https://github.com/Sivsankars/Datascience

LAB CYCLE – 1

DATA SCIENCE LAB

Submitted By,

Sivasankar S

S3,MCA

Roll No : 233

PROGRAM – 1

AIM : Matrix operations (using vectorization) and transformation using python and SVD using Python , numpy

import numpy as np

1) Create an array of 10 zeros

np.zeros(10) array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

2) Create an array of 10 ones

np.ones(10)

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

3) Create an array of 10 ves

np.ones(10)*5

array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.])

4) Create an array of the integers from 10 to 50

np.arange(10,51)

array([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, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50])

5) Create an array of all the even integers from 10 to 50

np.arange(10,51,2)

array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50])

6) Create a 3x3 matrix with values ranging from 0 to 8

np.arange(9).reshape(3,3)

array([[0, 1, 2], [3, 4, 5],

[6, 7, 8]])

7) Create a 3x3 identity matrix

np.eye(3)

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

8) Use NumPy to generate a random number between 0 and 1

np.random.rand
array([
0.42829726
])

9) Use NumPy to generate an array of 25 random numbers sampled from a standard normal distribution

np.random.normal(0,1,25)

array([-0.81945275, 0.03694451, -0.61741758, -0.37232392, -0.18165511, 0.37236063, 2.04195743, -1.90941553, 2.52910468, -3.07256018, -1.01467785, 0.77283185, 1.94027155, -0.443968 , 0.16738543,
-0.20444985, 0.90250309, 0.9922238 , -0.17137933, -0.94617983,
-1.01344804, -0.24003194, 1.35258435, -0.18654704, -1.24966977])

10) Create the following matrix:

arr = np.linspace(0.01, 1, 100).reshape(10,10)
arr

array([[0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1], [0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2],
[0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3],
[0.31, 0.32, 0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4],
[0.41, 0.42, 0.43, 0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5],

[0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59, 0.6],

[0.61, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67, 0.68, 0.69, 0.7],

[0.71, 0.72, 0.73, 0.74, 0.75, 0.76, 0.77, 0.78, 0.79, 0.8],

[0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.88, 0.89, 0.9],

[0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.]])

11) Create an array of 20 linearly spaced points between 0 and 1:

np.linspace(0,1,20)

array([0.   , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.   ])

Numpy Indexing and Selection

Now you will be given a few matrices, and be asked to replicate the resulting matrix outputs:

mat =
np.arange(1,26).reshape(5,5) mat

array([[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]])

mat[2:,1:]

array([[12, 13, 14, 15],
[17, 18, 19, 20],

[22, 23, 24, 25]])

mat[3][4]

20

mat[0:3,1:2]

array
([[

```
    2],
    [ 7],
       [12]])
```

```
mat[4][:]
```

```
array([21, 22, 23, 24, 25])
```

```
mat[3:,:]
```

```
array([[16, 17, 18, 19, 20],
[21, 22, 23, 24, 25]])
```

Get the sum of all the values in mat

```
sum(sum(mat))
```

```
    325
```

Get the standard deviation of the values in mat

```
mat.std()
```

```
    7.211102550927978
```

Get the sum of all the columns in mat

```
sum(mat)
```

```
array([55, 60, 65, 70, 75])
```

RESULT

Output obtained

successfully.

PROGRAM – 2

AIM : Programs using matplotlib / plotly / bokeh / seaborn for data visualisation.

Installation

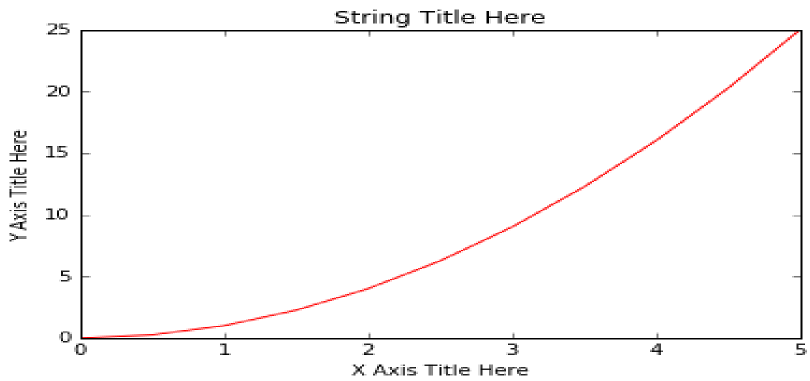
You'll need to install matplotlib first with either:

