**Numpy –Datacamp**

● Numeric Python

● Alternative to Python List: NumPy Array

● Calculations over entire arrays

● Easy and Fast

**Methods:** Import numpy as np

np.corrcoef(df[:,1],df[:,2])

np.mean(np\_city[:,0])

np.median(np\_city[:,0])

np.std(np\_city[:,0])

**Sort(): sort(a, axis=-1, kind='quicksort', order=None)**

dtype = [('name', 'S10'), ('height', float), ('age', int)]

values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38), ('Galahad', 1.7, 38)]

a = np.array(values, dtype=dtype) # create a structured array

np.sort(a, order='height')

**Sort by age, then height if ages are equal:**

np.sort(a, order=['age', 'height']) # doctest: +SKIP

array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38), ('Arthur', 1.8, 41)],

dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])

**column\_stack() :** **column\_stack(tup)**

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

Examples

a = np.array((1,2,3))

b = np.array((2,3,4))

np.column\_stack((a,b))🡺 array([[1, 2],

[2, 3],

[3, 4]])

🡺**(np.round(np.random.normal(1.75, 0.20, 5), 3))**

array([1.635, 1.964, 2.049, 1.474, 1.773])

**Stack():**Join a sequence of arrays along a new 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.

**Concatenate()** : Join a sequence of arrays along an existing axis.

**split ():** Split array into a list of multiple sub-arrays of equal size.

**block ():** Assemble arrays from blocks.

**\*\*TO iterate all items in 2D array using for loop:**

For val in np.nditer(array([[],[]])):

Print(val)

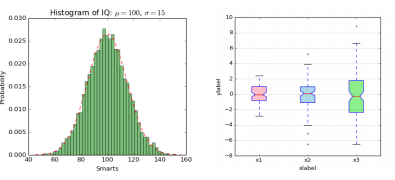
**Pandas**

TO add columns of one columns length use apply() method

i.e

df[‘new\_column’]=df[‘old\_column’].apply(len)

**Matplotlib**

**Data Visualization**

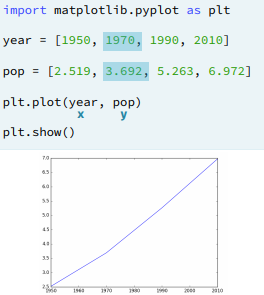
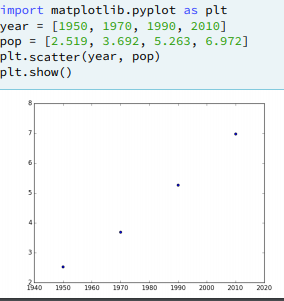
● Very important in Data Analysis

● Explore data

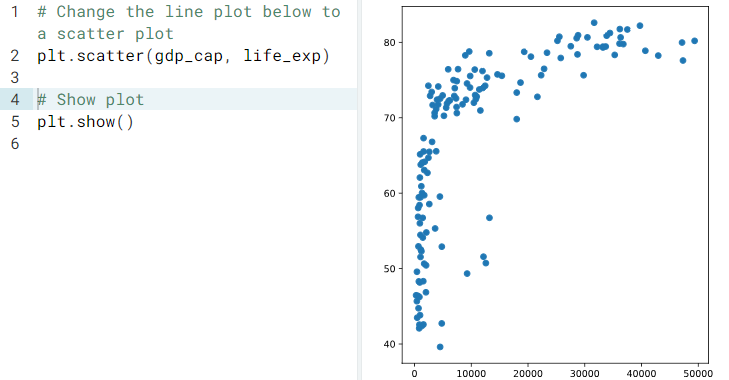
● Report insights

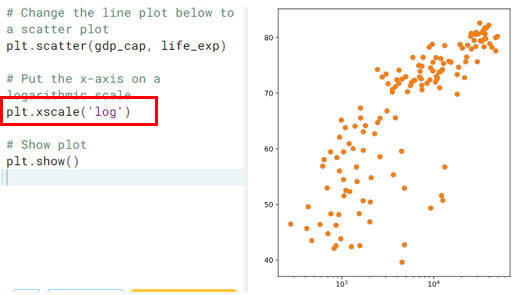
**Methods:**

Plot(): Scatter()

