# Numpy

- NumPy is a Python package.
- NumPy stands for 'Numerical Python'.
- It is core library for scientific computing.
- It provides a high-performance multidimensional array object, and tools for working with these arrays
- Basic operations using NumPy:
  - Mathematical and logical operations on arrays.
  - Operations related to linear algebra.
  - Random number generation.
  - Fourier transforms and shape manipulation etc.

# Importing the NumPy library

```
>>>import numpy
```

- For large amounts of calls to NumPy functions, it can become tedious to write numpy.Y over and over again.
- A common practice is to import numpy under the brief name np

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

# Array

- The central feature of NumPy is the ndarray object class.
- Arrays are similar to lists in Python, except that every element of an array must be of the same type, typically a numeric type.
- With large amounts of numeric data, arrays make operations very fast and efficient as compare to lists.

```
>>>import numpy as np
>>>arr = np.array([1,2,3,4,5], float)
>>>arr
array([1. 2. 3. 4. 5.])
>>>type(arr)
<class 'numpy.ndarray'>
```

- In np.array() the second argument is optional represents the desired data-type for the array. If not given, then the data type will be determined as the minimum data type required to hold the objects in the sequence.

# Array

- Array elements are accessed, sliced, and manipulated just like lists:

```
>>>import numpy as np
>>>arr = np.array([1,2,3,4,5])
>>>print(arr[:3])
[1 2 3]
>>>arr[3]
4
>>>arr[-1]
5
>>>arr[1] = 2.5
>>>print(arr)
[1 2 3 4 5]
>>arr[1] = 0.5
>>>print(arr)
[1 0 3 4 5]
```

## Array

- A two-dimensional array (e.g., a matrix) can be created using numpy as follow:

```
>>>import numpy as np
>>>arr = np.array([[1,2,3,4,5],[6,7,8,9,10]])
>>>arr[0]
array([1, 2, 3, 4, 5])
>>>arr[0][0]
1
>>>arr[1][2]
8
>>>arr[-1][-1]
10
>>arr[:,2]
array([3, 8])
```

## Array

```
# Create a 2x2 identity matrix
>>>arr4 = np.eye(2)
>>>print(arr4)
[[1. 0.]
 [0. 1.]]
# Create an array filled with random values
>>arr5 = np.random.random((1,2))
>>>print(arr5)
[[0.00739397, 0.35334824]]
```

## Array

- Numpy methods to create arrays:

```
>>>import numpy as np
# Create an array of all zeros
>>>arr1 = np.zeros((2,2))
>>>print(arr1)
[[0. 0.]
 [0. 0.]]
# Create an array of all ones
>>>arr2 = np.ones((3,2))
>>>print(arr2)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
# Create a constant array
>>>arr3 = np.full((2,2), 5)
>>>print(arr3)
[[5. 5.]
 [5. 5.]]
```

## Numpy array methods

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]],
    float)
# shape property returns the dimension of array
>>>arr1.shape
(2, 5)
# dtype tells the type of the values stored by an array
>>>print(arr1.dtype)
float64
# reshape method returns array of new specified dimension
>>>arr2 = arr1.reshape((5,2))
>>>print(arr2)
[[ 1.  2.]
 [ 3.  4.]
 [ 5.  6.]
 [ 7.  8.]
 [ 9. 10.]]
```

## Numpy array methods

```
# transpose method returns transpose versions of arrays
>>>print(arr1.transpose())
[[ 1.,  6.]
 [ 2.,  7.]
 [ 3.,  8.]
 [ 4.,  9.]
 [ 5., 10.]]
# flatten method returns One-dimensional versions of
    multi-dimensional array
>>>print(arr1.flatten())
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

## Numpy concatenate() function

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2], [3, 4]], float)
>>>arr2 = np.array([[5, 6], [7,8]], float)
>>>print(np.concatenate((arr1,arr2), axis = 0))
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
>>>arr3 = np.concatenate((arr1,arr2),axis = 1)
>>>arr3
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

## Numpy concatenate() function

- concatenate() function is used to join two or more arrays of the same shape along a specified axis.