conda install matplotlib

or pip install matplotlib

Importing

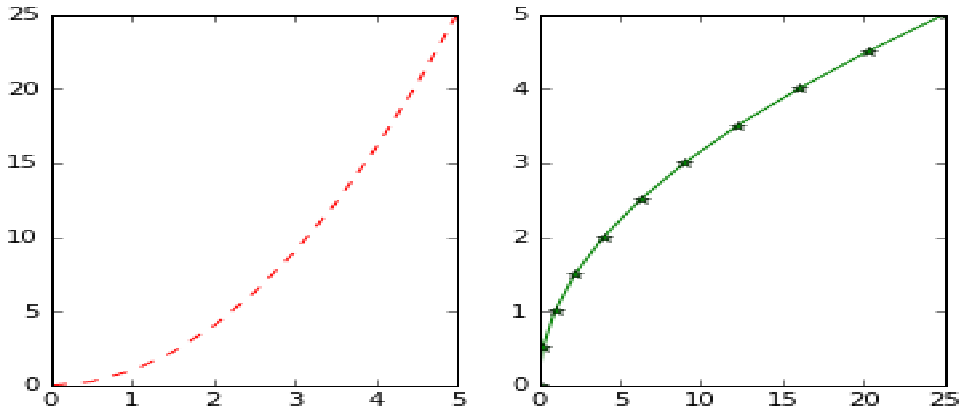
```
import matplotlib.pyplot as plt
plt.plot(x, y, 'r') # 'r' is the color red
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show()
```



Creating Multiplots on Same Canvas

[ ]

```
# plt.subplot(nrows, ncols, plot_number)
plt.subplot(1,2,1)
plt.plot(x, y, 'r--') # More on color options later
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```



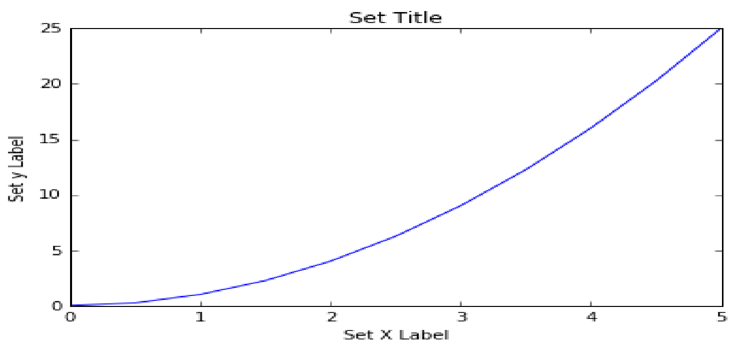
Introduction to the Object Oriented Method

The main idea in using the more formal Object Oriented method is to create figure objects and then just call methods or attributes off of that object. This approach is nicer when dealing with a canvas that has multiple plots on it.

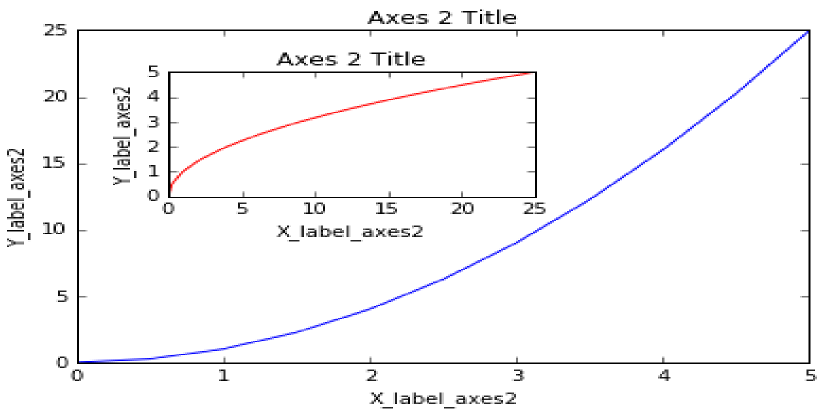
To begin we create a figure instance. Then we can add axes to that figure:

[ ]

```
# Create Figure (empty canvas)
fig = plt.figure()
# Add set of axes to figure
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)
# Plot on that set of axes
axes.plot(x, y, 'b')
axes.set_xlabel('Set X Label') # Notice the use of set_ to begin methods
axes.set_ylabel('Set y Label')
axes.set_title('Set Title')
```



```
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# Larger Figure Axes 1
axes1.plot(x, y, 'b')
axes1.set_xlabel('X_label_axes2')
axes1.set_ylabel('Y_label_axes2')
axes1.set_title('Axes 2 Title')

