NumPy Arrays

- Before proceeding towards Pandas’ data structure, let us have a brief review of NumPy arrays because-
  1. Pandas’ some functions return result in form of NumPy array.
  2. It will give you a jumpstart with data structure.
- NumPy (“Numerical Python” or Numeric Python”) is an open source module of Python that provides functions for fast mathematical computation on arrays and matrices.
- To use NumPy, it is needed to import. Syntax for that is-

```
>>>import numpy as np
```

(here np, is an alias for numpy which is optional)

```
>>> import numpy as np
>>> lst = [1,2,3,4]
>>> a1=np.array(lst)
>>> lst
[1, 2, 3, 4]
>>> print(a1)
[1 2 3 4]
>>> a1
array([1, 2, 3, 4])
>>>
```

See the difference between List and array

- NumPy arrays come in two forms-
  - 1-D array – also known as Vectors.
  - Multidimentional arrays – Also known as Matrices.

NumPy Arrays Vs Python Lists

- Although NumPy array also holds elements like Python List , yet Numpy arrays are different data structures from Python list. The key differences are-
- Once a NumPy array is created, you cannot change its size. you will have to create a new array or overwrite the existing one.
- NumPy array contain elements of homogenous type, unlike python lists.
- An equivalent NumPy array occupies much less space than a Python list.
- NumPy array supports Vectorized operation, i.e. you need to perform any function on every item one by one which is not in

```
>>> import numpy as np
>>> lst=[1,2,3,4]
>>> a1=np.array(lst)
>>> lst+2
Traceback (most recent
  File "<pyshell#4>", l
```

In list, it will generate error but will be executed in arrays.

```
>>> a1+2
array([3, 4, 5, 6])
```

NumPy Data Types

NumPy supports following data types-

S.No.	Data Type	Description	Size
1.	bool_	Boolean data type (stores *True* or *False*)	1 byte
2.	int_	Default type to store integers in *int32* or *int64*	4 or 8 bytes
3.	int8	Stores signed integers in range –128 to 127	1 byte
4.	int16	Stores signed integers in range –32768 to 32767	2 bytes
5.	int32	Stores signed integers in range -2^{16} to $2^{16}-1$	4 bytes
6.	int64	Stores signed integers in range -2^{32} to $2^{32}-1$	8 bytes
7.	uint8	Stores unsigned integers in range 0 to 255	1 byte
8.	uint16	Stores integers in range 0 to $2^{16}-1$	2 bytes
9.	uint32	Stores integers in range 0 to $2^{32}-1$	4 bytes
10.	uint64	Stores integers in range 0 to $2^{64}-1$	8 bytes
11.	float_	Default type to store floating point (*float64*)	8 bytes
12.	float16	Stores **half precision floating point values** (*5 bits* exponent, *10 bit* mantissa)	2 bytes
13.	float32	Stores **single precision floating point values** (*8 bits* exponent, *23 bit* mantissa)	4 bytes
14.	float64	Stores **double precision floating point values** (*11 bits* exponent, *52 bit* mantissa)	8 bytes
15.	complex_	Default type to store complex numbers *(complex128)*	16 bytes
16.	complex64	Complex numbers represented by *two float32 numbers* for *real* and *imaginary* value components.	8 bytes
17.	complex128	Complex numbers represented by *two float64numbers* for *real* and *imaginary* value components.	16 bytes
18.	string_	Fixed-length string type.	1 byte per character
19.	unicode_	Fixed-length Unicode type.	number of bytes platform specific

Ways to Create NumPy Arrays

1. array() function can be used to create array-

numpy.array(<arrayconvertible object>,[<datatype>])

*Assuming NumPy has been imported as np.

```
>>> import numpy as np
>>> nar1=np.array([2,5.2,1.0])
>>> nar1
array([2. , 5.2, 1. ])
>>>
```

Above statement will do the following things-

- nar1 will be created as an ndarray object.
- nar1 will have 3 elements (as passed in the list).
- A datatype will be assigned by default to the elements of the ndarray. You can specify own datatype using dtype argument.
- Itemsize will be as per the datatype of the elements.