**Syntax:**

numpy.concatenate((a1,a2,...), axis)

(a1,a2,...) : sequence of arrays of the same type

axis (optional): Axis along which arrays have to be joined. Default is 0.

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2], [3, 4]], float)
>>>arr2 = np.array([[5, 6], [7,8]], float)
>>>print(np.concatenate((arr1,arr2)))
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
```

## Array mathematics

- When standard mathematical operations are used with arrays, they are applied on element-by-element basis.

```
>>>import numpy as np
>>>arr1 = np.array([[1, 2], [3, 4]])
>>>arr2 = np.array([[5, 6], [7,8]])
>>>print(arr1/arr2)
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
>>>print(arr1 + 5)
[[6 7]
 [8 9]]
>>>arr3 = np.array([5,6])
>>>print(arr1 * arr3)
[[ 5 12]
 [15 24]]
>>>arr4 = np.array([[5,6],[7,8], [9, 10]])
>>>print(arr1 * arr4)
ValueError: operands could not be broadcast together with
    shapes (2,2) (3,2)
```

## Array iteration

- Iterate over arrays is possible in a manner similar to that of lists:

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
>>>for x in arr:
...    print(x)
[1. 2.]
[3. 4.]
[5. 6.]
[7. 8.]
>>>for (x, y) in arr:
...    print(x + y)
3.0
7.0
11.0
15.0
```

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
#product operation
>>>print(arr.prod())
40320.0
>>>print(arr.prod(axis = 0))
[105. 384.]
>>>print(arr.prod(axis = 1))
[ 2. 12. 30. 56.]
>>>print(np.prod(arr))
40320.0
>>>print(np.prod(arr, axis = 1))
[ 2. 12. 30. 56.]
>>>print(np.prod(arr, axis = 0))
[105. 384.]
```

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
#sum operation
>>>print(arr.sum())
36.0
>>>print(arr.sum(axis = 0))
[16. 20.]
>>>print(arr.sum(axis = 1))
[ 3.  7. 11. 15.]
>>>print(np.sum(arr))
36.0
>>>np.sum(arr, axis = 1)
[ 3.  7. 11. 15.]
>>>np.sum(arr, axis = 0)
[16. 20.]
```

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
# mean
>>>print(arr.mean())
4.5
>>>print(arr.mean(axis = 0))
[4. 5.]
>>>print(arr.mean(axis = 1))
[1.5 3.5 5.5 7.5]
>>>print(np.mean(arr))
4.5
>>>print(np.mean(arr, axis = 1))
[1.5 3.5 5.5 7.5]
>>>print(np.mean(arr, axis = 0))
[4. 5.]
```

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
# variance
>>>print(arr.var())
5.25
>>>print(arr.var(axis = 0))
[5. 5.]
>>>print(arr.var(axis = 1))
[0.25 0.25 0.25 0.25]
# standard deviation
>>>print(arr.std())
2.29128784747792
>>>print(arr.std(axis = 0))
[2.23606798 2.23606798]
>>>print(arr.std(axis = 1))
[0.5 0.5 0.5 0.5]
```

- We can also find variance and standard deviation using np.var() and np.std().

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[5, 7, 4], [8, 2, 1]], float)
# argmin
>>>print(arr.argmin())
5
>>>print(arr.argmin(axis = 0))
[0 1 1]
>>>print(arr.argmin(axis = 1))
[2 2]
# argmax
>>>print(arr.argmax())
3
>>>print(arr.argmax(axis = 0))
[1 0 0]
>>>print(arr.argmax(axis = 1))
[1 0]
```

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[1, 2], [3, 4], [5, 6], [7,8]], float)
# minimum
>>>print(arr.min())
1.0
>>>print(arr.min(axis = 0))
[1. 2.]
>>>print(arr.min(axis = 1))
[1. 3. 5. 7.]
# maximum
>>>print(arr.max())
8.0
>>>print(arr.max(axis = 0))
[7. 8.]
>>>print(arr.max(axis = 1))
[2. 4. 6. 8.]
```

- We can also use np.min() and np.max() to find minimum and maximum.