# Insert Figure Axes 2
axes2.plot(y, x, 'r')
axes2.set_xlabel('X_label_axes2')
axes2.set_ylabel('Y_label_axes2')
axes2.set_title('Axes 2 Title');
```



subplots()

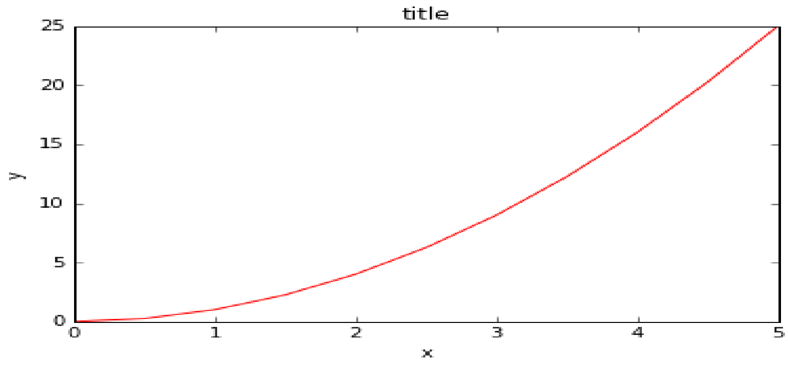
The plt.subplots() object will act as a more automatic axis manager.

Basic use cases:

[ ]

```
# Use similar to plt.figure() except use tuple unpacking to grab fig and axes

fig, axes = plt.subplots()

# Now use the axes object to add stuff to plot

axes.plot(x, y, 'r')

axes.set_xlabel('x')

axes.set_ylabel('y')
```

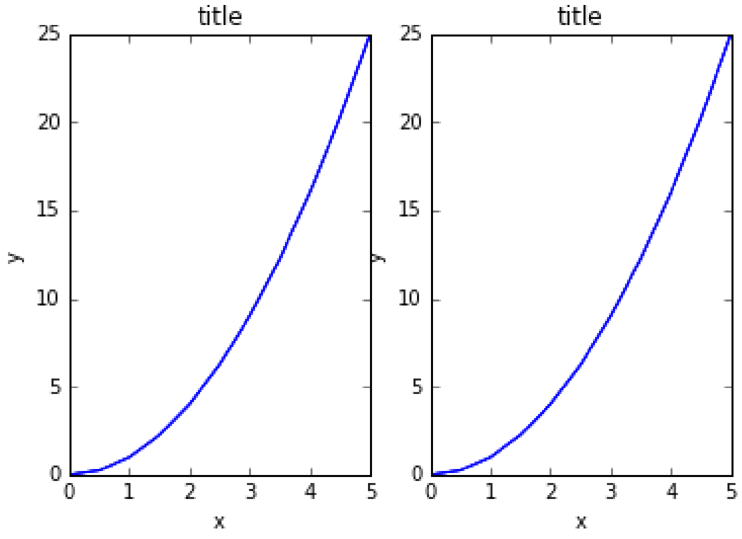
```
axes.set_title('title');
```



We can iterate through this array:

[ ]

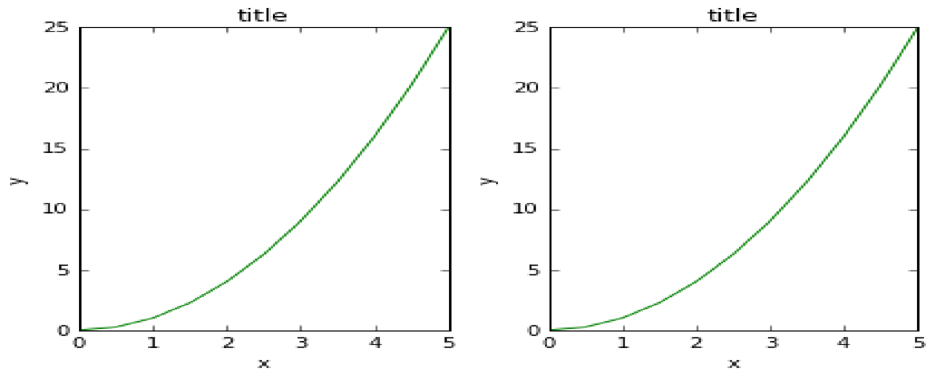
```
for ax in axes:
    ax.plot(x, y, 'b')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')
# Display the figure object
fig
```



A common issue with matplolib is overlapping subplots or figures. We ca use **fig.tight_layout()** or **plt.tight_layout()** method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

[ ]

