ADVANCED PYTHON PROGRAMMING

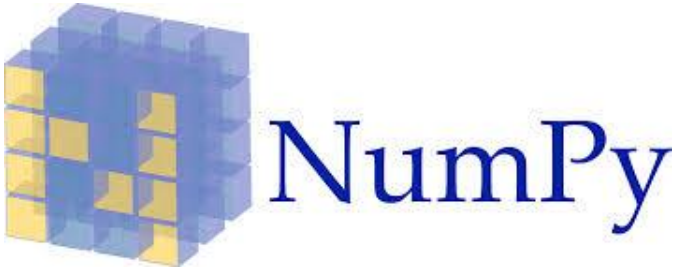
Array Processing by NumPy

Peter Lo

Overview

- NumPy stand for **Num**eric **Py**thon. It is the fundamental package for scientific computing with Python. It contains:
  - A powerful N-dimensional array object
  - Sophisticated (broadcasting) functions
  - Tools for integrating C/C++ and Fortran code
  - Useful linear algebra, Fourier transform, and random number capabilities

NumPy is often used along with packages like SciPy (Scientific Python) and Mat−plotlib (plotting library). This combination is widely used as a replacement for MatLab

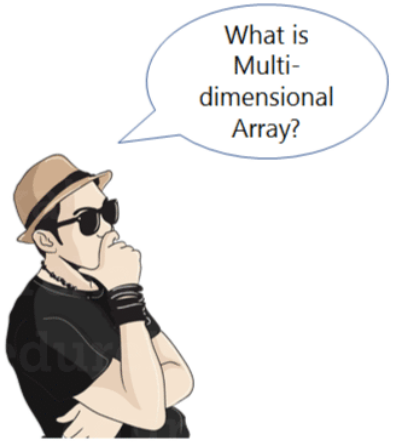


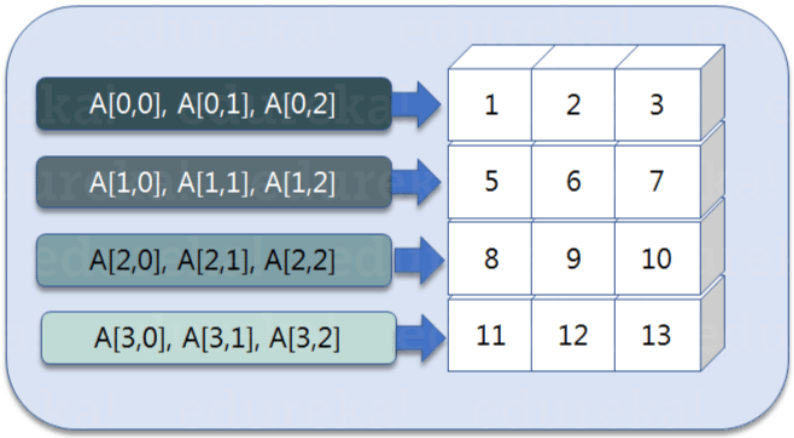
Why choose NumPy?

- **Much Faster:** NumPy uses algorithms written in C that complete in nanoseconds rather than seconds.
- **Fewer Loops:** NumPy helps you to reduce loops and keep from getting tangled up in iteration indices.
- **Clearer Code:** Without loops, your code will look more like the equations you're trying to calculate.
- **Better Quality:** There are thousands of contributors working to keep NumPy fast, friendly, and bug free.

NumPy Array

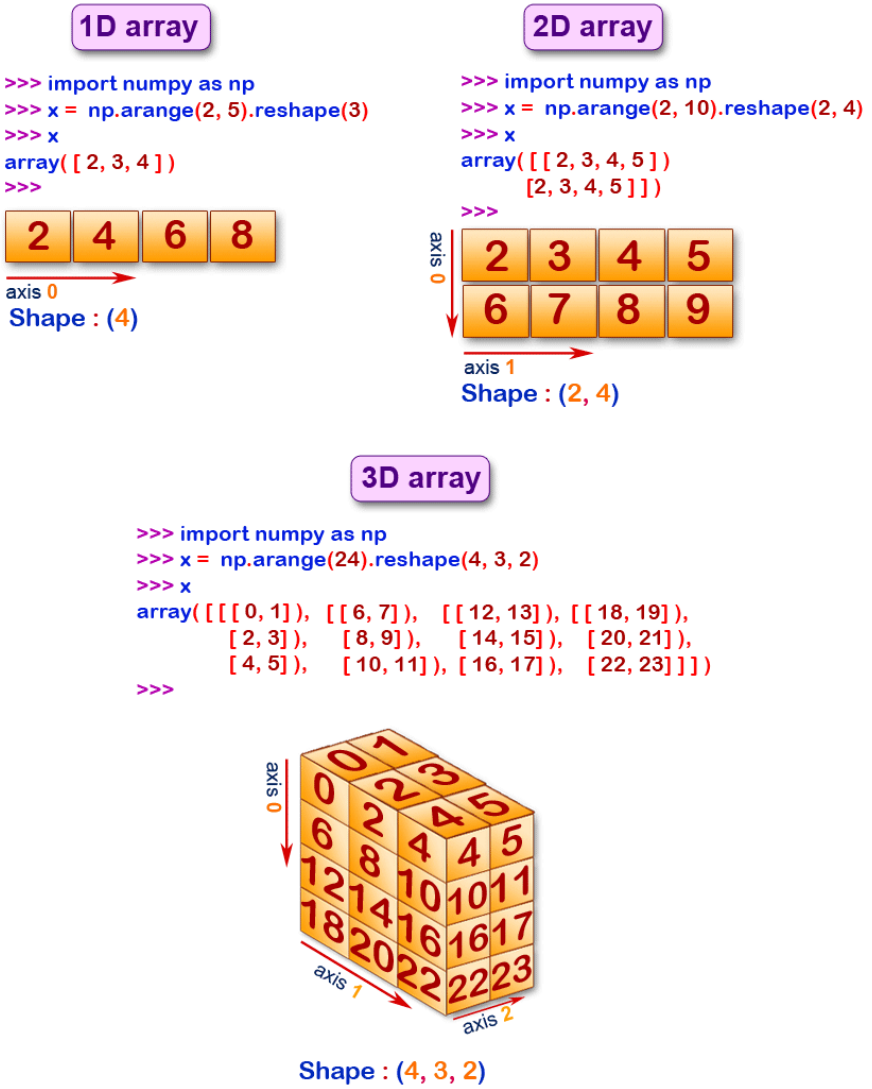
- The most important object defined in NumPy is an N-dimensional array type called ndarray.
- ndarray is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers.
- The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.





