2D array example:

```
1 array_2D = np.array([[2,4,5,6],[2,4,5,6],[2,4,5,6]])
2 array_2D
```

```
array([[2, 4, 5, 6],
       [2, 4, 5, 6],
       [2, 4, 5, 6]])
```

  - 3D array example:

```
array_3D = np.array([[[2,4,5,6],[2,4,5,6],[2,4,5,6]]])
array_3D
```

```
array([[[2, 4, 5, 6],
        [2, 4, 5, 6],
        [2, 4, 5, 6]]])
```

Creating arrays from Comprehension

Problem 1:

Create a two-dimensional array from a list comprehension that produces the even integers from 2 through 20 in the first row and the odd integers from 1 through 21 in the second row.

Solution:

```
1 import numpy as np
2 output = np.array([[x for x in range(2,21,2)],[x for x in range(1,22,2)]])
3 output
```

```
array([list([2, 4, 6, 8, 10, 12, 14, 16, 18, 20]),
       list([1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21])], dtype=object)
```

Exercise:

Create a **2-by-5** array containing **reverse order** of the even integers from 2 through 10 in the first row and the odd integers from 1 through 9 in the second row.

Array Attributes

- An array object provides attributes that enable to discover information about its structure and contents. Some important attributes of this NumPy array are:
  - **dtype**:- The array function determines an array's element type from its argument's elements. For this NumPy array [[1,2,3], [4,5,6]], **dtype** will be **int64.**

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

```
1  ary.dtype
```

```
dtype('int64')
```

  - **ndim**:- The attribute **ndim** contains an array's number of dimensions. This 2-dimensional array [[1,2,3], [4,5,6]], value of **ndim** will be **2**.

```
1  ary.ndim
```

```
2
```

Array Attributes

- **shape**:- It contains a tuple specifying an array's dimensions. Array [[1,2,3], [4,5,6]], value of **shape** will be (2,3).

```
1  ary.shape
```

```
(2, 3)
```

- **size**:- It can view an array's total number of elements. for this 2-dimensional array [[1,2,3], [4,5,6]], **size** will be multiplication of 2 and 3 (Shape=(2,3)) 2*3 = 6.

```
1  ary.size
```

```
6
```

- **itemsize**:- the number of bytes required to store each element with itemsize. This NumPy array [[1,2,3], [4,5,6]], **itemsize** will be 8, because this array consists of integers and size of integer (in bytes) is 8 bytes.

```
1  ary.itemsize
```

```
8
```

Array Operation

Operator	Equal Function	Operator	Equal Function
+	np.add	>	np.greater
-	np.subtract	>=	np.greater_equal
*	np.multiply	<	np.less
/	np.divide	<=	np.less_equal
//	np.floor_divide	==	np.equal
**	np.power	!=	np.not_equal
%	np.mod		

Array Operation

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

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

```
1   ary>5
```

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

```
1   ary<=5
```

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

```
1   ary==5
```

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

NumPy Methods

- An array has various methods that perform calculations using its contents. By default, these methods ignore the array's shape and use all the elements in the calculations.

  For example, calculating the mean of an array totals all of its elements regardless of its shape, then divides by the total number of elements.

- Few NumPy default methods such as **sum**, **min**, **max**, **mean**, **std** (standard deviation) and **var** (variance).

- Specifying **axis=0** performs the mean calculation on all the row values within each column and **axis=1** performs the mean calculation on all the column values within each individual row.

NumPy Methods

```
[82]  1 grades = np.array([[87, 96, 70], [100, 87, 90],
      2                    [94, 77, 90], [100, 81, 82]])
```

```
[83]  1 grades
```

```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