```
fig, axes = plt.subplots(nrows=1, ncols=2)
for ax in axes:
    ax.plot(x, y, 'g')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')
fig
plt.tight_layout()
```

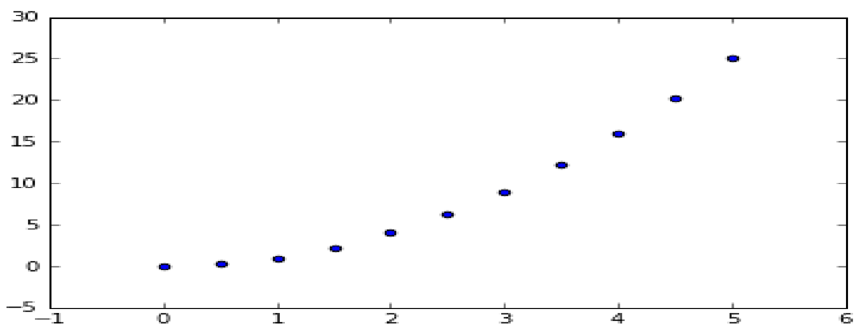


Special Plot Types

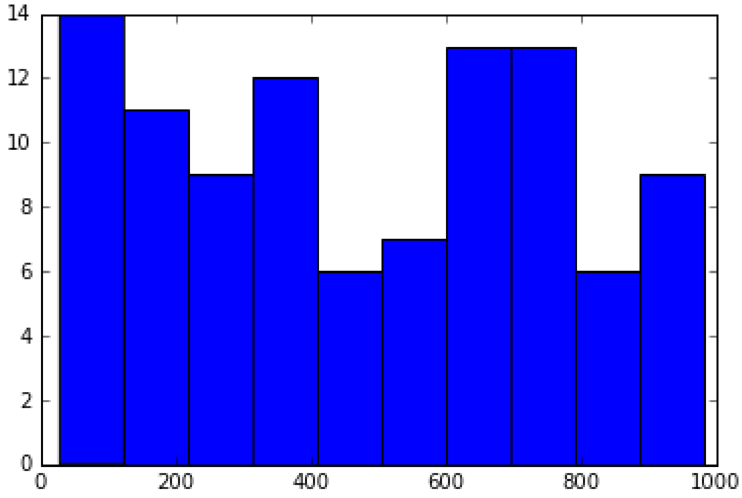
There are many specialized plots we can create, such as barplots, histograms, scatter plots, and much more. Most of these type of plots we will actually create using seaborn, a statistical plotting library for Python. But here are a few examples of these type of plots:

[ ]

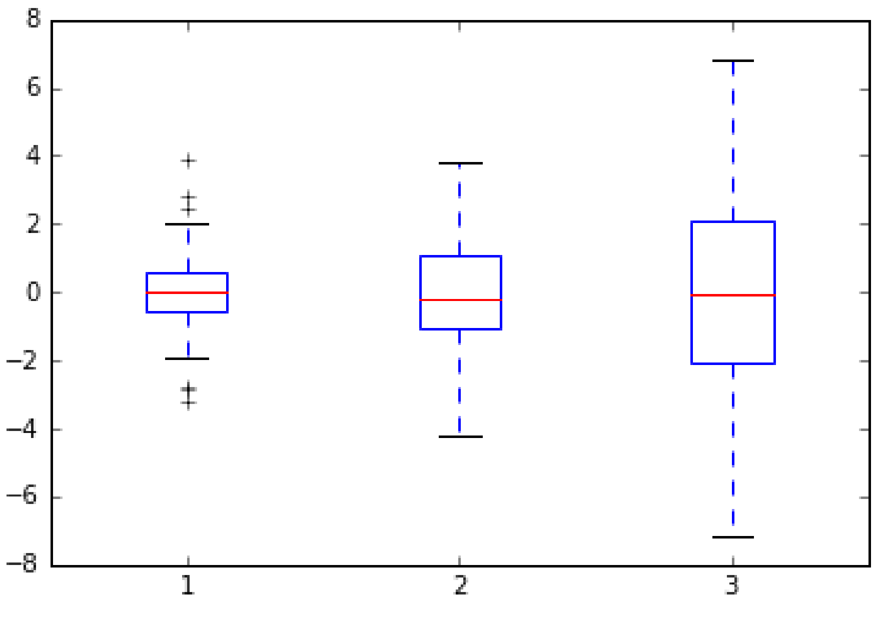
```
plt.scatter(x,y)
```



```
from random import sample
data = sample(range(1, 1000), 100)
plt.hist(data)
```




```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data,vert=True,patch_artist=True);
```



RESULT

Output obtained successfully.

PROGRAM – 3

AIM : Programs to handle data using pandas

Series

The first main data type we will learn about for pandas is the Series data type. Let's import Pandas and explore the Series object.

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object.

```
import numpy as np
import pandas as pd
```

Creating a Series

You can convert a list,numpy array, or dictionary to a Series:

[ ]

```
labels = ['a','b','c']
my_list = [10,20,30]
arr = np.array([10,20,30])
d = {'a':10,'b':20,'c':30}
```

** Using Lists**

[ ]

```
pd.Series(data=my_list)
```

```
0    10
1    20
2    30
dtype: int64
```

[ ]

```
pd.Series(data=my_list,index=labels)
```

```
a    10
b    20
c    30
dtype: int64
```

[ ]

```
pd.Series(my_list,labels)
```

```
a    10
b    20
c    30
dtype: int64
```

** NumPy Arrays **

[ ]

```
pd.Series(arr)
```

```
0    10
1    20
2    30
dtype: int64
```

[ ]

