DataRitz Technologies
Enhancing Technology Experience

Topics to be covered

- Re-shaping of array
- Transposing Arrays
- Rotation of array
- Joining of arrays
- Splitting of arrays
- Data Types for ndarrays
- Arithmetic with NumPy Arrays

NumPy

Importing Numpy Package

In [2]:

```
import numpy as np
```

Reshaping of Arrays

- Reshaping means changing the shape of an array.
- For example, if you want to put the numbers 1 through 9 in a 3×3 grid

In [10]:

```
# Reshape the numbers 1 through 9 in a 3×3 grid (Enter valid dimension)
# e.g. for 9 elements dimension should be 3X3 or 1X9 or 9X1
n = np.arange(1,10)
n.reshape((3,3))
```

Out[10]:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Check it Returns a Copy or View?

In [16]:

```
#Check if the returned array is a copy or a view:
n = np.arange(1,10)
n.reshape((3,3)).base
```

Out[16]:

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The example above returns the original array, so it is a view.

Flattening the arrays

> Converting a multidimensional array into a 1D array.

In [20]:

```
x = np.random.randint(10,size=(2,3))
print(x)
x.reshape(-1)
```

```
[[8 8 3]
 [5 2 9]]
```

Out[20]:

```
array([8, 8, 3, 5, 2, 9])
```

Transposing Arrays

Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything

In [90]:

```
x = np.arange(1,10).reshape(3,3)
print(x)
x.T
```

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

Out[90]:

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Rotation of Array

The rot90() function is used to rotate an array by 90 degrees in the plane specified by axes

Syntax -

```
numpy.rot90(m, k=1, axes=(0, 1))
* m    Array of two or more dimensions.
* k    Number of times the array is rotated by 90 degrees
* axes    The array is rotated in the plane defined by the axes.
```

In [3]:

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

Out[3]:

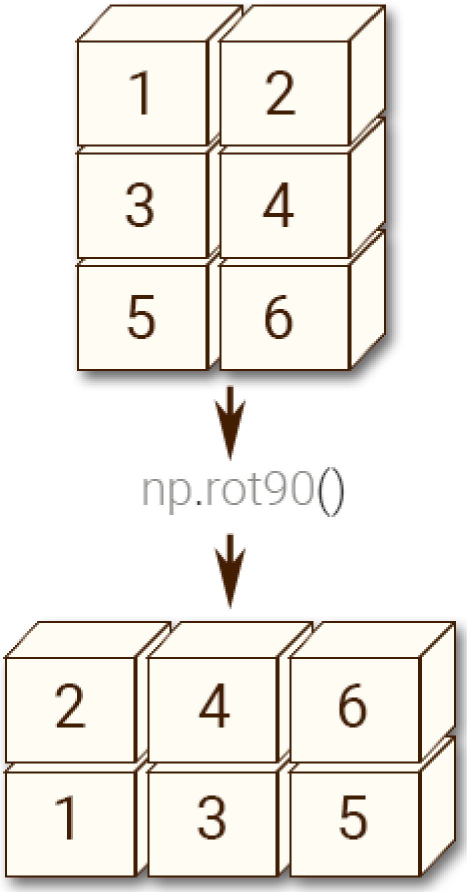
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [4]:

```
np.rot90(m)
```

Out[4]:

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



In [24]:

```
np.rot90(m,2)
```

Out[24]:

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

Joining of Array

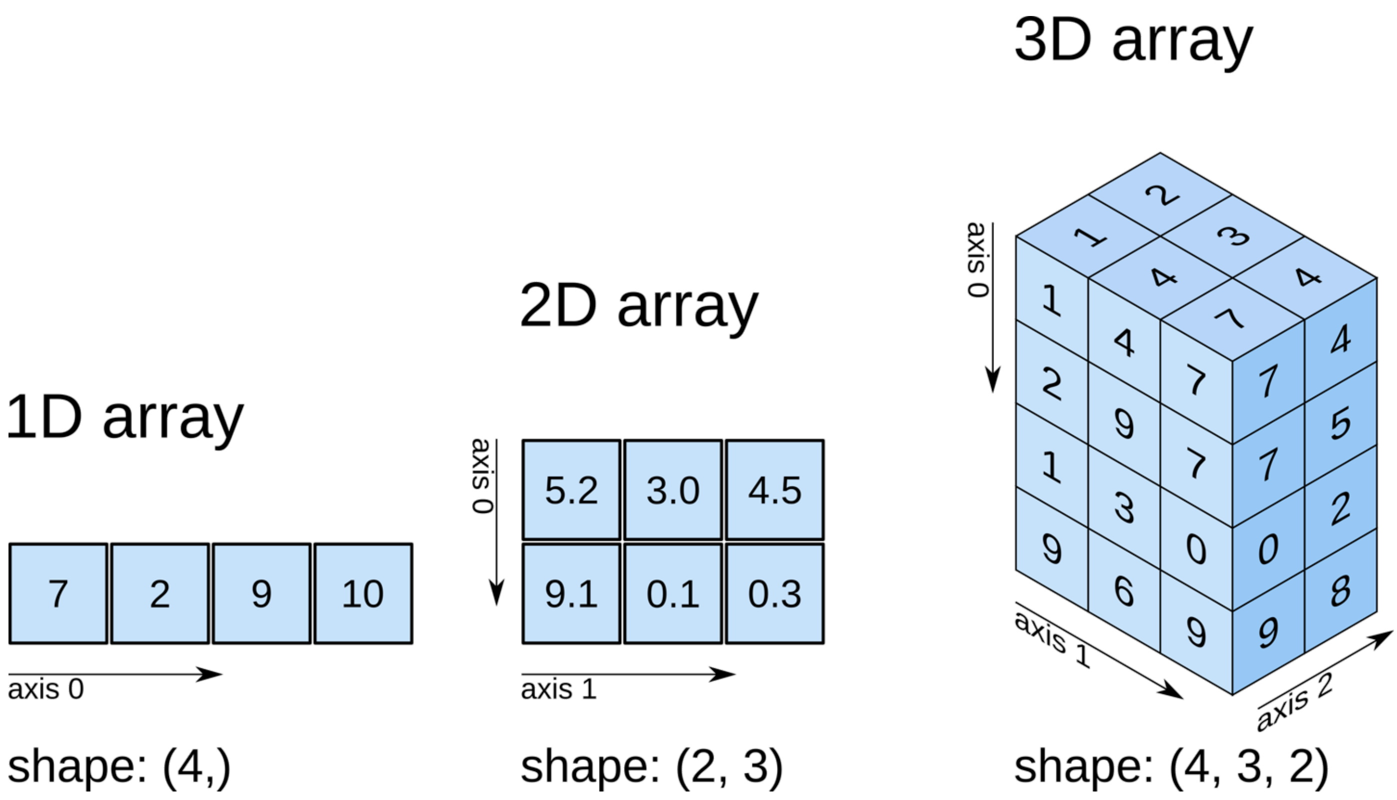
Joining means putting contents of two or more arrays in a single array.

- np.concatenate
- np.vstack
- np.hstack

*** We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis.**

*** If axis is not explicitly passed, it is taken as 0.**

Axis in numpy array



In [6]:

```
# np.concatenate
x = np.array([1,2,3])
y = np.array([4,3,2])
np.concatenate([x,y])
```

Out[6]:

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

Joining 2D arrays

In [32]:

```
# If axis is not explicitly passed, it is taken as 0
x = np.array([[1,1,1],[2,2,2]])
y = np.array([[4,4,4],[5,5,5]])
np.concatenate([x,y])
```

Out[32]:

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

In [33]:

```
# when axis =1
x = np.array([[1,1,1],[2,2,2]])
y = np.array([[4,4,4],[5,5,5]])
np.concatenate([x,y],axis=1)
```

Out[33]:

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

Some Error

In [35]:

```
# when we have different dimension (e.g. 3X3 and 2X3 ) then
x = np.array([[1,1,1],[2,2,2],[3,3,3]])
y = np.array([[4,4,4],[5,5,5]])
np.concatenate([x,y],axis=1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-35-9761ea55f7f1> in <module>
      2 x = np.array([[1,1,1],[2,2,2],[3,3,3]])
      3 y = np.array([[4,4,4],[5,5,5]])
----> 4 np.concatenate([x,y],axis=1)

ValueError: all the input array dimensions except for the concatenation axis
must match exactly
```

In [36]:

```
x = np.array([[1,1,1,1],[2,2,2,2],[3,3,3,3]])
y = np.array([[4,4,4],[5,5,5]])
np.concatenate([x,y],axis=0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-36-f7a0bf7d213e> in <module>
      1 x = np.array([[1,1,1,1],[2,2,2,2],[3,3,3,3]])
      2 y = np.array([[4,4,4],[5,5,5]])
----> 3 np.concatenate([x,y],axis=0)

ValueError: all the input array dimensions except for the concatenation axis
must match exactly
```

np.vstack

allows us to concatenate two multi-dimensional arrays vertically

In [42]:

```
x = np.array([[1,1,1],[2,2,2],[3,3,3]])
y = np.array([[4,4,4],[5,5,5]])
np.vstack([x,y])
```

Out[42]:

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

np.hstack

In [45]:

```
x = np.array([[1,1,1],[2,2,2]])
y = np.array([[4,4,4],[5,5,5]])
np.hstack([x,y])
```

Out[45]:

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

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions

> - np.split
> - np.hsplit
> - np.vsplit.

np.split

The return value of the split() method is an array containing each of the split as an array.