NumPy Axis

- Axes are defined for arrays with more than one dimension.
- A 2-dimensional array has two corresponding axes:
  - The first running vertically downwards across rows (axis 0)
  - The second running horizontally across columns (axis 1)



Using NumPy Module

- Before we can use NumPy we will have to import it. It has to be imported like any other module:

```
import numpy
```

- But you will hardly ever see this. Numpy is usually renamed to np:

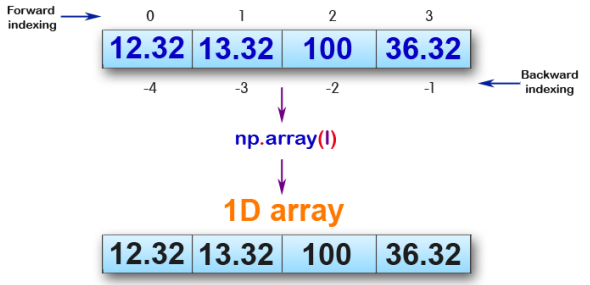
```
import numpy as np
```

Create Array from List

- We can initialize numpy arrays from nested Python lists, and access elements using square brackets

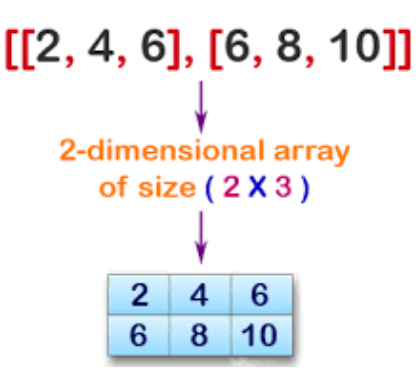
```
import numpy as np
```

```
# Create one dimension array from list
SimpleArray = np.array([12.32, 13.32, 100, 36.32])

# Print the Array and type
print(SimpleArray, type(SimpleArray))
```




```
[ 12.32  13.32 100.    36.32] <class 'numpy.ndarray'>
```

```
# Create two dimension array from list
SimpleArray = np.array([[2, 4, 6], [6, 8, 10]])

# Print the Array and type
print(SimpleArray, type(SimpleArray))
```




```
[[ 2  4  6]
 [ 6  8 10]] <class 'numpy.ndarray'>
```

Create Array by arrange

- This method ***arange*** returns an ndarray object containing evenly spaced values within a given range.

```
# Create an arrary with evenly spaced numbers over a specified interval.
SimpleArray = np.arange(5)
print(SimpleArray)
```

```
[0 1 2 3 4]
```

```
# Create an arrary with evenly spaced numbers over a specified interval.
SimpleArray = np.arange(1,5)
print(SimpleArray)
```

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

```
# Create an arrary with evenly spaced numbers over a specified interval.
SimpleArray = np.arange(1,10,3)
print(SimpleArray)
```

```
[1 4 7]
```

Create Array

- Numpy also provides many functions to create arrays

```
# Create an arrary with evenly spaced numbers over a specified interval.
SimpleArray = np.linspace(1,5,3)
print(SimpleArray)
```

Create a array with 3 values between 1 to 5.

```
[1. 3. 5.]
```

```
# Create an array filled with random values
SimpleArray = np.random.random((2,3))
print(SimpleArray)
```

```
[[0.3287198  0.23723362 0.83279085]
 [0.26649984 0.72562044 0.58184733]]
```

```
# Create an array of all zeros
SimpleArray = np.zeros((2,3))
print(SimpleArray)
```

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

Create Array (Cont.)

```
# Create an array of all ones
SimpleArray = np.ones((2,3))
print(SimpleArray)
```