```
pd.Series(arr,labels)
```

```
a    10
b    20
c    30
dtype: int64
```

** Dictionary**

[ ]

```
pd.Series(d)
```

```
a    10
b    20
c    30
dtype: int64
```

Data in a Series

A pandas Series can hold a variety of object types:

[ ]

```
pd.Series(data=labels)
```

```
0    a
1    b
```

2   c
dtype: object

[ ]

```
# Even functions (although unlikely that you will use this)
pd.Series([sum,print,len])
```

0   <built-in function sum>
1   <built-in function print>
2   <built-in function len>
dtype: object

Using an Index

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast look ups of information (works like a hash table or dictionary).

Let's see some examples of how to grab information from a Series. Let us create two sereis, ser1 and ser2:

[ ]

```
ser1 = pd.Series([1,2,3,4],index = ['USA', 'Germany','USSR', 'Japan'])
```

[ ]

```
ser1
```

```
USA        1
Germany    2
USSR       3
Japan      4
dtype: int64
```

[ ]

```
ser2 = pd.Series([1,2,5,4],index = ['USA', 'Germany','Italy', 'Japan'])
```

[ ]

```
ser2
```

```
USA        1
Germany    2
Italy      5
Japan      4
dtype: int64
```

[ ]

```
ser1['USA']
```

1

Operations are then also done based off of index:

[ ]

```
ser1 + ser2
```

Germany   4.0
Italy   NaN
Japan   8.0
USA   2.0
USSR   NaN
dtype: float64

DATAFRAMES

DataFrames are the workhorse of pandas and are directly inspired by the R programming language. We can think of a DataFrame as a bunch of Series objects put together to share the same index. Let's use pandas to explore this topic!

[ ]

```
import pandas as pd

import numpy as np
```

[ ]

```
from numpy.random import randn

np.random.seed(101)
```

[ ]

```
df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y Z'.split())
```

[ ]

Selection and Indexing

Let's learn the various methods to grab data from a DataFrame

---

[ ]

```
df['W']
```

```
A    2.706850
B    0.651118
C   -2.018168
D    0.188695
E    0.190794
Name: W, dtype: float64
```

[ ]

```
# Pass a list of column names
df[['W','Z']]
```

[ ]

```
# SQL Syntax (NOT RECOMMENDED!)
df.W
```

```
A    2.706850
B    0.651118
C   -2.018168
D    0.188695
E    0.190794
Name: W, dtype: float64
```

DataFrame Columns are just Series

[ ]

```
type(df['W'])
```

```
pandas.core.series.Series
```

Creating a new column:

[ ]

```
df['new'] = df['W'] + df['Y']
```

[ ]

```
df
```

** Removing Columns**

[ ]

```
df.drop('new',axis=1)
```

[ ]

```
# Not inplace unless specified!
df
```

[ ]

```
df.drop('new',axis=1,inplace=True)
```

[ ]

```
df
```

Can also drop rows this way:

[ ]

```
df.drop('E',axis=0)
```

** Selecting Rows**

[ ]

```
df.loc['A']
```

```
W    2.706850
X    0.628133
Y    0.907969
Z    0.503826
Name: A, dtype: float64
```

Or select based off of position instead of label

[ ]

```
df.iloc[2]
```

```
W   -2.018168
X    0.740122
```

Y   0.528813
Z   -0.589001
Name: C, dtype: float64

** Selecting subset of rows and columns **

[ ]

```
df.loc['B','Y']
```

-0.84807698340363147

[ ]

```
df.loc[['A','B'],['W','Y']]
```

Conditional Selection

An important feature of pandas is conditional selection using bracket notation, very similar to numpy:

```
df>0
```

[ ]

```
df[df>0]
```

[ ]

```
df[df['W']>0]
```

[ ]

```
df[df['W']>0]['Y']
```

A   0.907969
B   -0.848077
D   -0.933237
E   2.605967
Name: Y, dtype: float64

[ ]

```
df[df['W']>0][['Y','X']]
```

For two conditions you can use | and & with parenthesis:

[ ]

```
df[(df['W']>0) & (df['Y'] > 1)]
```

More Index Details

Let's discuss some more features of indexing, including resetting the index or setting it something else. We'll also talk about index hierarchy!

[ ]

```
df
```

[ ]

```
# Reset to default 0,1...n index
df.reset_index()
```

[ ]

```
newind = 'CA NY WY OR CO'.split()
```

[ ]

```
df['States'] = newind
df.set_index('States')
df.set_index('States',inplace=True)
```

Multi-Index and Index Hierarchy

Let us go over how to work with Multi-Index, first we'll create a quick example of what a Multi-Indexed DataFrame would look like:

[ ]

