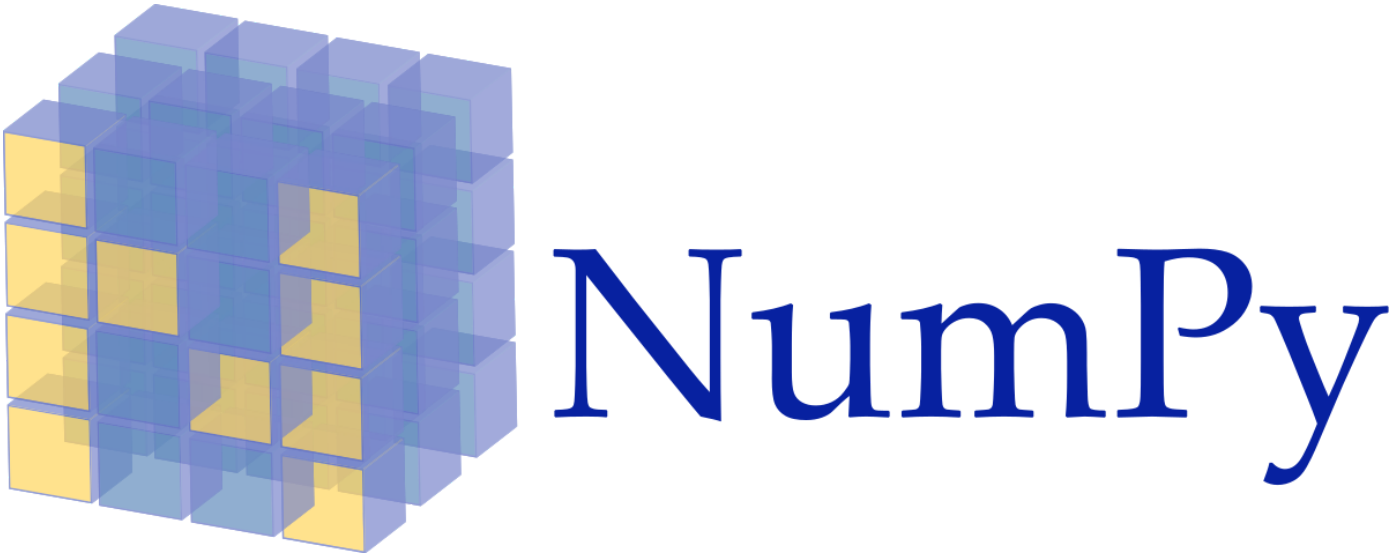


What is NumPy?

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- 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

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

To know more, please visit: https://numpy.org/ (https://numpy.org/)

Why NumPy?

NumPy is an open source numerical Python library. NumPy contains a multi-dimentional array and matrix data structures. It can be utilised to perform a number of mathematical operations on arrays such as trigonometric, statistical and algebraic routines. NumPy is an extension of Numeric and Numarray.

If you totally new in Kaggle, then I would like to recommend you to see this course.
Faster Data Science Education, Link: https://www.kaggle.com/learn/overview (https://www.kaggle.com/learn/overview)

Quickstart tutorial by numpy.org
https://numpy.org/devdocs/user/quickstart.html (https://numpy.org/devdocs/user/quickstart.html)

Import NumPy

In [1]:

```
import numpy as np
```

Stop!!! Let's take a short look on Python List.

**Lists* are just like the arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

Create a list:

In [2]:

```
fruits = ['apple', 'mango', 'orange', 'banana']
print(fruits)
```

```
['apple', 'mango', 'orange', 'banana']
```

The list items can be accessed by referring to the index number. Let's print second item in the list.

In [3]:

```
print(fruits[1])
```

```
mango
```

Let's print last item of the list.

In [4]:

```
print(fruits[-1])
```

```
banana
```

Let's change third item of the list.

In [5]:

```
fruits[2] = 'Pine Apple'
print(fruits)
```

```
['apple', 'mango', 'Pine Apple', 'banana']
```

Check if mango in the list.

In [6]:

```
if 'mango' in fruits:
    print('Yes!')
else:
    print('No!')
```

```
Yes!
```

Find length of a list.

In [7]:

```
len(fruits)
```

Out[7]:

4

Ok! Let's go to NumPy.

Convert a list to Numpy array.

new_array = np.array(list)

In [8]:

```
numbers = [1,2,3,4,5,6,7,8,9,0]
new_numbers = np.array(numbers)
print(new_numbers)
```

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

Or just pass in a list directly.

In [9]:

```
name = np.array(['Rashidul', 'Hasan', 'Hridoy'])
print(name)
```

```
['Rashidul' 'Hasan' 'Hridoy']
```

Pass in a list of lists to create a multidimensional array.