Creating array by specifying own datatype-

```
>>> l=[1,2,3,4]
>>> ar=np.array(l,dtype=np.int64)
>>> ar
array([1, 2, 3, 4], dtype=int64)
>>>
```

To check type, dtype and size-

```
>>> l=[1,2,3,4]
>>> ar=np.array(l,dtype=np.int64)
>>> ar
array([1, 2, 3, 4], dtype=int64)
>>> print(type(ar))
<class 'numpy.ndarray'>
>>> print(ar.dtype)
int64
>>> print(ar.itemsize)
8
>>>
```

Ways to Create NumPy Arrays

2. Creating ndarray using fromiter()-

The fromiter() function is useful when you want to create an ndarray from a non-numeric sequence.

numpy. fromiter (<iterable sequence name>,<target data type>,[<count>])

```
>>> ad={1:"A",2:"B",3:"C",4:"D",5:"E"}
>>> ar2=np.fromiter(ad,dtype=np.int32)
>>> print(ar2)
[1 2 3 4 5]
>>> ar2.dtype
dtype('int32')
>>> ar2.itemsize
4
>>> print(ar2[0],ar2[3])
1 4
```

Ways to Create NumPy Arrays

3. arange( ) function is used to create array from a range.

<arrayname> = numpy.arange([start],stop,[step],[dtype])

```
>>> import numpy as np
>>> arr1=np.arange(5)
>>> arr1
array([0, 1, 2, 3, 4])
```

Here, only stop value is passed.

```
>>> import numpy as np
>>> arr2=np.arange(1,7,2,dtype=np.float32)
>>> arr2
array([1., 3., 5.], dtype=float32)
```

Here, from 1-7 at the step of 2.

4. linspace( ) function can be used to prepare array of range.

<arrayname> = numpy.linspace([start],stop,[dtype])

```
>>> import numpy as np
>>> arr3 = np.linspace(2,3,6)
>>> arr3
array([2. , 2.2, 2.4, 2.6, 2.8, 3. ])
```

Here, an array of 6 values is created between the values 2 and 3.

```
>>> import numpy as np
>>> arr4 = np.linspace(2.5,5,8)
>>> arr4
array([2.5       , 2.85714286, 3.21428571, 3.57142857, 3.92857143,
       4.28571429, 4.64285714, 5.        ])
```

Here, an array of 8 values is created between the values 2.5 and 8.

2D NumPy Arrays

```
>>> import numpy as np
>>> A=np.array([[10,11,12,13],[21,22,23,24]])
>>> A[1,3]
24
>>> A[1][3]
24
>>> A
array([[10, 11, 12, 13],
       [21, 22, 23, 24]])
>>> print(A)
[[10 11 12 13]
 [21 22 23 24]]
>>> type(A),type(a1)
(<class 'numpy.ndarray'>, <class 'numpy.ndarray'>)
>>> a1.shape
(4,)
>>> A.shape
(2, 4)
>>> A.itemsize
4
```

With the help of list, 2D array is created.

Accessing Array elemets with index

Printing of Array

To see type of Array

To see shape of Array (use of different functions)

NumPy arrays arr also known as ndarray (n-dimentional array)

2. Creating 2D ndarrays using arange()-

In the similar manner as we have created 1D ndarrays with arange() function, we can also create 2D ndarrays with arange() function along with reshape().

<ndarray>.reshape(<shape tuple>)

```
>>> ar1=np.arange(10)
>>> ar1
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> ar3=ar1.reshape(5,2)
>>> ar3
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

3. ARRAYS CREATION ALTERNATIVE METHODS-

a. Using empty()-

empty() function can be used to create empty array or an unintialized array of specified shape and dtype.

numpy.empty(Shape,[dtype=<datatype>,] [order = ‘C’ or ‘F’]

Where:d**type: is a data type of python or numpy to set initial values.**

Shape: is dimension.

Order : ‘C’ means arrangement of data as row wise(C means C like).

Order : ‘F’ means arrangement of data as row wise (F means Fortran like)

```
>>> arr1=np.empty([2,3],dtype=np.int64, order='C')
>>> arr1
array([[0, 0, 0],
       [0, 0, 0]], dtype=int64)
```

b. Using zeros()-

numpy.zeros (Shape,[dtype=<datatype>,] [order = ‘C’ or ‘F’])

```
>>> ar3=np.zeros([2,3],dtype=np.int64,order='F')
>>> ar3
array([[0, 0, 0],
       [0, 0, 0]], dtype=int64)
```

c. Using ones()-

numpy.ones(Shape,[dtype=<datatype>,] [order = ‘C’ or ‘F’])

```
>>> ar4=np.ones([2,3],dtype=np.float32)
>>> ar4
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

Array Slicing-

it is possible to extract subsets of NumPy arrays using slices, just like lists.