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

```
# Create a constant array
SimpleArray = np.full((2,3), 7)
print(SimpleArray)
```

```
[[7 7 7]
 [7 7 7]]
```

```
# Create a 2x2 identity matrix
SimpleArray = np.eye(2)
print(SimpleArray)
```

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

NumPy Array vs. Python List

- The key difference between NumPy array and a list is, arrays are designed to handle vectorized operations while a python list is not. If you apply a function it is performed on every item in the array, rather than on the whole array object.
- Another characteristic is that, once a Numpy array is created, you cannot increase its size. Such a behavior of extending the size is natural in a list

Example

```
import numpy as np

# Define a List
SimpleList = [1, 2, 3, 4]

# Define an Array
SimpleArray = np.array(SimpleList)

# Print the List with formula
print (SimpleList, SimpleList * 2)

# Print the Array with formula
print (SimpleArray, SimpleArray * 2)
```

```
[1, 2, 3, 4] [1, 2, 3, 4, 1, 2, 3, 4]
[1 2 3 4] [2 4 6 8]
```

Copying Array

- If you just assign a portion of an array to another array, the new array you just created actually refers to the parent array in memory. That means, if you make any changes to the new array, it will reflect in the parent array as well.
- To avoid disturbing the parent array, you need to make a copy of it using the ***copy*** method.

Copying Array (Cont.)

```
# Create an 3x4 array
SimpleArray = np.array([[ 1,  2,  3,  4],
                        [ 3,  4,  5,  6],
                        [ 5,  6,  7,  8]])
print(SimpleArray)
```

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

```
NewArray = SimpleArray[:2, :3]
NewArray[0, 0] = 999
print(NewArray)
```

If using "=" to copy array, you make any changes to the new array, it will reflect in the parent array as well.

```
[[999   2   3]
 [  3   4   5]]
```

```
print(SimpleArray)
```

```
[[999   2   3   4]
 [  3   4   5   6]
 [  5   6   7   8]]
```

Copying Array (Cont.)

```
# Create an 3x4 array
SimpleArray = np.array([[ 1,  2,  3,  4],
                        [ 3,  4,  5,  6],
                        [ 5,  6,  7,  8]])
print(SimpleArray)
```

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

```
# Create a new 2x3 array by copy
NewArray = SimpleArray[:2, :3].copy()
NewArray[0, 0] = 999
print(NewArray)
```

```
[[999   2   3]
 [  3   4   5]]
```

```
print(SimpleArray)
```

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

Array Dimension

- The ***ndim*** method is used to find the dimension of the array, whether it is a two-dimensional array or a single dimensional array.

```
import numpy as np
```

```
# Define an Array
SimpleArray = np.array([1, 2, 3, 4, 5, 6])

# Print the Array dimension
print(SimpleArray.ndim)
```

```
1
```

```
# Define an Array
SimpleArray = np.array([(1,2,3),(4,5,6)])

# Print the Array dimension
print(SimpleArray.ndim)
```

```
2
```

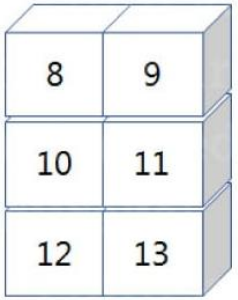
Reshape Array

- The ***reshape*** method gives a new shape to an array without changing its data,
  - i.e. it returns a new array with a new shape.

```
# Create an arrary
SimpleArray = np.array([(8,9,10), (11,12,13)])
print(SimpleArray)
```

```
[[ 8  9 10]
 [11 12 13]]
```

```
# Resharp it
SimpleArray = SimpleArray.reshape(3,2)
print(SimpleArray)
```

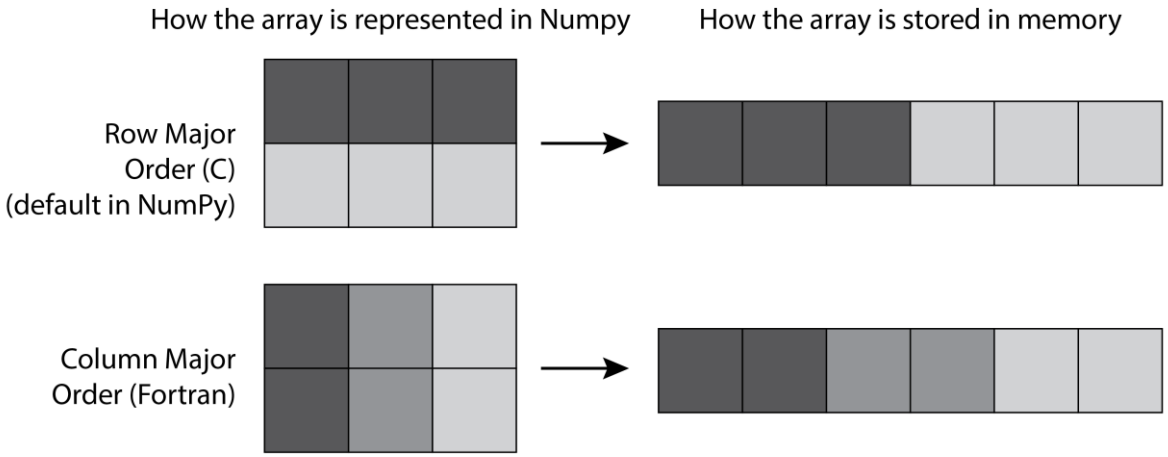



```
[[ 8  9]
 [10 11]
 [12 13]]
```

Flatten Array

- The ***flatten*** method returns a copy of an array collapsed into one dimension with an optional parameter “Order”.