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[5, 7, 4], [8, 2, 1]], float)
>>>arr1 = arr.copy()
>>>arr2 = arr.copy()
# sort
>>>arr.sort()
>>>print(arr)
[[4. 5. 7.]
 [1. 2. 8.]]
>>>arr1.sort(axis = 0)
print(arr1)
[[5. 2. 1.]
 [8. 7. 4.]]
>>>arr2.sort(axis = 1)
>>>print(arr2)
[[4. 5. 7.]
 [1. 2. 8.]]
```

## Basic Array Operations

```
>>>import numpy as np
>>>arr = np.array([[5, 7, 4], [8, 2, 1], [3, 9, 6]], float)
# diagonal
>>>print(arr.diagonal())
[5. 2. 6.]
>>>print(np.diag(arr))
[5. 2. 6.]
>>>arr = np.array([[5, 7, 4], [8, 2, 1]], float)
>>>print(arr.diagonal())
[5. 2.]
```

## Vector and matrix mathematics

```
>>>import numpy as np
>>>a = np.array([[4, 1], [3, 2]], float)
>>>b = np.array([2, 1], float)
>>>c = np.array([[1, 1], [4, 1]], float)
# dot product
>>>print(np.dot(b, a))
[11.  4.]
>>>print(np.dot(a, b))
[9. 8.]
>>>print(np.dot(a, c))
[[ 8.  5.]
 [11.  5.]]
>>>print(np.dot(c, a))
[[ 7.  3.]
 [19.  6.]]
```

## Comparison operators

```
>>>import numpy as np
>>>arr1 = np.array([5, 7, 4])
>>>arr2 = np.array([2, 8, 3])
# comparision
>>>print(arr1 > arr2)
[ True False  True]
>>>print(arr1 < arr2)
[False  True False]
>>>print(arr1 != arr2)
[ True  True  True]
>>>print(arr1 > 4)
[ True  True False]
```

## Outer product, inner product, cross product

```
>>>import numpy as np
>>>a = np.array([1, 2, 0], float)
>>>b = np.array([1, 2, 1], float)
# outer product
>>>print(np.outer(b, a))
[[1. 2. 0.]
 [2. 4. 0.]
 [1. 2. 0.]]
>>>print(np.outer(a, b))
[[1. 2. 1.]
 [2. 4. 2.]
 [0. 0. 0.]]
# inner product
>>>print(np.inner(a, b))
5.0
>>>print(np.inner(b, a))
5.0
#cross product
>>>print(np.cross(a, b))
[ 2. -1.  0.]
```

## numpy.linalg

```
>>>import numpy as np
>>>a = np.array([[1, 2, 1], [1, 3, 1], [1, 2, 0]], float)
# determinant
>>>print(np.linalg.det(a))
-1.0
# inverse matrix
>>>b = np.linalg.inv(a)
>>>print(b)
[[ 2. -2.  1.]
 [-1.  1.  0.]
 [ 1. -0. -1.]]
# dot product
>>>print(np.dot(a, b))
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

## numpy.linalg

```
>>>import numpy as np
>>>a = np.array([[1,2,3], [4,5,6]], float)

# Singular Value Decomposition
>>>P, D, Q = np.linalg.svd(a)
>>>print(P)
[[-0.3863177  -0.92236578]
 [-0.92236578  0.3863177 ]]
>>>print(D)
[9.508032    0.77286964]
>>>print(Q)
[[-0.42866713 -0.56630692 -0.7039467 ]
 [ 0.80596391  0.11238241 -0.58119908]
 [ 0.40824829 -0.81649658  0.40824829]]
```

## numpy.linalg

```
>>>import numpy as np
>>>a = np.array([[-2, -4, 2], [-2, 1, 2], [4, 2, 5]],
    float)
# eigen value and eigen vectors
>>>vals, vecs = np.linalg.eig(a)
>>>print(vals)
[-5.  3.  6.]
# normalized eigen vectors as columns of vecs
>>>print(vecs)
[[ 0.81649658  0.53452248  0.05842062]
 [ 0.40824829 -0.80178373  0.35052374]
 [-0.40824829 -0.26726124  0.93472998]]
>>>print(np.dot(a,vecs[:,0]))
[-4.0824829  -2.04124145  2.04124145]
>>>print(vals[0]*vecs[:,0])
[-4.0824829  -2.04124145  2.04124145]
```