```
# Index Levels
outside = ['G1','G1','G1','G2','G2','G2']
inside = [1,2,3,1,2,3]
hier_index = list(zip(outside,inside))
hier_index = pd.MultiIndex.from_tuples(hier_index)
```

[ ]

```
hier_index
```

```
MultiIndex(levels=[['G1', 'G2'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

[ ]

```
df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
```

Now let's show how to index this! For index hierarchy we use df.loc[], if this was on the columns axis, you would just use normal bracket notation df[]. Calling one level of the index returns the sub-dataframe:

[ ]

```
df.loc['G1']
```

[ ]

```
df.loc['G1'].loc[1]
```

```
A    0.153661
B    0.167638
Name: 1, dtype: float64
```

[ ]

```
df.index.names
```

```
FrozenList([None, None])
```

[ ]

```
df.index.names = ['Group','Num']
```

[ ]

```
df
```

[ ]

```
df.xs('G1')
```

[ ]

```
df.xs(['G1',1])
```

```
A    0.153661
B    0.167638
Name: (G1, 1), dtype: float64
```

[ ]

```
df.xs(1,level='Num')
```

Missing Data

Let's show a few convenient methods to deal with Missing Data in pandas:

[ ]

```
import numpy as np
import pandas as pd
```

[ ]

```
df = pd.DataFrame({'A':[1,2,np.nan],
                  'B':[5,np.nan,np.nan],
                  'C':[1,2,3]})
```

[ ]

```
df.dropna()
```

[ ]

```
df.dropna(axis=1)
```

[ ]

```
df.dropna(thresh=2)
```

[ ]

```
df.fillna(value='FILL VALUE')
```

[ ]

```
df['A'].fillna(value=df['A'].mean())
```

```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

Groupby

The groupby method allows you to group rows of data together and call aggregate functions

---

[ ]

```
import pandas as pd
# Create dataframe
data = {'Company':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
    'Person':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],'Sales':[200,120,340,124,243,350
```

use the .groupby() method to group rows together based off of a column name. For instance let's group based off of Company. This will create a DataFrameGroupBy object:**

[ ]

```
df.groupby('Company')
<pandas.core.groupby.DataFrameGroupBy object at 0x113014128>
```

You can save this object as a new variable:

[ ]

```
by_comp = df.groupby("Company")
```

And then call aggregate methods off the object:

[ ]

```
by_comp.mean()
```

[ ]

```
df.groupby('Company').mean()
```

More examples of aggregate methods:

[ ]

```
by_comp.std()
```

[ ]

```
by_comp.min()
```

[ ]

```
by_comp.max()
```

[ ]

```
by_comp.count()
```

[ ]

```
by_comp.describe()
```

[ ]

```
by_comp.describe().transpose()
```

[ ]

```
by_comp.describe().transpose()['GOOG']
```

Merging, Joining, and Concatenating

There are 3 main ways of combining DataFrames together: Merging, Joining and Concatenating. In this lecture we will discuss these 3 methods with examples.

Example DataFrames

[ ]

[ ]

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
```

[ ]

```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},[ ]
```

```
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])
```

Concatenation

Concatenation basically glues together DataFrames. Keep in mind that dimensions should match along the axis you are concatenating on. You can use **pd.concat** and pass in a list of DataFrames to concatenate together:

[ ]

```
pd.concat([df1,df2,df3])
```

[ ]

```
pd.concat([df1,df2,df3],axis=1)
```

Example DataFrames

[ ]

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})
```

Merging

The **merge** function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. For example:

[ ]

```
pd.merge(left,right,how='inner',on='key')
```

Or to show a more complicated example:

[ ]

```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                        'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                               'key2': ['K0', 'K0', 'K0', 'K0'],
                                  'C': ['C0', 'C1', 'C2', 'C3'],
                                  'D': ['D0', 'D1', 'D2', 'D3']})
```

[ ]

```
pd.merge(left, right, on=['key1', 'key2'])
```

[ ]

```
pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

[ ]

```
pd.merge(left, right, how='right', on=['key1', 'key2'])
```

[ ]

```
pd.merge(left, right, how='left', on=['key1', 'key2'])
```

Joining

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

[ ]

```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                      index=['K0', 'K1', 'K2'])

right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                    'D': ['D0', 'D2', 'D3']},
                      index=['K0', 'K2', 'K3'])
```

[ ]

```
left.join(right)
```

[ ]

```
left.join(right, how='outer')
```

Operations

There are lots of operations with pandas that will be really useful to you, but don't fall into any distinct category. Let's show them here in this lecture:

[ ]

```
import pandas as pd
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[444,555,666,444],'col3':['abc','def','ghi','xyz']})
df.head()
```

Info on Unique Values

[ ]

```
df['col2'].unique()
```

array([444, 555, 666])

[ ]

```
df['col2'].nunique()
```

3

[ ]

```
df['col2'].value_counts()
```

```
444    2
555    1
666    1
Name: col2, dtype: int64
```