In [10]:

```
a = [1,2,3]
b = [3,2,1]
c = np.array([a,b])
print(c)
```

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

In [11]:

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

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

Use the shape method to find the dimensions of the array. (rows, columns)

In [12]:

```
d.shape
```

Out[12]:

```
(2, 3)
```

In [13]:

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

Out[13]:

```
(3, 3)
```

Let's print diagonal.

In [14]:

```
e
```

Out[14]:

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

In [15]:

```
np.diag(e)
```

Out[15]:

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

Let's do some fun with ones, zeros, eye.

ones returns a new array of given shape and type, filled with ones.

In [16]:

```
np.ones((3,4))
```

Out[16]:

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [17]:

```
np.ones((3,1))
```

Out[17]:

```
array([[1.],
       [1.],
       [1.]])
```

In [18]:

```
np.ones((5,8))
```

Out[18]:

```
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.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])
```

zeros returns a new array of given shape and type, filled with zeros.

In [19]:

```
np.zeros((3,5))
```

Out[19]:

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [20]:

```
np.zeros((4,2))
```

Out[20]:

```
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]])
```

eye returns a 2-D array with ones on the diagonal and zeros elsewhere.

In [21]:

```
np.eye(4)
```

Out[21]:

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [22]:

```
np.eye(6)
```

Out[22]:

```
array([[1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 1.]])
```

Let's play with arange().

It creates an array by using the evenly spaced values over the given interval.

numpy.arange(start, stop, step, dtype)

- start: The starting of an interval. The default is 0.
- stop: represents the value at which the interval ends excluding this value.
- step: The number by which the interval values change.
- dtype: the data type of the numpy array items.

In [23]:

```
np.arange(1,10)
```

Out[23]:

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

In [24]:

```
np.arange(4,14)
```

Out[24]:

```
array([ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13])
```

In [25]:

```
np.arange(1,20,2)
```

Out[25]:

```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

In [26]:

```
np.arange(0,20,2)
```

Out[26]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [27]:

```
np.arange(0,40,5)
```

Out[27]:

```
array([ 0,  5, 10, 15, 20, 25, 30, 35])
```

In [28]:

```
np.arange(0,100,3,float)
```

Out[28]:

```
array([ 0.,  3.,  6.,  9., 12., 15., 18., 21., 24., 27., 30., 33., 3
6.,
       39., 42., 45., 48., 51., 54., 57., 60., 63., 66., 69., 72., 7
5.,
       78., 81., 84., 87., 90., 93., 96., 99.])
```

In [29]:

```
np.arange(1,20, 0.5, float)
```

Out[29]:

```
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6.
,
        6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. , 10.5, 11. , 11.
5,
       12. , 12.5, 13. , 13.5, 14. , 14.5, 15. , 15.5, 16. , 16.5, 17.
,
       17.5, 18. , 18.5, 19. , 19.5])
```

Let's play with numpy.reshape().

In some occasions, you need to reshape the data from wide to long. You can use the reshape function for this. The syntax is

numpy.reshape(a, newShape, order='C')

Here,

a: Array that you want to reshape

newShape: The new desires shape

Order: Default is C which is an essential row style.

In [30]:

```
x = np.arange(1,11)
x
```

Out[30]:

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

In [31]:

```
x.reshape(2,5)
```

Out[31]:

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

In [32]:

```
x.reshape(5,2)
```

Out[32]:

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

In [33]:

```
x.reshape(10,1)
```

Out[33]:

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

In [34]:

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

Out[34]:

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

In [35]:

```
y.reshape(4,2)
```

Out[35]:

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

In [36]:

```
z = np.array([1,2,3,4,5,6,7,8,9,0])
z.reshape(2,5)
```

Out[36]:

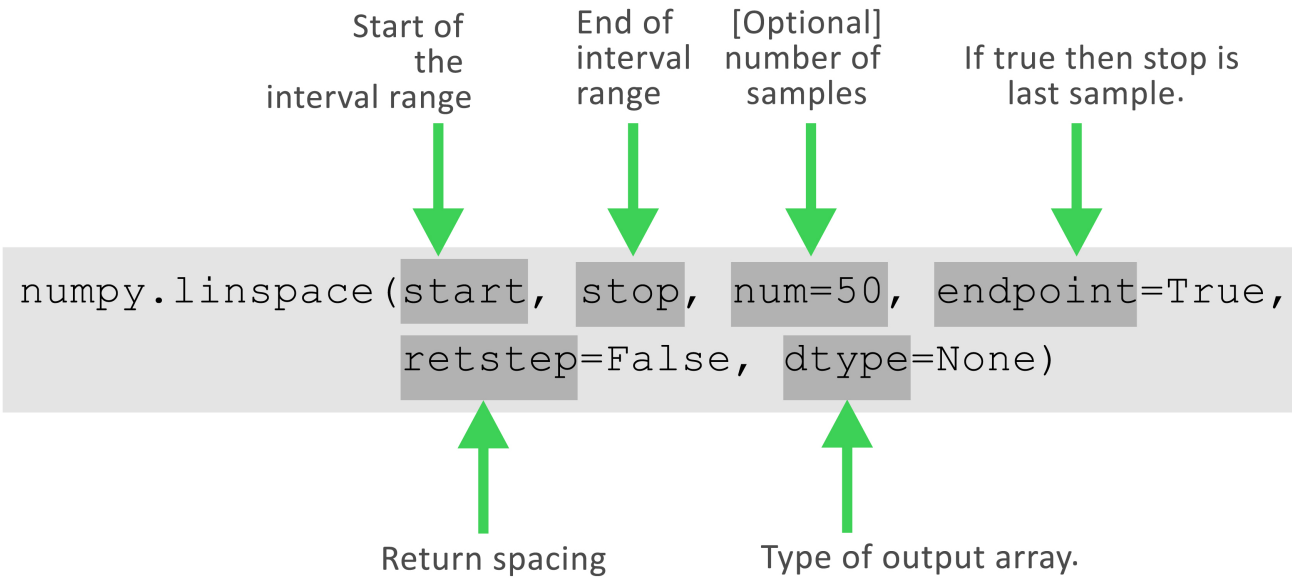
```
array([[1, 2, 3, 4, 5],
       [6, 7, 8, 9, 0]])
```

Let's play with numpy.linspace().

The numpy.linspace() function in Python returns evenly spaced numbers over the specified interval. This function is similar to The Numpy arange function but it uses the number instead of the step as an interval.

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

- start Specify the start of the interval range. The default value is 0.
- stop provide end of the interval range.
- num [Optional] the number of samples to generate. The default is 50.
- endpoint [Optional] if true then stop is the last sample.
- retstep [Optional] if the value is true then return the spacing between samples
- dtype [Optional] dtype is the type of output array.



In [37]:

```
np.linspace(0,4,9)
```

Out[37]:

```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ])
```

In [38]:

```
x = np.linspace(0,10,20)
x
```

Out[38]:

```
array([ 0.        ,  0.52631579,  1.05263158,  1.57894737,  2.1052631
6,
        2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.7368421
1,
        5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.3684210
5,
        7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.
])
```

In [39]:

```
np.linspace(10,100,10)
```

Out[39]:

```
array([ 10.,  20.,  30.,  40.,  50.,  60.,  70.,  80.,  90., 100.])
```

Let's play with numpy.resize().

With the help of Numpy numpy.resize(), we can resize the size of an array. Array can be of any shape but to resize it we just need the size i.e (2, 2), (2, 3) and many more. During resizing numpy append zeros if values at a particular place is missing.

Most of you are now thinking that what is the difference between reshape and resize. When we talk about reshape then an array changes it's shape as temporary but when we talk about resize then the changes made permanently.

In [40]:

```
x = np.arange(0,20)
x
```

Out[40]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16,
       17, 18, 19])
```

In [41]:

```
x.reshape(5,4)
```

Out[41]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

In [42]:

```
x
```

Out[42]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16,
       17, 18, 19])
```

In [43]:

```
x.resize(5,4)
```

In [44]:

```
x
```

Out[44]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

In [45]:

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

Out[45]:

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

In [46]:

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

Out[46]:

```
array([[1, 2, 4],
       [5, 4, 5],
       [6, 3, 0]])
```

Let's create an array using repeating list.

In [47]:

```
np.array([1, 2, 3] * 3)
```

Out[47]:

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

Repeat elements of an array using repeat.

In [48]:

```
np.repeat([1, 2, 3], 3)
```

Out[48]:

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

Let's play with operations.

Use +, -, *, / and to perform element wise addition, subtraction, multiplication, division and power.

In [49]:

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

In [50]:

```
x+y
# elementwise addition     [1 2 3] + [4 5 6] = [5  7  9]
```

Out[50]:

```
array([5, 7, 9])
```

In [51]:

```
x-y
# elementwise subtraction  [1 2 3] - [4 5 6] = [-3 -3 -3]
```

Out[51]:

```
array([-3, -3, -3])
```

In [52]:

```
x*y
# elementwise multiplication  [1 2 3] * [4 5 6] = [4  10  18]
```

Out[52]:

```
array([ 4, 10, 18])
```

In [53]:

```
x/y
# elementwise divison         [1 2 3] / [4 5 6] = [0.25  0.4  0.5]
```

Out[53]:

```
array([0.25, 0.4 , 0.5 ])
```

In [54]:

```
x**2
# elementwise power  [1 2 3] ^2 =  [1 4 9]
```

Out[54]:

```
array([1, 4, 9])
```

Dot Product:

In [55]:

```
x.dot(y)
# dot product  1*4 + 2*5 + 3*6
```

Out[55]:

```
32
```

In [56]:

```
y.dot(x)
```

Out[56]:

```
32
```

Let's look at transposing arrays. Transposing permutes the dimensions of the array.

In [57]:

```
a = np.array([x, x**2])
a
```

Out[57]:

```
array([[1, 2, 3],
       [1, 4, 9]])
```

In [58]:

```
a.shape
```

Out[58]:

```
(2, 3)
```

Use .T to get the transpose.

In [59]:

```
a.T
```

Out[59]:

```
array([[1, 1],
       [2, 4],
       [3, 9]])
```

In [60]:

```
a.T.shape
```

Out[60]:

```
(3, 2)
```

Let's check data type.

Use .dtype to see the data type of the elements in the array.

In [61]:

```
x.dtype
```

Out[61]:

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

Use .astype to cast to a specific type.

In [62]:

```
x = x.astype('f')
x.dtype
```

Out[62]:

```
dtype('float32')
```

Math Functions

Numpy has many built in math functions that can be performed on arrays.

In [63]:

```
a = np.array([1,4,5,6,-5,3,7])
```

In [64]:

```
a.sum()
```

Out[64]:

```
21
```

In [65]:

```
a.max()
```

Out[65]:

```
7
```

In [66]:

```
a.min()
```

Out[66]:

```
-5
```

In [67]:

```
a.mean()
```

Out[67]:

```
3.0
```

In [68]:

```
a.std()
```

Out[68]:

```
3.7416573867739413
```

argmax and argmin return the index of the maximum and minimum values in the array.

In [69]:

```
a.argmax()
```

Out[69]:

```
6
```

In [70]:

```
a.argmin()
```

Out[70]:

```
4
```

Let's learn indexing or slicing.

In [71]:

```
b = np.arange(14)**2
b
```

Out[71]:

```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 14
4,
       169])
```

Use bracket notation to get the value at a specific index. Remember that indexing starts at 0.

In [72]:

```
b[0], b[3], b[-1], b[-5]
```

Out[72]:

```
(0, 9, 169, 81)
```

Use : to indicate a range. array[start:stop]

Leaving start or stop empty will default to the beginning/end of the array.

In [73]:

```
a[1:7]
```

Out[73]:

```
array([ 4,  5,  6, -5,  3,  7])
```

In [74]:

```
a[3:4]
```

Out[74]:

```
array([6])
```

Use negatives to count from the back.

In [75]:

```
a[-2:]
```

Out[75]:

```
array([3, 7])
```

In [76]:

```
a[-5:]
```

Out[76]:

```
array([ 5,  6, -5,  3,  7])
```

A second : can be used to indicate step-size. array[start:stop:stepsize]

Here we are starting 5th element from the end, and counting backwards by 2 until the beginning of the array is reached.

In [77]:

```
a[-1::-3]
```

Out[77]:

```
array([7, 6, 1])
```

In [78]:

```
a[-1::-5]
```

Out[78]:

```
array([7, 4])
```

Let's look at a multidimensional array.

In [79]:

```
r = np.arange(36)
r.resize((6, 6))
r
```

Out[79]:

```
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],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

Use bracket notation to slice: array[row, column]

In [80]:

```
r[2,2]
```

Out[80]:

```
14
```

In [81]:

```
r[5,5]
```

Out[81]:

```
35
```

In [82]:

```
r[5,2]
```

Out[82]:

```
32
```

And use : to select a range of rows or columns

In [83]:

```
r[3, 3:6]
```

Out[83]:

```
array([21, 22, 23])
```

Here we are selecting all the rows up to (and not including) row 2, and all the columns up to (and not including) the last column.

In [84]:

```
r[:2, :-1]
```

Out[84]:

```
array([[ 0,  1,  2,  3,  4],
       [ 6,  7,  8,  9, 10]])
```

This is a slice of the last row, and only every other element.

In [85]:

```
r[-1, ::2]
```

Out[85]:

```
array([30, 32, 34])
```

We can also perform conditional indexing. Here we are selecting values from the array that are greater than 30. (Also see np.where)

In [86]:

```
r[r > 30]
```

Out[86]:

```
array([31, 32, 33, 34, 35])
```

Here we are assigning all values in the array that are greater than 30 to the value of 30.