<Arrayname>[<start>:<stop>:<step>]

- When <start><stop> or<step> values are not specified then Python will assume their default values as:
  Start=0; Stop=dimension size ;Step=1

```
>>> arr=np.array([2,4,6,8,10,12,14,16])
>>> arr[3:7]
array([ 8, 10, 12, 14])
>>> arr[:5]
array([ 2,  4,  6,  8, 10])
>>> arr[4:]
array([10, 12, 14, 16])
>>> arr[:-1]
array([ 2,  4,  6,  8, 10, 12, 14])
>>> arr[:-3]
array([ 2,  4,  6,  8, 10])
>>> arr[2:7:2]
array([ 6, 10, 14])
>>> 
```

Joining or Concatenating Numpy Arrays-

For joining or concatenating of two or more existing ndarrays, python provides following functions-

1. *hstack() and vstack()*
2. *concatenate()*

Combining existing arrays horizontally or vertically-

If you have two 1D arrays as-

1	**4**	**9**	**3**

6	**5**	**7**	**2**

Now, you may want to create a 2D array by stacking these two 1D arrays-

Horizontally as-

1	**4**	**9**	**3**	**6**	**5**	**7**	**2**

Or vertically as-

1	**4**	**9**	**3**
6	5	7	2

You can use the functions hstack() or vstack for this purpose.

Syntax-

Numpy.hstack(<tuple containing names of 1D arrays to be stacked>)

Numpy.vstack(<tuple containing names of 1D arrays to be stacked>)

```
>>> arr[2:7:2]
array([ 6, 10, 14])
>>> list1=[1,2,3,4,5]
>>> list2=[6,7,8,9,10]
>>> a1=np.array(list1)
>>> a1
array([1, 2, 3, 4, 5])
>>> a2=np.array(list2)
>>> a2
array([ 6,  7,  8,  9, 10])
>>> join1=np.hstack(a1,a2)
```

```
>>> join1
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> join2=np.vstack((a1,a2))
>>> join2
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

Similar operations can be applied on 2D arrays.

COMBINING EXISTING ARRAYS USING CONCATENATE()-

Using this function you can concatenate or join NumPy arrays along axis 0(rows) or axis 1(column).

```
numpy.concatenate((<tuple of arrays to be joined>),[axis=<n>])
```

Where-

-Axis argument specifies the axis along which arrays are to be joined. If skipped, axisis assumed as 0 (i.e. along the rows).

The arrays being joined must have the same shape except in the dimension corresponding to argument axis. i.e.-

- If axis is 0, then the shape of the arrays being joined must match on column dimension.
- If axis is 1, then the shape of the arrays being joined must match on rows dimension.

```
>>> a1=np.array([1,2,3,4,5,6,7,8,9]).reshape(3,3)
>>> a1
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a2=np.array([1,2,3,4,5,6]).reshape(2,3)
>>> a2
array([[1, 2, 3],
       [4, 5, 6]])
>>> a3=np.array([1,2,3,4,5,6]).reshape(3,2)
>>> a3
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a4=np.array([1,2,3]).reshape(3,1)
>>> a4
array([[1],
       [2],
       [3]])
...
```

Consider the above mentioned arrays-

a1 with the shape (3,3).

a2 with the shape(2,3)

a3 with the shape(3,2)

a4 with the shape(3,1)

```
>>> j1=np.concatenate((a1,a2),axis=0)
>>> j1
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
       [1, 2, 3],
       [4, 5, 6]])
>>> j2=np.concatenate((a1,a3),axis=1)
>>> j2
array([[1, 2, 3, 1, 2],
       [4, 5, 6, 3, 4],
       [7, 8, 9, 5, 6]])
>>> j3=np.concatenate((a1,a4),axis=1)
>>> j3
array([[1, 2, 3, 1],
       [4, 5, 6, 2],
       [7, 8, 9, 3]])
```

OBTAINING SUBSETS OF ARRAYS-

Subsets van be contiguous as well as non-contiguous.

a. Splitting NumPy Arrays to get contiguous Subsets

NumPy provides some functions namely split(), hpslit(), vsplit() to get the subset from an numpy array. Syntax are-