Selecting Data

```
#Select from DataFrame using criteria from multiple columns
newdf = df[(df['col1']>2) & (df['col2']==444)]
```

Applying Functions

[ ]

```
def times2(x):
    return x*2
```

[ ]

```
df['col1'].apply(times2)
```

```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

[ ]

```
df['col3'].apply(len)
```

```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

[ ]

```
df['col1'].sum()
```

```
10
```

** Permanently Removing a Column**

[ ]

```
del df['col1']
```

[ ]

```
df
```

** Get column and index names: **

[ ]

```
df.columns
```

Index(['col2', 'col3'], dtype='object')

[ ]

```
df.index
```
RangeIndex(start=0, stop=4, step=1)

** Sorting and Ordering a DataFrame:*

[ ]

```
df.sort_values(by='col2') #inplace=False by default
```

** Find Null Values or Check for Null Values**

[ ]

```
df.isnull()
```

[ ]

```
# Drop rows with NaN Values
df.dropna()
```

** Filling in NaN values with something else: **

[ ]

```
import numpy as np
```

[ ]

```
df = pd.DataFrame({'col1':[1,2,3,np.nan],
                   'col2':[np.nan,555,666,444],
                   'col3':['abc','def','ghi','xyz']})
df.head()
```

[ ]

```
df.fillna('FILL')
```

[ ]

```
data = {'A':['foo','foo','foo','bar','bar','bar'],
```

SF Salaries Exercise

Welcome to a quick exercise for you to practice your pandas skills! We will be using the SF Salaries Dataset from Kaggle! Just follow along and complete the tasks outlined in bold below. The tasks will get harder and harder as you go along.

** Import pandas as pd.**

```
import pandas as pd
```

In [3]:

** Read Salaries.csv as a dataframe called sal.**

```
sal = pd.read_csv('Salaries.csv',low_memory=False)
```

** Check the head of the DataFrame. **

```
sal.head()
```

** Use the .info() method to find out how many entries there are.**

```
sal.info()
```

[ ]

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 148654 entries, 0 to 148653
Data columns (total 13 columns):
Id                  148654 non-null int64
EmployeeName        148654 non-null object
JobTitle            148654 non-null object
BasePay             148045 non-null float64
OvertimePay         148650 non-null float64
OtherPay            148650 non-null float64
Benefits            112491 non-null float64
TotalPay            148654 non-null float64
TotalPayBenefits    148654 non-null float64
Year                148654 non-null int64
Notes               0 non-null float64
Agency              148654 non-null object
Status              0 non-null float64
dtypes: float64(8), int64(2), object(3)
memory usage: 14.7+ MB
```

What is the average BasePay ?

[ ] `sal['BasePay'].mean()`

66325.44884050643

** What is the highest amount of OvertimePay in the dataset ? **

[ ] sal['OvertimePay'].max()

245131.88

** What is the job title of JOSEPH DRISCOLL ? Note: Use all caps, otherwise you may get an answer that doesn't match up (there is also a lowercase Joseph Driscoll). **

---

24   CAPTAIN, FIRE SUPPRESSIONName: JobTitle, dtype: object

** How much does JOSEPH DRISCOLL make (including benefits)? **

[ ]
sal[sal['EmployeeName']=='JOSEPH DRISCOLL']['JobTitle']

24   270324.91
Name: TotalPayBenefits, dtype: float64

** What is the name of highest paid person (including benefits)?**

[ ] sal[sal['TotalPayBenefits']== sal['TotalPayBenefits'].max()]

** What is the name of lowest paid person (including benefits)? Do you notice something strange about how much he or she is paid?**

[ ]
sal[sal['TotalPayBenefits']== sal['TotalPayBenefits'].min()]

** What was the average (mean) BasePay of all employees per year? (2011-2014) ? **

[ ] sal.groupby('Year').mean()['BasePay']
Year
2011   63595.956517
2012   65436.406857

2013   69630.030216
2014   66564.421924
Name: BasePay, dtype: float64

** How many unique job titles are there? **

[ ] sal['JobTitle'].nunique()
2159

** What are the top 5 most common jobs? **

[ ] sal['JobTitle'].value_counts().head(5)

Transit Operator   7036
Special Nurse   4389
Registered Nurse   3736
Public Svc Aide-Public Works   2518
Police Officer 3   2421
Name: JobTitle, dtype: int64

** How many Job Titles were represented by only one person in 2013? (e.g. Job Titles with only one occurence in 2013?) **

[ ] sum(sal[sal['Year']==2013]['JobTitle'].value_counts() == 1)

202

** How many people have the word Chief in their job title? (This is pretty tricky) **

[ ] sum(sal['JobTitle'].apply(**lambda** x: chief_string(x)))
 477

** Bonus: Is there a correlation between length of the Job Title string and Salary? **

[ ] sal[['title_len','TotalPayBenefits']].corr()

[ ] title_lenTotalPayBenefitstitle_len1.000000-0.036878TotalPayBenefits-0.0368781.000000

Ecommerce Purchases Exercise

** Import pandas and read in the Ecommerce Purchases csv file and set it to a DataFrame called ecom. **

```
import pandas as pd
```

In [3]:

```
file = pd.read_csv('/content/Ecommerce Purchases')
ecom = pd.DataFrame(file)
```

Check the head of the DataFrame.

