Data science and Machine learning with Python

Designed by Abdur Rahman Joy - MCSD, MCPD, MCSE, MCTS, OCJP, Sr. Technical Trainer for VFX at IDB BISW (Scholarship program), and C#.net, R, Scala, Kotlin, JAVA, Android/IOS/Windows Mobile Apps, SQL server, Azure, Oracle, SharePoint Development, AWS , CEH, KALI Linux, Python, Data Science, Machine Learning ,Software Testing, Graphics, Multimedia and Game Developer at Joy Infosys and other premises like BITM, SkillsJob, PNTL, Leads Training and New Horizon inc , Cell #: +880-1712587348, email: jspaonline@gmail.com. Web URL: http://www.joyinfosys.com/me.

Python for Data Analysis – NumPy

NumPy is a commonly used Python data analysis package. By using NumPy, you can speed up your workflow, and interface with other packages in the Python ecosystem. NumPy was originally developed in the mid 2000s, and arose from an even older package called Numeric. This longevity means that almost every data analysis or machine learning package for Python leverages NumPy in some way.

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.

The pandas library has emerged into a power house of data manipulation tasks in python since it was developed in 2008. With its intuitive syntax and flexible data structure, it's easy to learn and enables faster data computation. The development of numpy and pandas libraries has extended python's multi-purpose nature to solve machine learning problems as well. The acceptance of python language in machine learning has been phenomenal since then.

This is just one more reason underlining the need for you to learn these libraries now. Published in early 2017, this blog claimed that python jobs outnumbered R jobs.

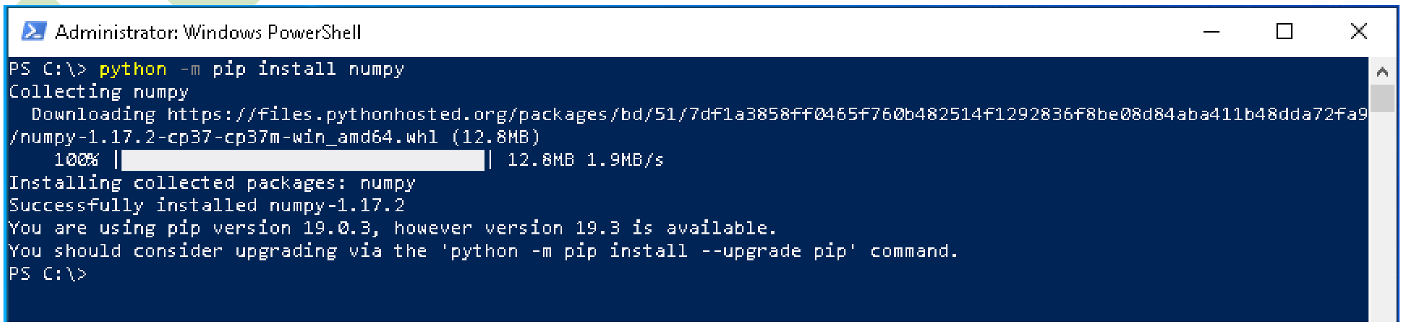
In this tutorial, we'll learn about using numpy and pandas libraries for data manipulation from scratch. Instead of going into theory, we'll take a practical approach.

First, we'll understand the syntax and commonly used functions of the respective libraries. Later, we'll work on a real-life data set.

Install NumPy:

Now, open a *cmd* window like before. Use the next set of commands to install *NumPy*,

python -m pip install numpy



```
PS C:\> python -m pip install numpy
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/bd/51/7df1a3858ff0465f760b482514f1292836f8be08d84aba411b48dda72fa9
/numpy-1.17.2-cp37-cp37m-win_amd64.whl (12.8MB)
    100% |████████████████████████████████| 12.8MB 1.9MB/s
Installing collected packages: numpy
Successfully installed numpy-1.17.2
You are using pip version 19.0.3, however version 19.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
PS C:\>
```

Start with NumPy:

Creating Arrays

Numpy arrays are homogeneous in nature, i.e., they comprise one data type (integer, float, double, etc.) unlike lists.