> numpy.split(ary, indices_or_sections, axis=0)

Ref - [https://docs.scipy.org/doc/numpy/reference/generated/numpy.split.html (https://docs.scipy.org/doc/numpy/reference/generated/numpy.split.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.split.html)

In [51]:

```
x = np.arange(1,10)
np.split(x,3)
```

Out[51]:

```
[array([1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]
```

In [54]:

```
# Split [0:3] , [3:5] ,[5:7] and [7:]
x = np.arange(1,10)
np.split(x,[3,5,7])
```

Out[54]:

```
[array([1, 2, 3]), array([4, 5]), array([6, 7]), array([8, 9])]
```

numpy.hsplit

Split an array into multiple sub-arrays horizontally (column-wise).

> numpy.hsplit(ary, indices_or_sections)

Ref - [https://docs.scipy.org/doc/numpy/reference/generated/numpy.hsplit.html#numpy.hsplit (https://docs.scipy.org/doc/numpy/reference/generated/numpy.hsplit.html#numpy.hsplit)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.hsplit.html#numpy.hsplit)

In [58]:

```
x = np.arange(1,10).reshape([3,3])
np.hsplit(x,3)
```

Out[58]:

```
[array([[1],
        [4],
        [7]]), array([[2],
        [5],
        [8]]), array([[3],
        [6],
        [9]])]
```

In [61]:

```
x = np.arange(1,11).reshape([2,5])
print(x)
np.hsplit(x,[2,5])
```

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

Out[61]:

```
[array([[1, 2],
        [6, 7]]), array([[ 3,  4,  5],
        [ 8,  9, 10]]), array([], shape=(2, 0), dtype=int32)]
```

numpy.vsplit

Split an array into multiple sub-arrays vertically (row-wise)

> numpy.vsplit(ary, indices_or_sections)

https://docs.scipy.org/doc/numpy/reference/generated/numpy.vsplit.html#numpy.vsplit (https://docs.scipy.org/doc/numpy/reference/generated/numpy.vsplit.html#numpy.vsplit)

In [68]:

```
x = np.arange(1,11).reshape([5,2])
print(x)
np.vsplit(x,[3])
```

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

Out[68]:

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

Data Types for ndarrays

- Don’t worry about memorizing the NumPy dtypes. It’s often only necessary to care about the general kind of data you’re dealing with, whether floating point, complex, integer, boolean, string, or general Python object.

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

In [71]:

```
x = np.array([1,2,3,4])
x.dtype
```

Out[71]:

```
dtype('int32')
```

In [72]:

```
x =np.array([1.2,2.3,4.3])
x.dtype
```

Out[72]:

```
dtype('float64')
```

In [75]:

```
x = np.array(['1.2','23.3','4.5'],dtype=np.string_)
x.dtype
```

Out[75]:

```
dtype('S6')
```

Casting

In [76]:

```
#In this example, integers were cast to floating point.
x = np.array([1,2,3,4])
y= x.astype(np.float64)
y
```

Out[76]:

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

In [77]:

```
#casting floating-point numbers to be of integer dtype,
#the decimal part will be truncated
x =np.array([1.2,2.3,4.3])

y = x.astype(np.int32)
y
```

Out[77]:

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

In [81]:

```
# you can use astype to convert an array of strings representing numbers
#to numeric form
x = np.array(['1.2','23.3','4.5'],dtype=np.string_)

y = x.astype(np.float)
y
```

Out[81]:

```
array([ 1.2, 23.3,  4.5])
```

Arithmetic with NumPy Arrays

- Arrays are important because they enable you to express batch operations on data without writing any for loops.
- NumPy users call this vectorization
- Any arithmetic operations between equal-size arrays applies the operation element-wise.

In [83]:

```
# Multiplication
x = np.array([[1,2,3],[4,5,6]])
y = np.array([[2,2,2],[3,3,3]])
x*y
```

Out[83]:

```
array([[ 2,  4,  6],
       [12, 15, 18]])
```

In [84]:

```
# Subtraction
x-y
```

Out[84]:

```
array([[-1,  0,  1],
       [ 1,  2,  3]])
```

In [85]:

```
1/x
```

Out[85]:

```
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
```

In [86]:

```
x*2
```

Out[86]:

```
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

In [87]:

```
x**2
```

Out[87]:

```
array([[ 1,  4,  9],
       [16, 25, 36]], dtype=int32)
```

In [88]:

```
# Comparisons between arrays of the same size yield boolean arrays
x>y
```

Out[88]:

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

In [ ]:

```

```