```
ecom.head(5)
```

** How many rows and columns are there? **

```
ecom.shape
```

Out[5]:

(10000, 14)

```
[ ]<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
Address            10000 non-null object
Lot                10000 non-null  object
AM or PM           10000 non-null object
Browser Info       10000 non-null object
Company            10000 non-null object
Credit Card        10000 non-null int64
CC Exp Date        10000 non-null object
CC Security Code   10000 non-null int64
CC Provider        10000 non-null object
Email              10000 non-null object
Job                10000 non-null object
IP Address         10000 non-null object
Language           10000 non-null object
Purchase Price     10000 non-null float64
dtypes: float64(1), int64(2), object(11)
memory usage: 1.1+ MB
```

** What is the average Purchase Price? **

```
[ ]  ecom['Purchase Price'].mean()
```

50.34730200000025

** What were the highest and lowest purchase prices? **

[ ] ecom['Purchase Price'].max()
99.989999999999995

[ ] ecom['Purchase Price'].min()
0.0

** How many people have English 'en' as their Language of choice on the website? **

```
[ ] ecom[ecom['Language']=='en'].count()
Address            1098
Lot                1098
AM or PM           1098
Browser Info       1098
Company            1098
Credit Card        1098
CC Exp Date        1098
CC Security Code   1098
CC Provider        1098
Email              1098
Job                1098
IP Address         1098
Language           1098
Purchase Price     1098
dtype: int64
```

** How many people have the job title of "Lawyer" ? **

```
[ ]  ecom[ecom['Job']=='Lawyer'].count()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30 entries, 470 to 9979
Data columns (total 14 columns):
Address            30 non-null object
Lot                30 non-null  object
AM or PM           30 non-null object
Browser Info       30 non-null object
Company            30 non-null object
Credit Card        30 non-null int64
CC Exp Date        30 non-null object
CC Security Code   30 non-null int64
CC Provider        30 non-null object
```

Email 30 non-null object
Job 30 non-null object
IP Address 30 non-null object
Language 30 non-null object
Purchase Price 30 non-null float64
dtypes: float64(1), int64(2), object(11)
memory usage: 3.5+ KB

** How many people made the purchase during the AM and how many people made the purchase during PM ? **

*(Hint: Check out value_counts()) *

[ ] ecom["AM or PM"].value_counts()
PM 5068
AM 4932
Name: AM or PM, dtype: int64

** What are the 5 most common Job Titles? **

[ ] ecom["Job"].value_counts()
Interior and spatial designer 31
Lawyer 30
Social researcher 28
Purchasing manager 27
Designer, jewellery 27
Name: Job, dtype: int64

** Someone made a purchase that came from Lot: "90 WT" , what was the Purchase Price for this transaction? **

[ ]
ecom[ecom["Lot"]=="90 WT"]['Purchase Price']

513 75.1
Name: Purchase Price, dtype: float64

** What is the email of the person with the following Credit Card Number: 4926535242672853 **

[ ] ecom[ecom["Credit Card"]==4926535242672853]['Email']
1234 bondellen@williams-garza.com
Name: Email, dtype: object

* *How many people have American Express as their Credit Card Provider *and* made a purchase above $95 ?**

[ ]
```
ecom[(ecom["CC Provider"]=="American Express") & (ecom["Purchase Price"]>95)].count()
```

```
Address            39
Lot                39
AM or PM           39
Browser Info       39
Company            39
Credit Card        39
CC Exp Date        39
CC Security Code   3
CC Provider        39
Email              39
Job                39
IP Address         39
Language           39
Purchase Price     39
dtype: int64
```

** Hard: How many people have a credit card that expires in 2025? **

[ ]
```
sum(ecom["CC Exp Date"].apply(lambda x:x[3:])=="25")
```
```
1033
```

** Hard: What are the top 5 most popular email providers/hosts (e.g. gmail.com, yahoo.com, etc..) **

[ ]
```
ecom['Email'].apply(lambda x: x.split('@')[1]).value_counts().head(5)
```
```
hotmail.com     1638
yahoo.com       1616
gmail.com       1605
smith.com         42
williams.com      37
Name: Email, dtype: int64
```

RESULT :

Output obtained successfully.