## Polynomial Mathematics

- Polynomials in one variable are algebraic expressions that consist of terms in the form $ax^n$ where $n$ is a non-negative (i.e. positive or zero) integer and $a$ is a real number and is called the coefficient of the term.
- The degree of a polynomial in one variable is the largest exponent in the polynomial.
- The roots or also called zeroes of a polynomial $P(x)$ are the values of $x$ for which polynomial $P(x)$ is equal to $0$.

$x^4 - 10x^3 + 35x^2 - 50x + 24$ degree:4

**Roots: 1, 2, 3, 4**

## numpy.poly

- np.roots() is a function to find the roots of a polynomial in python.

```
>>>import numpy as np
# poly = x^4 - 10x^3 + 35x^2 - 50x + 24
>>>poly = [1, -10, 35, -50, 24]
>>>print(np.roots(poly))
[4. 3. 2. 1.]
```

- np.polyval() method evaluates a polynomial at a particular point.

```
>>>import numpy as np
# poly = x^4 - 10x^3 + 35x^2 - 50x + 24
>>>poly = [1, -10, 35, -50, 24]
>>>print(np.polyval(poly, 2))
0
>>>print(np.polyval(poly, 5))
24
```

## numpy.poly

- np.polyder() method returns the coefficient of the derivative of given polynomial.

```
>>>import numpy as np
# poly = x^4 - 10x^3 + 35x^2 - 50x + 24
>>>poly = [1, -10, 35, -50, 24]
>>>print(np.polyder(poly))
[  4 -30  70 -50]
```

- np.polyint() method returns the coefficient array of the integral of the given polynomial.

```
>>>import numpy as np
# poly = x^4 - 10x^3 + 35x^2 - 50x + 24
>>>poly = [1, -10, 35, -50, 24]
>>>print(np.polyint(poly))
[  0.2    -2.5    11.66666667  -25.     24.     0.    ]
#By default, the constant C is set to zero
```

## numpy.poly

```
>>>import numpy as np
# poly1 = x^2 - 2x + 1
>>>poly1 = [1, -2, 1]
# poly2 =  x^3 - 10x^2 + 15x - 12
>>>poly2 = [1, -10, 15, -12]
# addition
>>>print(np.polyadd(poly1, poly2))
[  1  -9  13 -11]
# subtraction
>>>print(np.polysub(poly2, poly1))
[  1 -11  17 -13]
# multiplication
>>>print(np.polymul(poly1, poly2))
[  1 -12  36 -52  39 -12]
# division
>>>quotient, remainder = np.polydiv(poly2, poly1)
>>>print("Quotient:{}, remainder: {}".format(quotient,
    remainder))
Quotient:[ 1. -8.], remainder: [-2. -4.]
```

## Random Number

- NumPy uses a particular algorithm called the Mersenne Twister to generate pseudorandom numbers.
- In python numpy seed is an integer value.
- Any program that starts with the same seed will generate exactly the same sequence of random numbers each time it is run.
- Command to set random number seed np.random.seed(integer_value)
- If random seed is not set then, numpy automatically selects a random seed (based on the time) that is different seed on every run.

## numpy.random.rand()

- numpy.random.rand() generates random value in the half-open interval $[0.0, 1.0)$

  **Syntax:**

  **np.random.rand(d1,d2,d3,...,dn)**

  d1,d2,d3,...,dn(optional): integer values represent the dimension of the returned array we required. By default returns single python float value.

```
>>>import numpy as np
>>>print(np.random.rand()
0.5157163862030659
>>>print(np.random.rand(5))
[0.1542766  0.10432216 0.39556197 0.12116917 0.24038346]
>>>print(np.random.rand(2,2))
[[0.87827584 0.8671831 ]
 [0.12536745 0.3113395 ]]
```