```
[84]  1 grades.max()
```

```
100
```

```
[85]  1 grades.min()
```

```
70
```

```
[87]  1 grades.sum()
```

```
1054
```

```
 1 grades.std()
```

```
8.792357792739987
```

```
 1 grades.mean()
```

```
87.83333333333333
```

```
 1 grades.var()
```

```
77.30555555555556
```

```
 1 grades.mean(axis=0)
```

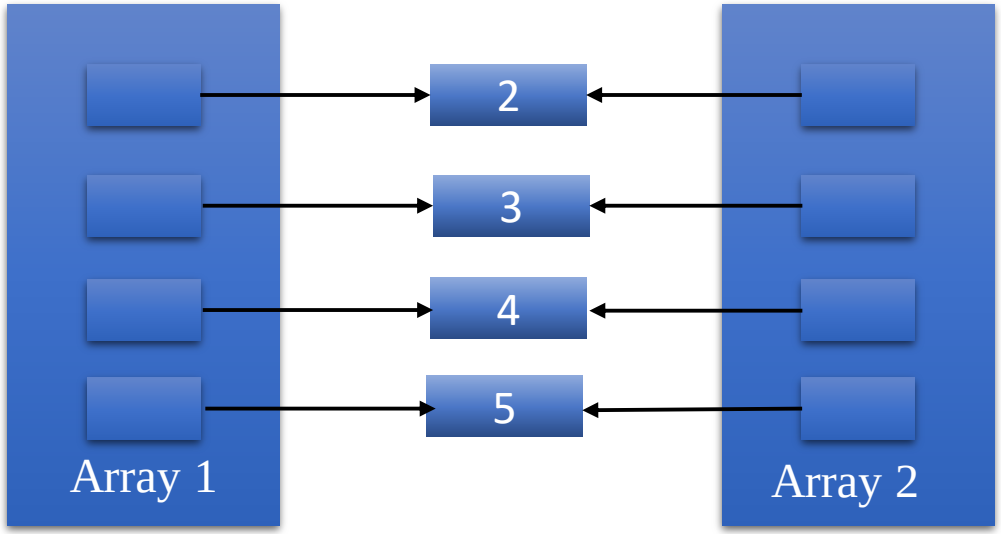
```
array([95.25, 85.25, 83.  ])
```

```
 1 grades.mean(axis=1)
```

```
array([84.33333333, 92.33333333, 87.        , 87.66666667])
```

Shallow Copies

- A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original.
- In case of shallow copy, a reference of object is copied in other object. It means that any changes made to a copy of object do reflect in the original object.



Shallow Copies

```
1   num1 = np.arange(1,6)
2   num1
```

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

```
1   num2 = num1.view()
2   num2
```

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

```
1   print(id(num1))
2   print(id(num2))
```

```
140306878771072
140306880697088
```

```
1   num2[1]*=5
```

```
1   num1
```

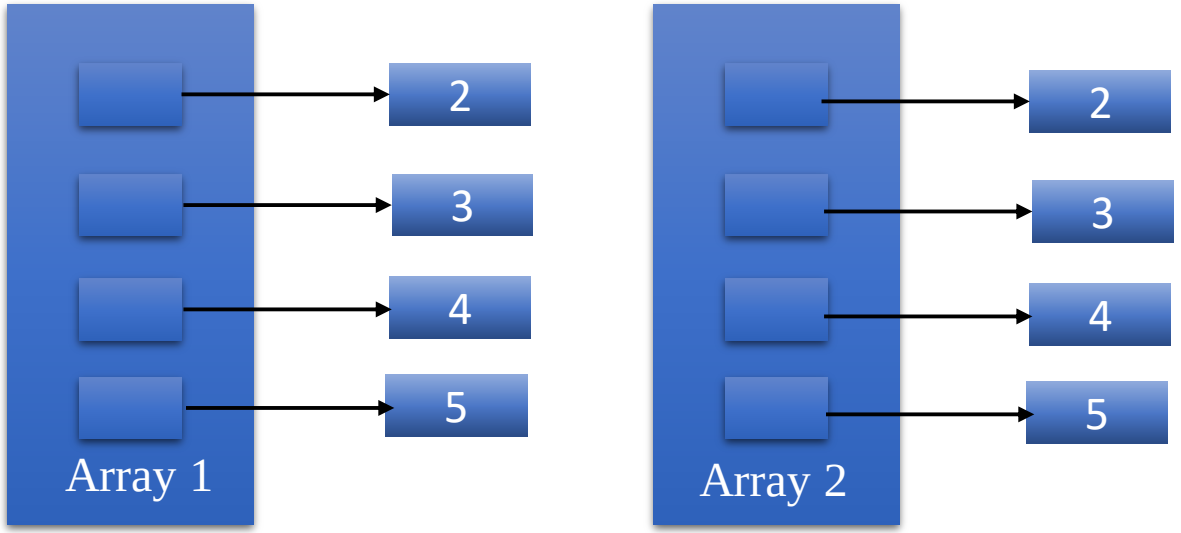
```
array([ 1, 10,  3,  4,  5])
```

```
1   num2
```

```
array([ 1, 10,  3,  4,  5])
```

Deep Copies

- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.
- In case of deep copy, a copy of object is copied in other object. It means that any changes made to a copy of object do not reflect in the original object.



Deep Copies

```
1   num1 = np.arange(1,6)
2   num1
```

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

```
1   num2 = num1.copy()
2   num2
```

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

```
1   num2[2]*=5
2   num2
```

```
array([ 1,  2, 15,  4,  5])
```

```
1   num1
```

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

Thank You