#creating arrays

```
>>> np.zeros(10, dtype='int')

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

#creating a 3 row x 5 column matrix

```
>>> np.ones((3,5), dtype=float)

array([[1., 1., 1., 1., 1.],

       [1., 1., 1., 1., 1.],

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

#creating a matrix with a predefined value

```
>>> np.full((3,5),1.23)

array([[1.23, 1.23, 1.23, 1.23, 1.23],

       [1.23, 1.23, 1.23, 1.23, 1.23],

       [1.23, 1.23, 1.23, 1.23, 1.23]])
```

#create an array with a set sequence

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

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

#create an array of even space between the given range of values

```
>>> np.linspace(0, 1, 5)

array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

#create a 3x3 array with mean 0 and standard deviation 1 in a given dimension

```
>>> np.random.normal(0, 1, (3,3))

array([[-2.0589693 , -2.06847621,  0.42492852],

       [ 0.4764281 ,  1.42815186,  0.32102325],

       [-0.41534891,  0.57511683,  0.1321772 ]])
```

#create an identity matrix

```
>>> np.eye(3)

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

       [0., 1., 0.],

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

Array Indexing

The important thing to remember is that indexing in python starts at zero.

```
>>> x1 = np.array([4, 3, 4, 4, 8, 4])

>>> x1

array([4, 3, 4, 4, 8, 4])
```

#assess value to index zero

```
>>> x1[0]

4
```

#assess fifth value

```
>>> x1[4]

8
```

#get the last value

```
>>> x1[-1]

4
```

#get the second last value

```
>>> x1[-2]

8
```

#in a multidimensional array, we need to specify row and column index

```
>>> x2 = np.random.randint(10, size=(3,4))

>>> x2

array([[5, 0, 3, 3],

       [7, 9, 3, 5],

       [2, 4, 7, 6]])
```

#1st row and 2nd column value

```
>>> x2[2,3]

6
```

#3rd row and last value from the 3rd column

```
>>> x2[2,-1]

6
```

#replace value at 0,0 index

```
>>> x2[0,0] = 12

>>> x2

array([[12,  0,  3,  3],

       [ 7,  9,  3,  5],

       [ 2,  4,  7,  6]])
```

Array Slicing

Now, we'll learn to access multiple or a range of elements from an array.

```
>>> x = np.arange(10)

>>> x

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

#from start to 4th position

```
>>> x[:5]

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

#from 4th position to end

```
>>> x[4:]

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

#from 4th to 6th position

```
>>> x[4:7]

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

#return elements at even place

```
>>> x[ : : 2]

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

#return elements from first position step by two

```
>>> x[1::2]

array([1, 3, 5, 7, 9])
```

#reverse the array

```
>>> x[::-1]

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

Array Concatenation

Many a time, we are required to combine different arrays. So, instead of typing each of their elements manually, you can use array concatenation to handle such tasks easily.

#You can concatenate two or more arrays at once.

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

>>> y = np.array([3, 2, 1])

>>> z = [21,21,21]

>>> np.concatenate([x, y,z])

array([ 1,  2,  3,  3,  2,  1, 21, 21, 21])
```

#You can also use this function to create 2-dimensional arrays.

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

>>> np.concatenate([grid,grid])

array([[1, 2, 3],

       [4, 5, 6],

       [1, 2, 3],

       [4, 5, 6]])
```

#Using its axis parameter, you can define row-wise or column-wise matrix

```
>>> np.concatenate([grid,grid],axis=1)

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

       [4, 5, 6, 4, 5, 6]])
```

Until now, we used the concatenation function of arrays of equal dimension. But, what if you are required to combine a 2D array with 1D array? In such situations, np.concatenate might not be the best option to use. Instead, you can use np.vstack or np.hstack to do the task. Let's see how!

```
>>> x = np.array([3,4,5])

>>> grid = np.array([[1,2,3],[17,18,19]])

>>> np.vstack([x,grid])

array([[ 3,  4,  5],

       [ 1,  2,  3],

       [17, 18, 19]])
```