## numpy.random.shuffle()

- numpy.random.shuffle() method is used to used to randomly shuffle the order of the items in a list or array.

```
>>>import numpy as np
>>>l1 = [1, 2, 3, 4, 5]
>>>np.random.shuffle(l1)
>>>print(l1)
[5, 3, 1, 4, 2]
```

## Random Number

- numpy.random.randint(min,max) generates random integer value between the integer values min and max.

```
>>>import numpy as np
>>>print(np.random.randint(10,15))
13
```

- numpy.random.normal($\mu$, $\sigma$ ) generates a normal distributed random number with mean $\mu$ and variance $\sigma$.
- $\mu$ and $\sigma$ sigma are optional, by default takes value 0 and 1 i.e. standard normal distribution.

```
>>>import numpy as np
>>>print(np.random.normal(1.5, 4.0))
0.8599952207824869
>>>print(np.random.normal(1.5, 4.0, 5))
[ 0.40508438,  0.09259794, -1.50249049, -0.20905265,
   -0.17405997]
```

## Pandas

- Pandas is a open source python library.
- Built on top of Numpy with its high performance array-computing features.
- Pandas offers rich data structure and functions to make working with structured data fast, easy, and expressive.
- **Main Features:**
  - Support CSV, Excel, JSON, SQL, SAS, clipboard, HDF5 and many more formats.
  - Data cleansing/cleaning
  - Re-shape and merge data
  - Data Visualisation

## Series

```
import pandas as pd
S = pd.Series() #empty series
print(S)
```

```
Series([], dtype: float64)
```

```
#list to series with default index
S = pd.Series([1,2,3])
print(S)
```

```
0    1
1    2
2    3
dtype: int64
```

```
#list to series with  index
S = pd.Series([1,2,3],index=['a','b','c'])
print(S)
```

```
a    1
b    2
c    3
dtype: int64
```

## Series

- Series is a one-dimensional labeled array capable of holding data of any type (integer, string, oat, python objects, etc.). The axis labels are collectively called index.
- **Syntax:**

  **pandas.Series(data, index, dtype, copy)**

  **data :** array-like, dict, or scalar value

  **index :** array-like or Index (1d), Values must be hashable and have the same length as data. Non-unique index values are allowed. Will default to RangeIndex **(0, 1, 2,. . . , n)** if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

  **dtype :** data type

  **copy(boolean) :** copy data, default **False**

## Series

```
# dictionary to series
S = pd.Series({'a': 1, 'b': 2, 'c': 3})
print(S)
```

```
a    1
b    2
c    3
dtype: int64
```

```
S = pd.Series({'a': 1, 'b': 2, 'c': 3}, index = ['a', 'c',
    'd', 'b'])
print(S)
```

```
a    1.0
c    3.0
d    NaN
b    2.0
dtype: float64
```

## Indexing and Slicing

```
import pandas as pd
S = pd.Series((1, 2, 3, 4), index = ['a', 'b', 'c', 'b'])
print(S)
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

```
print(S[1])
```

```
2
```

```
print(S[:2])
```

```
a    1
b    2
dtype: int64
```

## Operations on Series

```
print(S[S>2])
```

```
c    3
d    4
dtype: int64
```

```
print(S>2)
```

```
a    False
b    False
c     True
d     True
dtype: bool
```

```
print(S*4)
```

```
a     4
b     8
c    12
d    16
dtype: int64
```

## Indexing and Slicing

```
import pandas as pd
S = pd.Series((1, 2, 3, 4), index = ['a', 'b', 'c', 'b'])
print(S)
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

```
print(S['a'])
```

```
1
```

```
print(S[['a','b']])
```

```
a    1
b    2
dtype: int64
```

## DataFrame

- ☐ A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.
- ☐ **Features:**
  - ☐ Heterogeneous tabular data structure
  - ☐ Size – Mutable
  - ☐ Labeled axes (rows and columns)
  - ☐ Can Perform Arithmetic operations on rows and columns