Order	Description
C	Flatten in row-major (C-style) order, default setting
F	Flatten in column-major (Fortran-style) order
A	Flatten in column-major order if it is Fortran contiguous in memory, row-major order otherwise
K	Flatten in the order the elements occur in the memory



Flatten Array Example

```
import numpy as np
```

```
# Create an arrary
SimpleArray = np.array([(8,9,10), (11,12,13)])
print(SimpleArray)
```

```
[[ 8  9 10]
 [11 12 13]]
```

```
# Flatten in row-major (C-style) order
SimpleArray.flatten()
```

```
array([ 8,  9, 10, 11, 12, 13])
```

```
# Flatten in column-major (Fortran-style) order
SimpleArray.flatten("F")
```

```
array([ 8, 11,  9, 12, 10, 13])
```

Ravel Array

- This function returns a flattened one-dimensional array. A copy is made only if needed. The returned array will have the same type as that of the input array.

Order	Description
C	Flatten in row-major (C-style) order, default setting
F	Flatten in column-major (Fortran-style) order
A	Flatten in column-major order if it is Fortran contiguous in memory, row-major order otherwise
K	Flatten in the order the elements occur in the memory

Ravel Array Example

```
import numpy as np
```

```
# Create an array
SimpleArray = np.array([(8,9,10), (11,12,13)])
print(SimpleArray)
```

```
[[ 8  9 10]
 [11 12 13]]
```

```
# Flatten in row-major (C-style) order
SimpleArray.ravel()
```

```
array([ 8,  9, 10, 11, 12, 13])
```

```
# Flatten in column-major (Fortran-style) order
SimpleArray.ravel("F")
```

```
array([ 8, 11,  9, 12, 10, 13])
```

Differences between Flatten and Ravel

Flatten

- Return copy of original array
- If you modify any value of this array value of original array is not affected.
- Flatten is comparatively slower than ravel as it occupies memory.
- Flatten is a method of an ndarray object.

Ravel

- Return only reference/view of original array
- If you modify the array you would notice that the value of original array also changes.
- Ravel is faster than flatten as it does not occupy any memory.
- Ravel is a library-level function.

Example

```
import numpy as np
```

```
# Create an array
SimpleArray = np.array([(8,9,10), (11,12,13)])
print(SimpleArray)
```

```
[[ 8  9 10]
 [11 12 13]]
```

```
# Flatten array and modify value
NewArray = SimpleArray.flatten()
NewArray[0] = 999
```

```
print(NewArray)
```

```
[999   9  10  11  12  13]
```

```
print(SimpleArray)
```

```
[[ 8  9 10]
 [11 12 13]]
```

```
# Ravel array and modify value
NewArray = SimpleArray.ravel()
NewArray[0] = 999
```

```
print(NewArray)
```

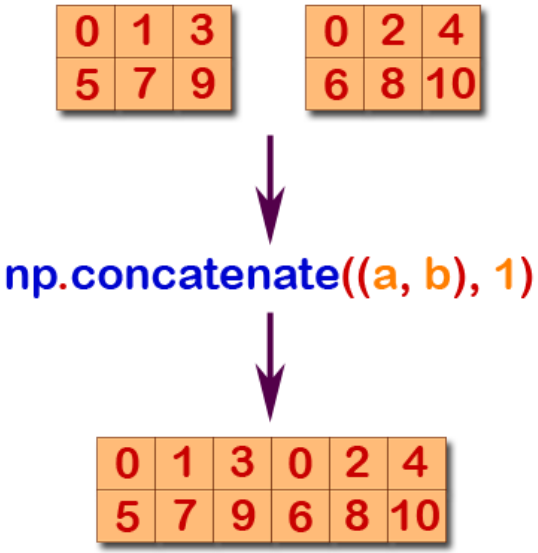
```
[999   9  10  11  12  13]
```

```
print(SimpleArray)
```

```
[[999   9  10]
 [ 11  12  13]]
```

Concatenating Arrays

- Concatenation refers to joining. The ***concatenate*** method is used to join two or more arrays of the same shape along a specified axis.
- The axis parameter specifies the index of the new axis in the dimensions of the result.
  - For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension.



