Lists are handy, flexible, mutable, contain different data type in same list

Numpy - Focused on performance, Comes with built-in mathematical functions and array operations. Good for large amount of data.

pandas - High performance mathematical computation and array operation. Allows mixed Data types. Access to values using integer position or index.

In Python, we need some kind of rectangular data structure. 2D Numpy array is not necessarily the best option. Pandas is great at handling data having different data types. Each Row has a label and each column has a label as well.

Pandas is a high level data manipulation tool, built on Numpy.

Pandas can be made from dictionary { }. Each dictionary has keys and columns. Here keys are column labels. Values are data, column by column.

dict = { key\_name : [ , , , ] , key\_name\_2 : [ , , ,] }

dictionary can be converted to Data frame using **pd.DataFrame( )**

Also, you can directly import the data.

csv - comma separated values

Data frames is a collection of Columns and Rows. Unlike a matrix, data frame can have different data types for each column.

**pd.read\_csv( ) - The link inside should be in inverted comma**

To not read the row indexes (first column ) we use **index\_col = 0**

1st column is always of rows index. It depends upon us , to delete the column or rename the column.

df\_**name.index = list\_name** - changes the df row index to list name.

**Selecting, Indexing and Slicing in Pandas**

It is important that the rows and columns are given labels. This, is important to make accessing columns, rows and single elements in Dataframe easy.

df\_name.column\_name - Fetch columns by dot

* Square Brackets
* loc and iloc

**Column Access using Square brackets**

df\_name [ Column name] - Python prints out the column. But the accessed column is not a data frame but a Panda Series.

To select the columnbut keep it as data frame we use double square brackets. '[[ ]]'

df\_name [[ Column name ]] - Python prints and retains the data type.

We can select a lot of columns by separating it with commas.

**Rows Access using Square Brackets**

can be done by specifing the index

df\_name [ Row ]

Square brackets have limited functionality. We want something similar to Numpy array.

loc - selection based on labels

iloc - selection based on integer position

**loc Function**

Here we use [[ ]] to keep it as a data frame. Otherwise it gets converted to object.

In loc, selection is done by specifying the rows and column labels in inverted commas.

Multiple row selection by separating commas.

df\_name.loc [[ Row names], [Column Names]]

Select All Rows and specific Columns

df\_name.loc [ : , [Column Names]]

**iloc Functions**

Sub-setting Pandas based on their positions, we need to use iloc.

**iloc uses single square brackets and Retains Data Frames.**

**Row with column name has no index.**

**Similarly, Column with Row Name has no index.**

We can use all the Numpy array function of selecting loc in 'iloc'.

We can use ':' or [ ] to specify the rows.

When we use numbers only to specify rows or columns we use square brackets.

Lab Sessions:

Various methods that can be applied on data frames.

students\_df\*\*.\*\*Students\*\*.\*\*min()

Students is the column in students\_df

students\_df["Scores"]\*\*.\*\*describe()

Scores is the column in students\_df

students\_df\*\*.\*\*index

data.columns.get\_loc("CouncilArea")

Fetches index

students\_df\*\*.\*\*drop(3)

Drop row having index 3

Panda Series can be made.

x \*\*=\*\* pd\*\*.\*\*Series([1,2,3, 4, 5, 6, 7])

printing 'x' gives row index as well as the

Slicing can be done on x using [ ].

course\_df \*\*=\*\* pd\*\*.\*\*Series(['Programming for Analytics', 'MPBA', 507, 61])

print(course\_df[:-\*\*\*\*1])

here everything gets printed other than the index -1 element.

You can assign index to each element as well.

course\_df= pd.Series(['Programming for Analytics', 'MPBA', 507, 61], index=['Course','TCode','NCode', 'ClassSize'])

**date\_range Function**

\*pandas.date\_range(start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=None, closed=None, \**kwargs)*

dates\_days \*\*=\*\* pd\*\*.\*\*date\_range('20210101',periods\*\*=\*\*365)

***start :****Left bound for generating dates. 'YYYYMMDD' format can be used.*

***end :****Right bound for generating dates.*

***periods :****Number of periods to generate.*

***freq :****Frequency strings can have multiples, e.g. ‘5H’. See here for a list of frequency aliases.*

***tz :****Time zone name for returning localized DatetimeIndex. By default, the resulting DatetimeIndex is timezone-naive.*

N***ormalize :****Normalize start/end dates to midnight before generating date range.*

***name :****Name of the resulting DatetimeIndex.*

***closed :****Make the interval closed with respect to the given frequency to the ‘left’, ‘right’, or both sides (None, the default).*

***Returns:****DatetimeIndex*

numpy.random.randint( )

the above function is used for sampling.

stock\_price \*\*=\*\* np\*\*.\*\*random\*\*.\*\*randint(160,260,size\*\*=\*\*365)