In [87]:

```
r[r > 30] = 30
r
```

Out[87]:

```
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],
       [24, 25, 26, 27, 28, 29],
       [30, 30, 30, 30, 30, 30]])
```

Copying Data

Be careful with copying and modifying arrays in NumPy!

r2 is a slice of r

In [88]:

```
r2 = r[:3,:3]
r2
```

Out[88]:

```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [12, 13, 14]])
```

Set this slice's values to zero ([:] selects the entire array)

In [89]:

```
r2[:] = 0
r2
```

Out[89]:

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

r has also been changed!

In [90]:

```
r
```

Out[90]:

```
array([[ 0,  0,  0,  3,  4,  5],
       [ 0,  0,  0,  9, 10, 11],
       [ 0,  0,  0, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 30, 30, 30, 30, 30]])
```

To avoid this, use r.copy to create a copy that will not affect the original array.

In [91]:

```
r_copy = r.copy()
r_copy
```

Out[91]:

```
array([[ 0,  0,  0,  3,  4,  5],
       [ 0,  0,  0,  9, 10, 11],
       [ 0,  0,  0, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 30, 30, 30, 30, 30]])
```

Now when r_copy is modified, r will not be changed.

In [92]:

```
r_copy[:] = 10
print(r_copy, '\n')
print(r)
```

```
[[10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]]

[[ 0  0  0  3  4  5]
 [ 0  0  0  9 10 11]
 [ 0  0  0 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 30 30 30 30 30]]
```

Let's Iterate Over Arrays

Let's create a new 4 by 3 array of random numbers 0-9.

In [93]:

```
t = np.random.randint(0, 10, (4,3))
t
```

Out[93]:

```
array([[9, 1, 2],
       [2, 6, 1],
       [8, 0, 0],
       [9, 9, 4]])
```

Iterate by row.

In [94]:

```
for row in t:
    print(row)
```

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

Iterate by index.

In [95]:

```
for i in range(len(t)):
    print(t[i])
```

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

Iterate by row and index.

In [96]:

```
for i, row in enumerate(t):
    print('row', i, 'is', row)
```

```
row 0 is [9 1 2]
row 1 is [2 6 1]
row 2 is [8 0 0]
row 3 is [9 9 4]
```

Use zip to iterate over multiple iterables.

In [97]:

```
t2 = t**2
t2
```

Out[97]:

```
array([[81,  1,  4],
       [ 4, 36,  1],
       [64,  0,  0],
       [81, 81, 16]])
```

In [98]:

```
for i, j in zip(t, t2):
    print(i,'+',j,'=',i+j)
```

```
[9 1 2] + [81  1  4] = [90  2  6]
[2 6 1] + [ 4 36  1] = [ 6 42  2]
[8 0 0] + [64  0  0] = [72  0  0]
[9 9 4] + [81 81 16] = [90 90 20]
```

Ok, stop! To be continued..!

Learn Python Pandas using Pokemon Dataset.

Click on the link below.

Learn Python Pandas using Pokemon Dataset (https://www.kaggle.com/rhhridoy/learn-python-pandas-using-pokemon-dataset)

This Python cheat sheet is a quick reference for NumPy beginners.

by **DataCamp**

Given the fact that it's one of the fundamental packages for scientific computing, NumPy is one of the packages that you must be able to use and know if you want to do data science with Python. It offers a great alternative to Python lists, as NumPy arrays are more compact, allow faster access in reading and writing items, and are more convenient and more efficient overall.

In addition, it's (partly) the fundament of other important packages that are used for data manipulation and machine learning which you might already know, namely, Pandas, Scikit-Learn and SciPy:

- The Pandas data manipulation library builds on NumPy, but instead of the arrays, it makes use of two other fundamental data structures: Series and DataFrames,
- SciPy builds on Numpy to provide a large number of functions that operate on NumPy arrays, and
- The machine learning library Scikit-Learn builds not only on NumPy, but also on SciPy and Matplotlib.

You see, this Python library is a must-know: if you know how to work with it, you'll also gain a better understanding of the other Python data science tools that you'll undoubtedly be using.

It's a win-win situation, right?

Nevertheless, just like any other library, NumPy can come off as quite overwhelming at start; What are the very basics that you need to know in order to get started with this data analysis library?

This cheat sheet means to give you a good overview of the possibilities that this library has to offer.

Go and check it out for yourself!

Click on the link below.

NumPy Cheat Sheet: Data Analysis in Python (https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf)

Thanks <3 .