Concatenating Arrays Example

```
# Create first array
FirstArray = np.array([(0,1,3),(5,7,9)])
print(FirstArray)
```

```
[[0 1 3]
 [5 7 9]]
```

```
# Create second array
SecondArray = np.array([(0,2,4),(6,8,10)])
print(SecondArray)
```

```
[[ 0  2  4]
 [ 6  8 10]]
```

```
# Concatenate Array (vertically)
ResultArray = np.concatenate((FirstArray,SecondArray),axis=0)
print(ResultArray)
```

```
[[ 0  1  3]
 [ 5  7  9]
 [ 0  2  4]
 [ 6  8 10]]
```

```
# Concatenate Array (horizontally)
ResultArray = np.concatenate((FirstArray,SecondArray),axis=1)
print(ResultArray)
```

```
[[ 0  1  3  0  2  4]
 [ 5  7  9  6  8 10]]
```

Extract Specific Items from Array

- You can extract specific portions on an array using indexing starting with 0, something similar to how you would do with python lists.
- But unlike lists, numpy arrays can optionally accept as many parameters in the square brackets as there is number of dimensions.
- Numpy offers several ways to index into arrays.
  - Slicing
  - Integer Array Indexing
  - Boolean Array Indexing

Slicing

- Slicing is basically extracting particular set of elements from an array.
- This slicing operation is pretty much similar to the one which is there in the list as well.

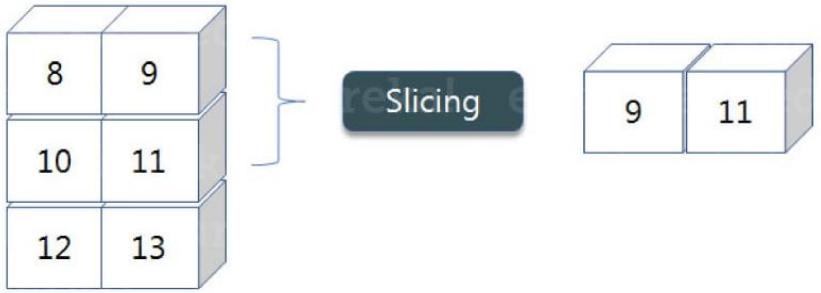
```
import numpy as np
```

```
# Create an arrary
SimpleArray = np.array([(8,9),(10, 11),(12,13)])
print(SimpleArray)
```

```
[[ 8  9]
 [10 11]
 [12 13]]
```

```
# Slicing
print(SimpleArray[0:2,1])
```

```
[ 9 11]
```



Integer Array Indexing

- When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array.
- Integer array indexing allows you to construct arbitrary arrays using the data from another array.

```
# Create an 3x2 array
SimpleArray = np.array([[1,2], [3, 4], [5, 6]])
print(SimpleArray)
```

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

```
print(SimpleArray[[0, 1, 2], [0, 1, 0]])
```

```
[1 4 5]
```

The row index contains all row numbers, and the column index specifies the element to be selected

Boolean Array Indexing

- Boolean array indexing lets you pick out arbitrary elements of an array.
- Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
# Create an array
SimpleArray = np.array([[1,2], [3, 4], [5, 6]])
print(SimpleArray)
```

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

```
ResultArray = (SimpleArray > 2)
print(ResultArray)
```

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

Find the elements of a that are bigger than 2; this returns a numpy array of Booleans of the same shape as a, where each slot of result tells whether that element of a is > 2.

Reverse Array

- Reversing an array works like how you would do with lists, but you need to do for all the axes (dimensions) if you want a complete reversal

```
# Create an array
SimpleArray = np.array([[ 1,  2,  3,  4],
                        [ 3,  4,  5,  6],
                        [ 5,  6,  7,  8]])
print(SimpleArray)
```

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

```
# Reverse only the row positions
print(SimpleArray[::-1, ])
```

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

Reverse Array (Cont.)

```
import numpy as np
```

```
# Create an array
SimpleArray = np.array([[ 1,  2,  3,  4],
                        [ 3,  4,  5,  6],
                        [ 5,  6,  7,  8]])
print(SimpleArray)
```

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

```
# Reverse the row and column positions
SimpleArray[::-1, ::-1]
```

```
array([[8, 7, 6, 5],
       [6, 5, 4, 3],
       [4, 3, 2, 1]])
```

Mathematics Operation

- The ndarray has the respective methods to compute for the whole array

```
# Create an array
SimpleArray = np.array([[ 1,  2,  3,  4],
                        [ 3,  4,  5,  6],
                        [ 5,  6,  7,  8]])
print(SimpleArray)
```

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

```
# mean, max and min
print("Mean value is: ", SimpleArray.mean())
print("Max value is: ", SimpleArray.max())
print("Min value is: ", SimpleArray.min())
```

```
Mean value is:  4.5
Max value is:  8
Min value is:  1
```

Row-wise / Column-wise Calculation

- If you want to compute the minimum values row wise or column wise, use the ***amin*** version instead.