```
numpy.hsplit(<array>,<n>)

numpy.vsplit(<array>,<n>)
```

Where-

<array> is the NumPy arrayand <n> is the number of sections/subsets in which the array is to be divided.

The <n> must be chosen so that it results in equal division of <array>, otherwise an array will be raised.

```
>>> ary=np.arange(24.0).reshape(4,6)
>>> ary
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.]])
>>> np.hsplit(ary,2)
[array([[ 0.,  1.,  2.],
       [ 6.,  7.,  8.],
       [12., 13., 14.],
       [18., 19., 20.]]), array([[ 3.,  4.,  5.],
       [ 9., 10., 11.],
       [15., 16., 17.],
       [21., 22., 23.]])]

>>> np.vsplit(ary,2)
[array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.]]), array([[12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.]])]
>>> 
```

b. Using the split() function-

numpy.split(<array>,<n>|<1D array>, [axis=0])

- <array> is the Numpy array to split.
- If 2nd arugument is <n>, then <array> is divided in <n> equal subarrays as per axis argument.
- With 2nd argument as <n>, for axis=0, it behaves as vsplit() and for axis=1, it behaves as hsplit().
- If 2nd argument is given as 1D array then <array> is split in unequal subarrays.
- The axis argument is optional and if skipped, it takes the value 0 i.e. on horizontal axis. For axis=1, the split happens on vertical axis.

```
>>> ar1d=[10,11,12,13,14,15,16,17,18,19]
>>> np.split(ar1d,[2,6])
[array([10, 11]), array([12, 13, 14, 15]), array([16, 17, 18, 19])]
>>>
```

0:2, 2:6, 6:

The given argument 2,6 has divided the array into 3 slices i.e. 0:2, 2:6 and 6:

```
>>> ary=np.arange(24.0).reshape(4,6)
>>> ary
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.]])
>>> np.split(ary,[1,4])
[array([[0., 1., 2., 3., 4., 5.]]), array([[ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.]]), array([], shape=(0, 6), dtype=float64)]
>>> np.split(ary,[2,5],axis=1)
[array([[ 0.,  1.],
       [ 6.,  7.],
       [12., 13.],
       [18., 19.]]), array([[ 2.,  3.,  4.],
       [ 8.,  9., 10.],
       [14., 15., 16.],
       [20., 21., 22.]]), array([[ 5.],
       [11.],
       [17.],
       [23.]])]
```

divided as 0:1, 1:4,,,4: horizontally (axis=0 because skipped)

divided as 0:2, 2:5, 5: vertically (axis=1)

Extracting condition based Non-Contiguous Subsets

This is done with the help of extracr() as per the syntax-

numpy.extract (<condition>,<array>)

The extract() always returns the elements of given ndarray that fulfills the criteria of <condition> in 1D ndarray form.

Framing Condition-

To find the subset of a 2D ndarray which is fully divisible by then, then you must write-

condition=no.mod(ary,5)==0