## DataFrame

- The most common way to create a DataFrame is by using the dictionary of equal-length list.

```
import pandas as pd
data = {'Date': ['01-10-2001','15-02-2008','10-06-2010'],
        'Name': ['TCS','IBM','GOOG'],
        'Shares': [80, 30, 90],
        'Price': [15, 17.3, 30.2] }
df = pd.DataFrame(data)
print(df)
```

| | Date | Name | Shares | Price |
|---|---|---|---|---|
| **0** | 2001-10-01 | TCS | 80 | 15.0 |
| **1** | 2008-02-15 | IBM | 30 | 17.3 |
| **2** | 2010-06-10 | GOOG | 90 | 30.2 |

## DataFrame

```
# any column of the DataFrame can be set as index using
    set_index()
>>>df = df.set_index(['Name'])
>>>df
```

| | Date | Shares | Price | owner |
|---|---|---|---|---|
| **Name** | | | | |
| **TCS** | 01-10-2001 | 80 | 15.0 | Ram |
| **IBM** | 15-02-2008 | 30 | 17.3 | Shyam |
| **GOOG** | 10-06-2010 | 90 | 30.2 | Mohan |

## DataFrame

```
# addition of extra column
df['owner'] = ['Ram', 'Shyam', 'Mohan']
print(df)
```

| | Date | Name | Shares | Price | owner |
|---|---|---|---|---|---|
| **0** | 01-10-2001 | TCS | 80 | 15.0 | Ram |
| **1** | 15-02-2008 | IBM | 30 | 17.3 | Shyam |
| **2** | 10-06-2010 | GOOG | 90 | 30.2 | Mohan |

```
# add Row index
df.index = ['1st', '2nd', '3rd']
print(df)
```

| | Date | Name | Shares | Price | owner |
|---|---|---|---|---|---|
| **1st** | 01-10-2001 | TCS | 80 | 15.0 | Ram |
| **2nd** | 15-02-2008 | IBM | 30 | 17.3 | Shyam |
| **3rd** | 10-06-2010 | GOOG | 90 | 30.2 | Mohan |

## Data Accessing

- Data can be accessed in two ways i.e. using row and column index

```
# Acessing data using column
>>>df['Price']
```

```
Name
TCS     15.0
IBM     17.3
GOOG    30.2
Name: Price, dtype: float64
```

```
# Acessing data using row
>>>df.loc['IBM']
```

```
Date      15-02-2008
Shares            30
Price           17.3
owner          Shyam
Name: IBM, dtype: object
```

# Delete Column

- Column can be deleted using **del** or **drop** commands

```
>>>del df['Shares']
>>>df
```

| | Date | Price | owner |
|---|---|---|---|
| **Name** | | | |
| **TCS** | 01-10-2001 | 15.0 | Ram |
| **IBM** | 15-02-2008 | 17.3 | Shyam |
| **GOOG** | 10-06-2010 | 30.2 | Mohan |

```
>>>df.drop('owner', axis = 1)
```

| | Date | Price |
|---|---|---|
| **Name** | | |
| **TCS** | 01-10-2001 | 15.0 |
| **IBM** | 15-02-2008 | 17.3 |
| **GOOG** | 10-06-2010 | 30.2 |

# File Reading

```
>>>import pandas as pd
>>>data = pd.read_csv('property_data.csv')
>>>data.head()
#head(n) show only first n rows of the DataFrame, by
    default n = 5
```

| | PID | ST_NUM | ST_NAME | OWN_OCCUPIED | NUM_BEDROOMS | NUM_BATH | SQ_FT |
|---|---|---|---|---|---|---|---|
| **0** | 100001000 | 104.0 | PUTNAM | Y | 3.0 | 1.0 | 1000.0 |
| **1** | 100004000 | 201.0 | BERKELEY | 12 | 1.0 | NaN | 700.0 |
| **2** | 100002000 | 197.0 | LEXINGTON | N | 3.0 | 3.0 | NaN |
| **3** | 100006000 | 207.0 | BERKELEY | Y | NaN | 1.0 | 800.0 |
| **4** | 100007000 | 210.0 | WASHINGTON | NaN | 2.0 | NaN | 950.0 |