```
# Create an array
SimpleArray = np.array([[ 1,  2,  3,  4],
                        [ 3,  4,  5,  6],
                        [ 5,  6,  7,  8]])
print(SimpleArray)
```

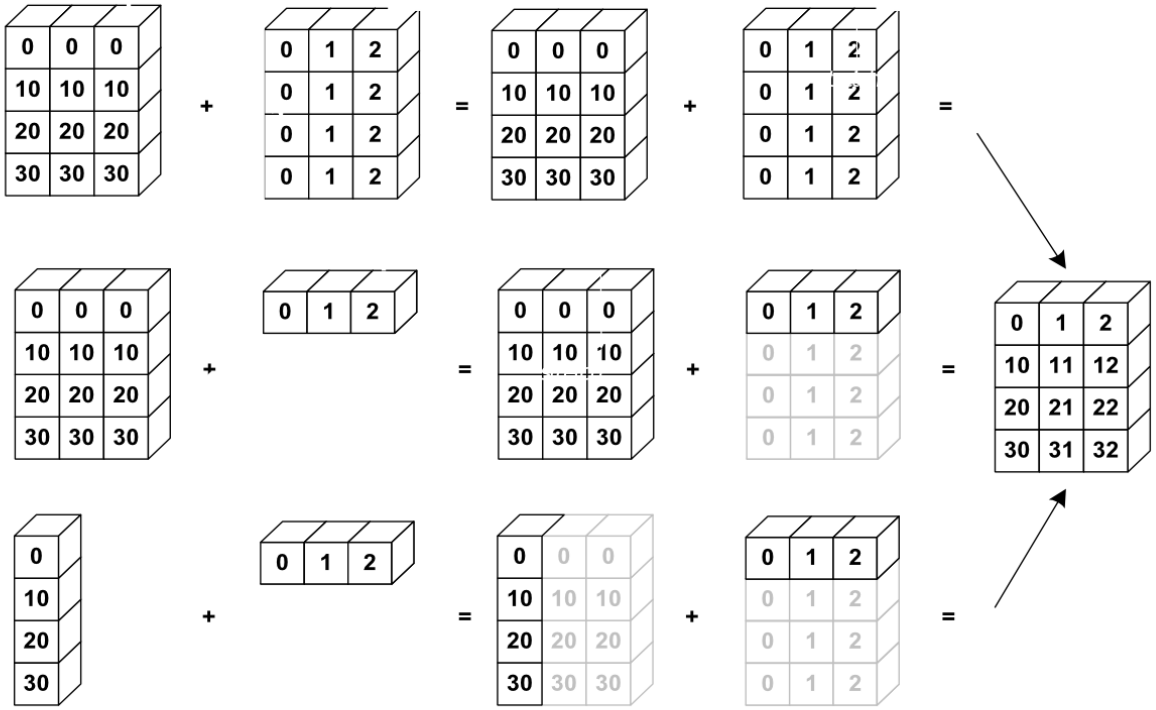
```
[[1 2 3 4]
 [3 4 5 6]
 [5 6 7 8]]
```

```
# Row wise and column wise min
print("Column wise minimum: ", np.amin(SimpleArray, axis=0))
print("Row wise minimum: ", np.amin(SimpleArray, axis=1))
```

```
Column wise minimum:  [1 2 3 4]
Row wise minimum:  [1 3 5]
```

Broadcasting

- Basic operations on numpy arrays (addition, etc.) are elementwise.
- It's also possible to do operations on arrays of different sizes if Numpy can transform these arrays so that they all have the same size: this conversion is called broadcasting.



Broadcasting Example

```
import numpy as np
```

```
# Create an array
FirstArray = np.array([[0], [1], [2]])
print(FirstArray)
```

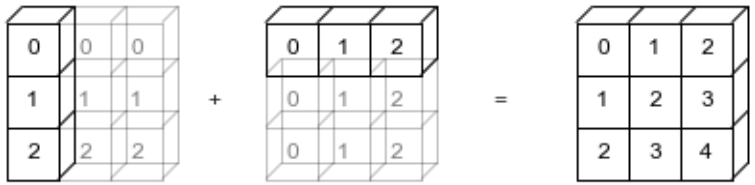
```
[[0]
 [1]
 [2]]
```

```
SecondArray = np.array([0,1,2])
print(SecondArray)
```

```
[0 1 2]
```

```
ResultArray = FirstArray + SecondArray
print(ResultArray)
```

```
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```



Get Unique Items and Counter

- The ***unique*** method can be used to get the unique items. If you want the repetition counts of each item, set the ***return_counts*** parameter to True.

```
# Create random integers of size 10 between 0-5
SimpleArray = np.random.randint(0, 5, size=10)
print(SimpleArray)
```

```
[4 2 3 3 4 2 4 3 1 0]
```

```
# Get the unique items and their counts
Unique_Item, Counter = np.unique(SimpleArray, return_counts=True)
print("Unique items: ", Unique_Item)
print("Counts      : ", Counter)
```

```
Unique items:  [0 1 2 3 4]
Counts      :  [1 1 2 3 3]
```

Filtering

- By using the Boolean array indexing, NumPy can filter array

```
# Create an Array containing elements from 1 to 30 but at equal interval of 2
SampleArray = np.arange(1, 30, 2)
```

```
# Generate Boolean array indexing by condition
SampleArray>15
```

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

```
# Apply condition for filtering
SampleArray[SampleArray>15]
```

```
array([17, 19, 21, 23, 25, 27, 29])
```

Example

- Filtering with one operator

```
# Create an Array containing elements from 1 to 30 but at equal interval of 2
SampleArray = np.arange(1, 30, 2)

# Print original list
print(SampleArray)
```

```
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29]
```

```
# Filter the Array which are greater than 15
Result = SampleArray[SampleArray > 15]

# Print the Result
print(Result)
```

```
[17 19 21 23 25 27 29]
```