```
>>> ary
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.]])
```

```
>>> cond=np.mod(ary,5)==0
>>> cond
array([[ True, False, False, False, False,  True],
       [False, False, False, False,  True, False],
       [False, False, False,  True, False, False],
       [False, False,  True, False, False, False]])
```

Table 1.2 Various Functions for Ndarrays

Function (x: ndarray)	Description	Example condition
sin(x), cos(x), tan(x) And other trigonometric functions(*e.g.*, arcsin, arccos, arctan etc.)	Trigonometric sine, cosine, tangent etc., element-wise	`cond = np.sin(ary) > 0.25`
around(x)	Evenly round to the given number of decimals.	`cond = np.rint(ary) > 10`
rint(x)	Round elements of the array to the nearest integer.	
fix(x)	Round to nearest integer towards zero	
floor(x)	Return the floor of the input, element-wise.	
ceil(x)	Return the ceiling of the input, element-wise.	
trunc(x) And other mathematical functions	Return the truncated value of the input, element-wise.	
exp(x)	Calculate the exponential of all elements in the input array.	`cond = np.exp2(ary) > 250`
exp2(x)	Calculate 2**p for all p in the input array.	
log(x)	Natural logarithm, element-wise.	
log10(x)	Return the base 10 logarithm of the input array, element-wise.	
log2(x)	Base-2 logarithm of x.	
add(x, <n>)	Add <n> to all elements of array.	`cond = np.mod(ary) < 3`
multiply(x, <n>)	Multiply <n> to all elements of array.	
divide(x, <n>)	Divide <n> to all elements of array.	
power(x, <n>)	Raise elements to the power of <n>.	
subtract(x, <n>)	Subtract <n> to all elements of array.	
mod(x, <n>)	Return element-wise remainder of division.	
remainder(x, <n>)	Return element-wise remainder of division.	
sqrt(x)	Return the positive square-root of an array, element-wise.	
cbrt(x)	Return the cube-root of an array, element-wise.	
square(x)	Return the element-wise square of the input.	
absolute(x])	Calculate the absolute value element-wise.	
fabs(x])	Compute the absolute values element-wise.	

Arithmetic operations on 2D arrays-

a. Using operators-

<ndarray1>+<n>|<ndarray2>

<ndarray1>-<n>|<ndarray2>

<ndarray1>*<n>|<ndarray2>

<ndarray1>/<n>|<ndarray2>

<ndarray1>%<n>|<ndarray2>

b. Using NumPy Functions-

Numpy.add(<ndarray1>,<n>|<ndarray2>)

Numpy.subtract(<ndarray1>,<n>|<ndarray2>)

Numpy.multiply(<ndarray1>,<n>|<ndarray2>)

Numpy.divide(<ndarray1>,<n>|<ndarray2>)

Numpy.mod(<ndarray1>,<n>|<ndarray2>)

Numpy.remainder(<ndarray1>,<n>|<ndarray2>)

```
>>> ary+1.5
array([[ 1.5,  2.5,  3.5,  4.5,  5.5,  6.5],
       [ 7.5,  8.5,  9.5, 10.5, 11.5, 12.5],
       [13.5, 14.5, 15.5, 16.5, 17.5, 18.5],
       [19.5, 20.5, 21.5, 22.5, 23.5, 24.5]])
>>> new=ary+2.1
>>> ary+new
array([[ 2.1,  4.1,  6.1,  8.1, 10.1, 12.1],
       [14.1, 16.1, 18.1, 20.1, 22.1, 24.1],
       [26.1, 28.1, 30.1, 32.1, 34.1, 36.1],
       [38.1, 40.1, 42.1, 44.1, 46.1, 48.1]])

>>> np.multiply(ary,3)
array([[ 0.,  3.,  6.,  9., 12., 15.],
       [18., 21., 24., 27., 30., 33.],
       [36., 39., 42., 45., 48., 51.],
       [54., 57., 60., 63., 66., 69.]])
>>>
```

Applications on NumPy Arrays-

1. Covariance- The intuitive idea behind covariance is that it tells how similar varying two datasets are. A high positive covariance between 2 datasets means they are very strongly Similar. Similarly, a high negative covariance between 2 datasets means they are very dissimilar.

numpy.co(<arr1>m<arr2>)

```
>>> a=np.array([1,2,3,4,5])
>>> b=np.array([3,4,0,-1,-3])
>>> check=np.cov(a,b)
>>> check
array([[ 2.5 , -4.25],
       [-4.25,  8.3 ]])
```

Negative values indicates that they are not very similar

The output is a 2X2 matrix, which is generated as-

Check[0][0]=var(a)

Check[0][1]=covariance(a,b)

Check[1][0]=covariance(b,a)=covariance(a,b)

Check[1][1]=var(b)

2. **Correlation**- Correlation is basically normalized covariance. It gives two values: 1 if the data sets have positive covariance and -1 if the datasets have negative covariance.

np.corrcoef(<array1>, <array2>)

```
>>> corr=np.corrcoef(a,b)
>>> corr
array([[ 1.        , -0.93299621],
       [-0.93299621,  1.        ]])
```