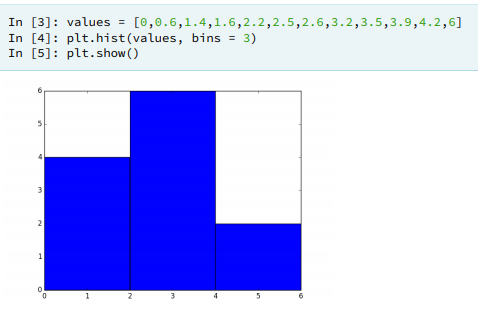


**\*\*Change the axis scale to log:**

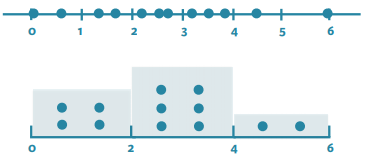




**Histogram:**

● Explore dataset

● Get idea about distribution

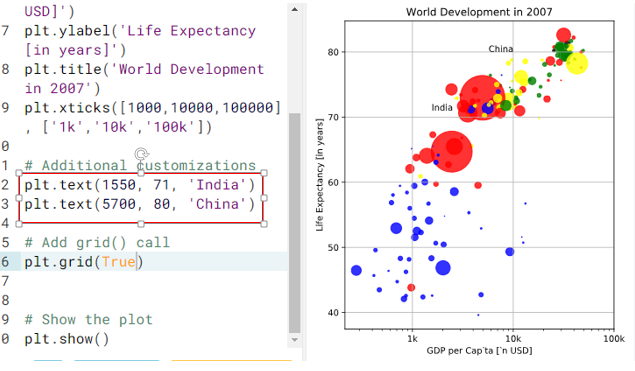


**Import matplotplot.pyplot as plt**

plt.hist([1,2,3,4,5,56],bins=15,orientation=’’vertical/horizontal)

orientation: Define the position of plot

**Adding text in plot:**



**Iterators and Iterables:**

**Iterables:** Which are associated with next() method.

Iter() method is used to create iterator object, by which a next() can be used.

Eg: list,string,dictonories,file connections

**Iterators:**

Produces the next values by using next()

Eg:

L=[1,2,4]

Print(next(L))🡺Throws error

L=iter(L)

Print(next(L))🡺 1, Print(next(L))🡺 2, Print(next(L))🡺 4, Print(next(L))🡺 Throws error as no values is there to print

**TO Print all values in iterator object, use \***

**EG: s=’koushik’**

**S=iter(s)**

**Print(\*S)🡺 k o u s h I k**

**Note🡺** Iterator does not create a list , instead it creates a object which goes till the last value even though the iterator range crosses the computer memory range

Eg: range(10\*\*100) is very large size , in creating list to this range throws memory error but if we create a iter() , it will reach to the end

**List Comprehension:**

[[output expression] for iterator variable in iterable]

[[i\*j **for** j in range(1,11)] **for** i in range(7,9)]

**Output**:

* 7\*1=
* 7\*2=
* 8\*1=
* 8\*2=

**It is same as**

for j in range(7,9):

for i in range(1,11):

i\*j

**Generators:**

Generators are used when we want to itterate very large range. In normal for loop we can’t iterate very large range,so we use generators.

* Generators wont store the values at the time of creation. Just an object is created and calls the each value by calling next() method

**Syntax:**

**Genrator comprehesion:gen= ( i for i in range(1000000000000))**

**Genrator function: def demo():**

**for i in range(100000000):**

**yield i**

**To call the value:** next(gen) and

Dem=demo()

next(Dem)

**Reading very very large files:**

Reading very very large files causes the memory exception, to over come this problem,

Use chunks concepts to read very cery large files.

In pandas.read\_csv(), we have parameter: **chunksize** whichdefines the size of chunk

* Chunk means dividing the large file into pieces and iterating on each piece
* Large chunk size helps in reducing the oversize problems
* In pandas.read\_csv(file,chucksize=1000),each chuck defines the each data frame.

Note🡺 To write csv file from chuck use **mode=’a’** to append tall the chunks, if not only last chuck data will be created