Example 2

- Filtering with AND conditions

```
# Create an Array containing elements from 1 to 30 but at equal interval of 2
SampleArray = np.arange(1, 30, 2)

# Print original list
print(SampleArray)
```

```
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29]
```

```
# Filter the Array which between 5 and 20
Result = SampleArray[(SampleArray >= 5) & (SampleArray <= 20)]

# Print the Result
print(Result)
```

```
[ 5  7  9 11 13 15 17 19]
```

Example 3

- Filtering with OR conditions

```
# Create an Array containing elements from 1 to 30 but at equal interval of 2
SampleArray = np.arange(1, 30, 2)

# Print original list
print(SampleArray)
```

```
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29]
```

```
# Filter the Array which less than 5 or larger than 20
Result = SampleArray[(SampleArray < 5) | (SampleArray <= 20)]

# Print the Result
print(Result)
```

```
[ 1  3  5  7  9 11 13 15 17 19]
```

Example 4

- Filtering with Fancy Indexing

```
# Create an Array containing elements from 1 to 30 but at equal interval of 2
SampleArray = np.array([100, 200, 300])

# Print original list
print(SampleArray)
```

```
[100 200 300]
```

```
# Create Filter Array
FilterArray = np.array([1, 3, 5])

# Print filter array
print(FilterArray)
```

```
[1 3 5]
```

```
# Filter by Fancy Indexing
Result = SampleArray[FilterArray > 2]

# Print the Result
print(Result)
```

```
[200 300]
```

Delete Data from Array

- Rows and columns can be deleted using ***np.delete( )*** and ***np.where( )***.
- In ***np.delete( )***, set the target ***ndarray***, the index to delete and the target axis.
- In the case of a two-dimensional array
  - rows are deleted if axis=0
  - columns are deleted if axis=1
- ***np.where( )*** returns the index of the element that satisfies the condition.
- In the case of a multidimensional array, a tuple of a list of indices (row number, column number) that satisfy the condition for each dimension (row, column) is returned.

Delete Row by Index

```
# Create an Array
SampleArray = np.array([[ 0, 1, 2, 3],
                        [ 4, 5, 6, 7],
                        [ 8, 9, 10, 11]])

# Print original list
print(SampleArray)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
# Delete Row 1 and Row 3
ResultArray = np.delete(SampleArray, [0, 2], axis=0)

# Print the Result
print(ResultArray)
```

```
[[4 5 6 7]]
```

Delete Column by Index

```
# Create an Array
SampleArray = np.array([[ 0, 1, 2, 3],
                        [ 4, 5, 6, 7],
                        [ 8, 9, 10, 11]])

# Print original list
print(SampleArray)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
# Delete Column 1 and Column 3
ResultArray = np.delete(SampleArray, [0, 2], axis=1)

# Print the Result
print(ResultArray)
```

```
[[ 1  3]
 [ 5  7]
 [ 9 11]]
```

Delete Row by Condition

```
# Create an Array
SampleArray = np.array([[ 0, 1, 2, 3],
                        [ 4, 5, 6, 7],
                        [ 8, 9, 10, 11]])

# Print original list
print(SampleArray)
```

```
# Delete Row with Value < 2
ResultArray = np.delete(SampleArray, np.where(SampleArray < 2)[0], axis=0)

# Print the Result
print(ResultArray)
```

```
[[ 0  1  2  3]
 [ 8  9 10 11]]
```

Delete Column by Condition

```
# Create an Array
SampleArray = np.array([[ 0, 1, 2, 3],
                        [ 4, 5, 6, 7],
                        [ 8, 9, 10, 11]])

# Print original list
print(SampleArray)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
# Delete Column with Value < 2
ResultArray = np.delete(SampleArray, np.where(SampleArray < 2)[0], axis=1)

# Print the Result
print(ResultArray)
```

```
[[ 1  2  3]
 [ 5  6  7]
 [ 9 10 11]]
```

Using NumPy to Read Files

- Three main functions available to directly read csv or other files into arrays.
  - ***fromfile*** - A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the ***tofile*** method can be read using this function.
  - ***genfromtxt*** - Load data from a text file, with missing values handled as specified. Each line past the first skip_header lines is split at the delimiter character, and characters following the comments character are discarded.
  - ***loadtxt*** - Load data from a text file. Each row in the text file must have the same number of values.

Reading Text File

- A standard way to import datasets is to use the ***genfromtxt*** function.
- This function can read the text file (Default delimited ir space) into NumPy array

```
# Read the file which is separated by space
SimpleArray = np.genfromtxt("Sample.txt")

# Print the array
print(SimpleArray)
```