#Similarly, you can add an array using np.hstack

```
>>> z = np.array([[9],[9]])

>>> np.hstack([grid,z])

array([[ 1,  2,  3,  9],

       [17, 18, 19,  9]])
```

Also, we can split the arrays based on pre-defined positions. Let's see how!

```
>>> x = np.arange(10)

>>> x

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

>>> x1,x2,x3 = np.split(x,[3,6])

>>> print (x1,x2,x3)

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

>>> grid = np.arange(16).reshape((4,4))

>>> grid

array([[ 0,  1,  2,  3],

       [ 4,  5,  6,  7],

       [ 8,  9, 10, 11],

       [12, 13, 14, 15]])

>>> upper,lower = np.vsplit(grid,[2])

>>> print (upper, lower)
```

```
[[0 1 2 3]

 [4 5 6 7]] [[ 8  9 10 11]

 [12 13 14 15]]
```

In addition to the functions we learned above, there are several other mathematical functions available in the numpy library such as sum, divide, multiple, abs, power, mod, sin, cos, tan, log, var, min, mean, max, etc. which you can be used to perform basic arithmetic calculations. Feel free to refer to numpy documentation for more information on such functions.

np.arange( ) is the best way to **create large matrices with n-dimensional.** By default, np.arange( ) created one dimensional array, if you want to create 2-Dimensional or 3-Dimensional matrices, you can use **np.reshape ( ) with np.arange** .

np.arange(start=None, stop=None, step=None, dtype=None)

np.arange ( ) has four parameters:

1. **start — starting the array from the start number.**
2. **stop — end the array (excluded in stop value)**
3. **step — jump the value**
4. **dtype — the type of array or matrices**

```
>>> np.arange(0, 10, 3)

array([0, 3, 6, 9])
```

===============================
start = 0
stop = 10
step = 3

Create an evenly spaced array between 1 and 60 with a difference of 2.

```
>>> dif=np.arange(1,60,2)

>>> dif

array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33,

       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59])
```

Reshape the above array into a desired shape.

```
>>> dif.reshape(10,3)

array([[ 1,  3,  5],

       [ 7,  9, 11],

       [13, 15, 17],

       [19, 21, 23],

       [25, 27, 29],

       [31, 33, 35],

       [37, 39, 41],

       [43, 45, 47],

       [49, 51, 53],

       [55, 57, 59]])
```

Generate an evenly spaced list between the interval 1 and 8. (Take a minute here to understand the difference between 'linspace' and 'arange')

```
>>> gen = np.linspace(1,8,40)

>>> gen

array([1.        , 1.17948718, 1.35897436, 1.53846154, 1.71794872,

       1.8974359 , 2.07692308, 2.25641026, 2.43589744, 2.61538462,

       2.79487179, 2.97435897, 3.15384615, 3.33333333, 3.51282051,

       3.69230769, 3.87179487, 4.05128205, 4.23076923, 4.41025641,

       4.58974359, 4.76923077, 4.94871795, 5.12820513, 5.30769231,

       5.48717949, 5.66666667, 5.84615385, 6.02564103, 6.20512821,

       6.38461538, 6.56410256, 6.74358974, 6.92307692, 7.1025641 ,

       7.28205128, 7.46153846, 7.64102564, 7.82051282, 8.        ])
```

Now, change the shape of the array in place (‘resize’ function changes the shape of the array in place, unlike ‘reshape’)

```
>>> gen.resize(10,4)

>>> gen

array([[1.        , 1.17948718, 1.35897436, 1.53846154],
       [1.71794872, 1.8974359 , 2.07692308, 2.25641026],
       [2.43589744, 2.61538462, 2.79487179, 2.97435897],
       [3.15384615, 3.33333333, 3.51282051, 3.69230769],
       [3.87179487, 4.05128205, 4.23076923, 4.41025641],
       [4.58974359, 4.76923077, 4.94871795, 5.12820513],
       [5.30769231, 5.48717949, 5.66666667, 5.84615385],
       [6.02564103, 6.20512821, 6.38461538, 6.56410256],
       [6.74358974, 6.92307692, 7.1025641 , 7.28205128],
       [7.46153846, 7.64102564, 7.82051282, 8.        ]])
```