# Python Training

A **basic** overview

# Introduction to pandas

- **Python supports multiple packages for data analysis**
  - **Numpy**
    - **Useful for arranging data in n dimensional matrix**
  - **Pandas**
    - Makes importing and analysing data much easier. Pandas builds on packages like NumPy and matplotlib
  - **MatplotLib**

  - **Theona**
  - **Tensorflow**
  - **Keras**
  - **Scikit_learn**
  - **Scipy**
    - **Useful for statistical calculations**

# pandas

- Integrated data manipulation and analysis capabilities
- Integration with data visualization libraries
- Built in time-series capabilities
- Optimized for speed
- Built-in support for grabbing data from multiple sources
  - csv, xls, html tables, yahoo, google, worldbank, FRED

# Data types in Pandas

- Series
  - Series is a one-dimensional labelled array capable of holding any data type
  - Like integers, strings, floating point numbers, Python objects, etc.

  e.g class pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)

  - data
    - array-like, dict, or scalar value
    - Contains data stored in Series

- index :
  - array-like I dimensional array used for labelling rows
  - Values must be hashable and have the same length as data.
  - by 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 : numpy.dtype or None
If None, dtype will be inferred

copy : boolean, default False
Copy input data

- To create a series of size 5 random numbers and rows are indexed with given labels and not numbers.

```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e’])
```

```
In [4]: s
Out[4]:
a    0.4691
b   -0.2829
c   -1.5091
d   -1.1356
e    1.2121
dtype: float64
```

- Slicing is possible on series also

```
s[:3]
a    0.4691
b   -0.2829
c   -1.5091
dtype: float64
```

- To print only indexes

```
s.index
```

Output :

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object’)
```

- If index is not given the rows are labelled as numbers by default

```
pd.Series(np.random.randn(5))
```

Output :

```
0   -0.1732
1    0.1192
2   -1.0442
3   -0.8618
4   -2.1046
dtype: float64
```

# Dictionary as series

```
d = {'b' : 1, 'a' : 0, 'c' : 2}

pd.Series(d)
```

Output

```
b    1
a    0
c    2
dtype: int64
```

## Series from scalar values

```
 pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

```
s[s > s.median()]
Out[15]:
a    0.4691
e    1.2121
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]:
e    1.2121
d   -1.1356
b   -0.2829
dtype: float64

In [17]: np.exp(s)
Out[17]:
a    1.5986
b    0.7536
c    0.2211
d    0.3212
e    3.3606
dtype: float64
```

# Setting values by index label

A Series is like a fixed-size dict in that you can get and set values by index label:

```
s['a']
0.469112299907186 28

s['e'] = 12.

 s
Output
a     0.4691
b    -0.2829
c    -1.5091
d    -1.1356
e    12.0000
dtype: float64
```

## To check whether index exists

'e' in s
True

'f' in s
False

If a label is not contained, an exception is raised:
s['f']
KeyError: 'f’

Using the get method, a missing label will return None or specified default:

s.get('f’)

- If not found display np.nan value can be used. One may change it.

s.get('f', np.nan)
nan

Memberwise addition

s + s

Multiplication

s * 2

Exponential

np.exp(s)