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

Reading CSV File with Heading

- By using ***skip_header***, ***delimiter***, and ***dtype = None***, you can read the CSV file with heading and different data type in column.

```
# Read the CSV file with heading
SimpleArray = np.genfromtxt(fname = "Employee.csv",
                            skip_header = 1,
                            delimiter = ",",
                            dtype = None)

# Print the array
print(SimpleArray)
```

```
[(  1, 'Abercrombie', 'Kim', '24/06/1985', '16/10/1962', 'D',  91000, 55)
 (  2, 'Ackerman', 'Pilar', '26/04/1989', '05/10/1950', 'E',  31000, 68)
 (  3, 'Ajenstat', 'François', '01/02/1981', '21/12/1964', 'C',  48000, 53)
 (  4, 'Akers', 'Kim', '29/05/1979', '08/04/1958', 'C',  47000, 60)
 (  5, 'Alberts', 'Amy E.', '15/10/1989', '22/04/1970', 'D',  60000, 48)
 (  6, 'Alderson', 'Gregory F. (Greg)', '12/01/1992', '28/05/1964', 'D', 100000, 54)
 (  7, 'Alexander', 'Sean P', '06/07/2000', '16/10/1962', 'E',  29000, 55)
 (  8, 'Anderson', 'Nancy', '06/06/1986', '20/06/1976', 'C',  68000, 42)
```

Reading CSV from URL

- ***genfromtxt*** can also import datasets from web URLs, handle missing values, multiple delimiters, handle irregular number of columns etc.

```
# Import data from csv file url
path = "https://raw.githubusercontent.com/selva86/datasets/master/Auto.csv"
SimpleArray = np.genfromtxt(path, delimiter=",", skip_header=1,
                            filling_values=-999, dtype="float")

# Print the array
print(SimpleArray)
```

```
[[  18.    8.  307. ...   70.    1. -999.]
 [  15.    8.  350. ...   70.    1. -999.]
 [  18.    8.  318. ...   70.    1. -999.]
 ...
 [  32.    4.  135. ...   82.    1. -999.]
 [  28.    4.  120. ...   82.    1. -999.]
 [  31.    4.  119. ...   82.    1. -999.]]
```

Save Data to Text File

- Python's Numpy module provides a function to save numpy array to a txt file with custom delimiters and other custom options.

Argument	Description
arr	1D or 2D numpy array (to be saved)
fmt	A formatting pattern or sequence of patterns, that will be used while saving elements to file. • If a single formatter is specified like '%d' then it will be applied to all elements. • In case of 2D arrays, a list of specifier i.e. different for each column. (Optional)
delimiter	String or character to be used as element separator (Optional)
newline	String or character to be used as line separator (Optional)
header	String to be written at the beginning of the txt file.
footer	String to be written at the end of the txt file.
comments	Custom comment marker , default is '#'. Will be pre-appended to the header and footer

Save Data to Text File

- If using ***savetxt( )*** without any parameters, the output format will be in scientific format

```
# Create an arry from 1 to 5
SimpleArray = np.arange(5)

# Save the data to file
np.savetxt("output.txt", SimpleArray)
```

- Output file

```
0.00000000000000000e+00
1.00000000000000000e+00
2.00000000000000000e+00
3.00000000000000000e+00
4.00000000000000000e+00
```

Writing Data to File with Format

- By using *fmt* parameter, you can define the output

```
# Create an arry from 1 to 5
SimpleArray = np.arange(5)

# Save the data to file (Integer Format)
np.savetxt("output.txt", SimpleArray, fmt="%d")
```

```
0
1
2
3
4
```

Writing Data to File with Header/Footer

- By using ***header*** and ***footer*** parameters, you can add header and footer to the file

```
# Create an arry from 1 to 5
SimpleArray = np.arange(5)

# Save the data to file (Integer Format)
np.savetxt("output.txt",
           SimpleArray,
           delimiter=',',
           fmt = "%d",
           header = "Header Line",
           footer = "Footer Line")
```

```
# Header Line
0
1
2
3
4
# Footer Line
```

Case Study: S&P 100

- Within the S&P 100, companies are associated with specific sectors. For example, the largest sector is made up of companies associated with the consumer discretionary sector. The next largest sectors are information technology, healthcare, and financial sectors.
- In this case study, we'll be analyzing all the S&P 100 companies as well as sector specific companies.
- For each company, we have data on its name, sector, stock price, and earnings per share, abbreviated EPS. The earnings per share is the profit for each share of stock.
- Our objective for the first part of our case study is to analyze growth expectations of companies within the S&P 100 by calculating the price to earnings ratio of each company.

Pandas vs. NumPy

PANDAS	NUMPY
When we have to work on **Tabular data**, we prefer the p*andas* module.	When we have to work on **Numerical data**, we prefer the n*umpy* module.
The powerful tools of pandas are **Data frame and Series.**	Whereas the powerful tool of *numpy* is **Arrays.**
Pandas consume **more memory**.	*Numpy* is **memory efficient.**
Pandas has a better performance when a number of rows is **500K or more.**	*Numpy* has a better performance when number of rows is **50K or less.**
Indexing of the *pandas* series is **very slow** as compared to *numpy* arrays.	Indexing of *numpy* Arrays is **very fast**.
Pandas offer a have2d table object called **DataFrame.**	*Numpy* is capable of providing **multi-dimensional arrays.**
It was developed by Wes McKinney and was released in 2008.	It was developed by Travis Oliphant and was released in 2005.
It is used in a lot of organizations like Kaidee, Trivago, Abeja Inc. , and a lot more.	It is being used in organizations like Walmart Tokopedia, Instacart, and many more.
It has a higher industry application.	It has a lower industry application.