# Property_data

- Let **property_data.csv** is a comma separated file having following data:

| PID | ST_NUM | ST_NAME | OWN_OCCUPIED | NUM_BEDROOMS | NUM_BATH | SQ_FT |
|---|---|---|---|---|---|---|
| 100001000 | 104 | PUTNAM | Y | 3 | 1 | 1000 |
| 100004000 | 201 | BERKELEY | 12 | 1 | NaN | 700 |
| 100002000 | 197 | LEXINGTON | N | 3 | 3 | |
| 100006000 | 207 | BERKELEY | Y | | 1 | 800 |
| 100007000 | 210 | WASHINGTON | | 2 | | 950 |
| 100009000 | 215 | TREMONT | Y | 2 | 2 | 1800 |
| 100003000 | | LEXINGTON | N | | 1 | 850 |
| 100008000 | 213 | TREMONT | Y | 1 | 1 | |

# Null Value

- **.isnull()** returns Dataframe of boolean values which are **True** for NaN values and **False** for not NaN values

```
>>>data.isnull()
```

| | PID | ST_NUM | ST_NAME | OWN_OCCUPIED | NUM_BEDROOMS | NUM_BATH | SQ_FT |
|---|---|---|---|---|---|---|---|
| **0** | False | False | False | False | False | False | False |
| **1** | False | False | False | False | False | True | False |
| **2** | False | False | False | False | False | False | True |
| **3** | False | False | False | False | True | False | False |
| **4** | False | False | False | True | False | True | False |
| **5** | False | False | False | False | False | False | False |
| **6** | False | True | False | False | True | False | False |
| **7** | False | False | False | False | False | False | True |

## Null Value

- **.isnull().any(axis)** returns a boolean Series correspond to row number if **axis = 1** or boolean series correspond to columns **axis = 0** (default value) with **True** if they contain atleast one NaN value else **False**

```
>data.isnull().any(axis = 1)
```

```
0     False
1      True
2      True
3      True
4      True
5     False
6      True
7      True
dtype: bool
```

```
>>>data.isnull()
```

```
PID             False
ST_NUM           True
ST_NAME         False
OWN_OCCUPIED     True
NUM_BEDROOMS     True
NUM_BATH         True
SQ_FT            True
dtype: bool
```

## Null Value

- **.fillna()** is used to fill NaN values in Dataframe.

```
>>>data['SQ_FT'] =
    data['SQ_FT'].fillna(round(data['SQ_FT'].mean()))
>>>data['SQ_FT']
```

```
0    1000.0
1     700.0
2    1017.0
3     800.0
4     950.0
5    1800.0
6     850.0
7    1017.0
Name: SQ_FT, dtype: float64
```

## Null Value

- **.fillna()** is used to fill NaN values in Dataframe.

```
>>>data['NUM_BEDROOMS'] = data['NUM_BEDROOMS'].fillna(1)
>>>data['NUM_BEDROOMS']
```

```
0    3.0
1    1.0
2    3.0
3    1.0
4    2.0
5    2.0
6    1.0
7    1.0
Name: NUM_BEDROOMS, dtype: float64
```

## Data Filtering

- Data can be filtered by providing some boolean expression in DataFrame.

```
>>>data[(data['SQ_FT'] > 900) & (data['NUM_BEDROOMS'] > 1)]
```

| | PID | ST_NUM | ST_NAME | OWN_OCCUPIED | NUM_BEDROOMS | NUM_BATH | SQ_FT |
|---|---|---|---|---|---|---|---|
| **0** | 100001000 | 104.0 | PUTNAM | Y | 3.0 | 1.0 | 1000.0 |
| **2** | 100002000 | 197.0 | LEXINGTON | N | 3.0 | 3.0 | 1017.0 |
| **4** | 100007000 | 210.0 | WASHINGTON | NaN | 2.0 | NaN | 950.0 |
| **5** | 100009000 | 215.0 | TREMONT | Y | 2.0 | 2.0 | 1800.0 |

# Plotting

- Pandas supports the matplotlib library and can be used to plot the data as well.

```
import matplotlib.pyplot as plt
data.plot.bar(x = 'ST_NAME', y ='SQ_FT' )
plt.show()
```