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
s[1:] + s[:-1]
Output
a      NaN
b   -0.5657
c   -3.0181
d   -2.2713
e      NaN
dtype: float64
```

The result of an operation between unaligned Series will have the union of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing NaN.

# Importing Data with Pandas

- To use pandas in code

  >>>import pandas as pd
- To read data from file to Data_Frame
  - mydata=pd.read_table("http://bit.ly/chiporders")
  - mymoviedata=pd.read_table("http://bit.ly/movieusers" ,sep="|",header=None)
  - mydata1=pd.read_csv("http://bit.ly/uforeports")
- To check the data type use type function
  - Print(type(mydata)) --------- o/p will be dataframe
- prints the first N rows of a DataFrame. By default 5.
  - pd.DataFrame.head() or pd.DataFrame.head(12)
- prints last N rows of a DataFrame. By default 5.
  - pd.DataFrame.tail() or pd.DataFrame.tail(12)

# Importing Data with Pandas

- Pandas to find how many rows are there
  - len(mydataframe.index)
- pandas.DataFrame.shape property to see number of rows and columns
  - myframe.shape ---------displays (rows ,column)
- pandas.DataFrame.iloc method.
  - The iloc method allows us to retrieve rows and columns by position. In order to do that, we'll need to specify the positions of the rows that we want, and the positions of the columns that we want as well.
  - myframe.iloc[0:,5:]  -------will print 5rows and all columns
- If col1 contains data abc|pqr|xyz u want to create another column
- myframe[‘newcol'] = myframe[‘col1'].str.split('|')
- If column contains integer to convert it in string
  - myframe[[‘col1’]].astype(str)
- Myframe.head()

# Some indexing examples

- myframe.iloc[:5,:] — the first 5 rows, and all of the columns for those rows.
- myframe.iloc[:,:] — the entire DataFrame.
- myframe.iloc[5:,5:] — rows from position 5 onwards, and columns from position 5 onwards.
- myframe.iloc[:,0] — the first column, and all of the rows for the column.
- myframe.iloc[9,:] — the 10th row, and all of the columns for that row.
- remove the first column, which doesn't have any useful information:
  - reviews = reviews.iloc[:,1:]
  - reviews.head()

# Some indexing examples

- We can specify column labels in the loc method to retrieve columns by label instead of by position.

myframe.loc[:5,"score"]

- more than one column at a time by passing in a list:

reviews.loc[:5,["score", "release_year"]]

- Ways to retrieve columns from frame:

reviews.iloc[:,1] — will retrieve the second column.
reviews.loc[:,"score_phrase"] — will also retrieve the second column.
reviews["score"]  ------- easiest way

- Multiple columns:
  myframe[["score", "release_year"]]
- pandas.DataFrame.mean -------- find the mean of each numerical column in a DataFrame by default:

  - reviews.mean()
  - reviews.mean(axis=0) -------- to calculate mean of each column
  - reviews.mean(axis=1) -------- to calculate mean of each row

# Some more statistical functions

1.pandas.DataFrame.corr — finds the correlation between columns in a DataFrame.
2.pandas.DataFrame.count — counts the number of non-null values in each DataFrame column.
3.pandas.DataFrame.max — finds the highest value in each column.
4.pandas.DataFrame.min — finds the lowest value in each column.
5.pandas.DataFrame.median — finds the median of each column.
6.pandas.DataFrame.std — finds the standard deviation of each column.

# DataFrame Math with Pandas

- mydataframe.[“new column”]=mydataframe.str.split(“|”)
- Mydataframe.head()

- Mydataframe[‘mycnt’]=mydataframe[“new column”].map(lambda x: len(x))

- To filter data
- data.loc[(data["Gender"]=="Female") & (data["Education"]=="Not Graduate") & (data["Loan_Status"]=="Y"), ["Gender","Education","Loan_Status"]]

# DataFrame Math with Pandas

- Abc|pqr|xyz

[Abc,pqr,xyz]

- mydataframe[“new column”]=mydataframe[[hobbies]].str.split(“|”)
- Mydataframe.head()

- Mydataframe[“mycnt”]=mydataframe[“new column”].map(lambda x: len(x))
- def myfunction(val):

  return val*10
- Mydataframe[“mycnt”]=mydataframe[“new column”].appy(myfunction)

- To filter data
- data.loc[(data["Gender"]=="Female") & (data["Education"]=="Not Graduate") & (data["Loan_Status"]=="Y"), ["Gender","Education","Loan_Status"]]

```
df_row = pd.concat([df1, df2])

df_row
```

- To avoid duplicate index
- df_row_reindex = pd.concat([df1, df2], ignore_index=True)

- df_row_reindex

- To join the frame based on id

```
df_merge_col = pd.merge(df_row, df3, on='id')

df_merge_col
```

```
dummy_data1 = {
        'id': ['1', '2', '3', '4', '5'],
        'Feature1': ['A', 'C', 'E', 'G', 'I'],
        'Feature2': ['B', 'D', 'F', 'H', 'J']}

df1 = pd.DataFrame(dummy_data1, columns = ['id', 'Feature1', 'Feature2'])
df1
```

```
dummy_data2 = {
        'id': ['1', '2', '6', '7', '8'],
        'Feature1': ['K', 'M', 'O', 'Q', 'S'],
        'Feature2': ['L', 'N', 'P', 'R', 'T']}
df2 = pd.DataFrame(dummy_data2, columns = ['id', 'Feature1', 'Feature2'])

Df2
Df1[‘dept’]=‘sales’
Df2[‘dept’]=‘Purchase’
Combined_fr=Pd.concat([Df1,Df2],ignore_index=true)
```

```
dummy_data3 = {
      'id': ['1', '2', '3', '4', '5', '7', '8', '9', '10', '11'],
      'Feature3': [12, 13, 14, 15, 16, 17, 15, 12, 13, 23]}
df3 = pd.DataFrame(dummy_data3, columns = ['id', 'Feature3'])

df3
```

If join column name is same in both the frames

```
df_merge_difkey = pd.merge(df_row, df3, on='id’)
```

---

It might happen that the column on which you want to merge the DataFrames have different names (unlike in this case). For such merges, you will have to specify the arguments left_on as the left DataFrame name and right_on as the right DataFrame name, like :

```
df_merge_difkey = pd.merge(df_row, df3, left_on='id', right_on=‘empid')
df_merge_difkey
```

---

Merging based on index

```
merged = df1.merge(df2, left_index=True, right_index=True, how='inner’)
merged = pd.merge(left=df, left_index=True , right=df_annon, right_index=True,
```

# Full Outer Join

The FULL OUTER JOIN combines the results of both the left and the right outer joins. The joined DataFrame will contain all records from both the DataFrames and fill in NaNs for missing matches on either side. You can perform a full outer join by specifying the how argument as outer in the merge() function:

```
df_outer = pd.merge(df1, df2, on='id', how='outer')
```

```
df_outer
```

Possible values of how are ------inner, left, right, outer

# Group_by

```
gr_ob=movies_df.groupby('release_year’)
gr_ob.first()
gr_ob.get_group(2018)


profits_year = movies_df.groupby('release_year')['profit'].sum()
```

# Finding first n largest

To find first n largest
DataFrame.nlargest(n, columns, keep='first’)

To display first 5 rows
Sf=movies_df.sort_values(by = ['budget'], ascending=False)
sf.head()

# Converting data

- Convert data from int to string

df['A'].apply(str)

```
# Prepare data
df['year'] = [d.year for d in df.date]
df['month'] = [d.strftime('%b') for d in df.date]
years = df['year'].unique()
```

# To convert data into numbers

df['DataFrame Column'] = pd.to_numeric(df['DataFrame Column'], errors='coerce')
By setting errors=’coerce’, you’ll transform the non-numeric values into NaN.

```
import pandas as pd

Data = {'Product': ['AAA','BBB','CCC'],
        'Price': ['210','250','22XYZ']}

df = pd.DataFrame(Data)
df['Price'] = pd.to_numeric(df['Price'],errors='coerce')

print (df)
print (df.dtypes)
```

# To Replace ‘NaN’ with ‘0’

- Use  df.replace:

```
import pandas as pd
import numpy as np

Data = {'Product': ['AAA','BBB','CCC'],
        'Price': ['210','250','22XYZ']}

df = pd.DataFrame(Data)
df['Price'] = pd.to_numeric(df['Price'],errors='coerce')
df = df.replace(np.nan, 0, regex=True)
df['Price'] = df['Price'].astype(int)

print (df)
print (df.dtypes)
```

# Dropping Rows And Columns In pandas Dataframe

```
Import modules
import pandas as pd
#Create a dataframe
data = {'name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [4, 24, 31, 2, 3]}
df = pd.DataFrame(data, index = ['Cochice', 'Pima', 'Santa Cruz', 'Maricopa',
'Yuma'])
df
```

Drop an observation (row)

df.drop(['Cochice', 'Pima’])

| name | reports | year | | |
|---|---|---|---|---|
| Santa Cruz | | Tina | 31 | 2013 |
| Maricopa | Jake | 2 | 2014 | |
| Yuma | Amy | 3 | 2014 | |

Drop a variable (column)
Note: axis=1 denotes that we are referring to a column, not a row

df.drop('reports', axis=1)

| name | year | |
|---|---|---|
| **Cochice** | Jason | 2012 |
| **Pima** | Molly | 2012 |
| **Santa Cruz** | Tina | 2013 |
| **Maricopa** | Jake | 2014 |
| **Yuma** | Amy | 2014 |

Drop a row if it contains a certain value (in this case, “Tina”)
Specifically: Create a new dataframe called df that includes all rows where the value of a cell in the name column does not equal “Tina”

df[df.name != 'Tina’]

| name | reports | year | |
|---|---|---|---|
| **Cochice** | Jason | 4 | 2012 |
| **Pima** | Molly | 24 | 2012 |
| **Maricopa** | Jake | 2 | 2014 |
| **Yuma** | Amy | 3 | 2014 |

Drop a row by row number (in this case, row 3)
Note that Pandas uses zero based numbering, so 0 is the first row, 1 is the second row, etc.

df.drop(df.index[2])

| name | reports | year | |
|---|---|---|---|
| **Cochice** | Jason | 4 | 2012 |
| **Pima** | Molly | 24 | 2012 |
| **Maricopa** | Jake | 2 | 2014 |
| **Yuma** | Amy | 3 | 2014 |

can be extended to dropping a range

df.drop(df.index[[2,3]])

| name | reports | year | |
|---|---|---|---|
| **Cochice** | Jason | 4 | 2012 |
| **Pima** | Molly | 24 | 2012 |
| **Yuma** | Amy | 3 | 2014 |

or dropping relative to the end of the DF.

df.drop(df.index[-2])

| name | reports | year | |
|---|---|---|---|
| **Cochice** | Jason | 4 | 2012 |
| **Pima** | Molly | 24 | 2012 |
| **Santa Cruz** | Tina | 31 | 2013 |
| **Yuma** | Amy | 3 | 2014 |

you can select ranges relative to the top or drop relative to the bottom of the DF as well.

df[:3] #keep top 3

| name | reports | year | |
|---|---|---|---|
| **Cochice** | Jason | 4 | 2012 |
| **Pima** | Molly | 24 | 2012 |
| **Santa Cruz** | Tina | 31 | 2013 |

df[:-3] #drop bottom 3

| name | reports | year | |
|---|---|---|---|
| **Cochice** | Jason | 4 | 2012 |
| **Pima** | Molly | 24 | 2012 |

# Sampling using pandas

```
import pandas as pd

Online_Retail=pd.read_csv(" Retail_data.csv")
Online_Retail.shape
(541909, 8)

sample_data=Online_Retail.sample(n=1000,replace="False")
sample_data.shape
(1000, 8)
```

```
# sort - ascending order
from pandas import DataFrame

Cars = {'Brand': ['Honda Civic','Toyota Corolla','Ford Focus','Audi A4'],
        'Price': [22000,25000,27000,35000],
        'Year': [2015,2013,2018,2018]
        }

df = DataFrame(Cars, columns= ['Brand', 'Price','Year'])

# sort Brand - ascending order
df.sort_values(by=['Brand'], inplace=True)

print (df)

#for sorting in       descending

#df.sort_values(by=['Brand'], inplace=True, ascending=False)

df.sort_values(by=['Year','Brand'], inplace=True)  #multiple columns
```

modDfObj = dfObj.append({'Name' : 'Sahil' , 'Age' : 22} , ignore_index=**True**)

return a new dataframe containing copy of contents of existing dataframe and with a new row appended at it’s end