Plot

itc\_stock\_daily\*\*.\*\*plot(kind\*\*=\*\*'line', alpha \*\*=\*\* 0.4)

nse\_data \*\*=\*\* pd\*\*.\*\*read\_excel('C:/Users/user/Documents/GitHub/AI-ML-Algorithms-for-Business-Applications/Datasets/NSE Stocks 22-Nov-2021.xlsx', skiprows\*\*=\*\* 5)

skiprows and index\_col are used.

head( ) - top of the table

tail( ) - bottom of the table

info( ) - gives the information about non-null.

replace ( old, new) - replaces old with new

rename( ) - rename column names

lower( ) - converts all letters to lower case

describe( ) - count, mean, std, min, max

isin( ) - name of rows after specifying the column

isna( ) - selects all the NA values.

notna( ) - removes all the NA values.

sum( ) -

count( ) -

agg( ) - aggregate. Can have count, sum, min, max

value\_counts ( ) - counts the frequency of each value.

sort\_values( by = 'Column\_name', ascending = False) - sort or arranges values default in ascending order.

Attributes

df\_name.shape - no. of rows and columns

df\_name.columns - Selects the 1st column of the data frame

df\_name.dtypes - Gives data types of each column

nse\_data\*\*.\*\*columns \*\*=\*\* nse\_data\*\*.\*\*columns\*\*.\*\*str\*\*.\*\*lower()\*\*.\*\*str\*\*.\*\*replace(' ','')\*\*.\*\*str\*\*.\*\*replace('/','\_')

nse\_data. columns selects the column.

str selects the text and lower converts into lower case.

replaces part of name into something else.

**Subsetting**

When selecting subsets of data, square brackets [ ] are used.

Inside these brackets, you can use a single column/row label, a list of column/row labels, a slice of labels, a conditional expression or a colon. Select specific rows and/or columns using loc when using the row and column names.

nse\_data["facevalue"]\*\*.\*\*describe()

nse\_data[nse\_data["facevalue"] \*\*>\*\* 10]

Square brackets and loc have differences in calling the row names.

nse\_data[nse\_data["companyname"]\*\*.\*\*isin(["3I Infotech Ltd.", "3M India Ltd."])]

OR

nse\_data[(nse\_data["companyname"]== "3I Infotech Ltd.")|(nse\_data["companyname"] == "3M India Ltd.")]

(condition 1 | condition 2)

**NaN stands for 'Not a Number'**

nse\_data[nse\_data["p\_b"]\*\*.\*\*notna()]

column p\_b notna values are printed.

nse\_data\*\*.\*\*loc[nse\_data["closingprice"] \*\*>\*\* 10000, ["companyname", "closingprice" ]]

Selective printing columns based on condition

df\_name.loc [ condition 1 , [ column 1, column 2]]

Why condition is written inside nse\_data[ ]

**Pandas Operations on Tabular Data**

nse\_data= nse\_data.rename(

columns= {

"companyname" : "company",

"openingprice" : "open",

"highprice" : "high",

"lowprice" : "low",

"closingprice" : "close",

"adjustedclosingprice": "close\_adj",

"marketcapitalisation" : "marketcap"

}

)

Renaming columns. Columns is a dictionary.

Creating Columns by logical operators

nse\_data["total\_stocks"] \*\*=\*\* (nse\_data["marketcap"])\*\*/\*\*(nse\_data["close\_adj"])

Separate Categorical Column, the most efficient method is np.select

[np.select](http://np.select)(Set\_of\_Conditions, values\_as\_per\_conditions)

conditions= [

(nse\_data['marketcap']<= 5000),

(nse\_data['marketcap']> 5000)& (nse\_data['marketcap']<= 20000),

(nse\_data['marketcap']> 20000),

(nse\_data['marketcap'].isna())# for companies with invalid marketcap values]

nse\_data['company\_type'] = np.select(conditions, cap\_values)

nse\_data[{ ' ', ' ' , ' ' }] - the columns you want to print or are interested in

**nse\_data[ condition ] - Used everytime**

**Group\_by( )**

grouping based on column names.

groupby happens in this pattern:

* Split the data into groups
* Apply a function to each group independently
* Combine the results into a data structure

nse\_data\*\*.\*\*groupby(['company\_type'])\*\*.\*\*describe()

group by company type. Apply Describe function. Combine the results.

nse\_data\*\*.\*\*groupby(['company\_type'])\*\*.\*\*describe()[{'marketcap', 'close\_adj'}]

group by company type. Apply describe function. Apply to only specify columns.

[Joins](https://www.notion.so/Joins-047ec47543b94a6580be720fdadc5150)

[Time Series](https://www.notion.so/Time-Series-13099c00ed2947d4bfa61134941341a4)

[Text](https://www.notion.so/Text-6a634a484fad488bbe1fed099f40f8dc)

changing values of column

if w['female'] =='female':

w['female'] = '1';

else:

w['female'] = '0';

replace values of a column

students\_df['Rank']\*\*.\*\*replace(to\_replace \*\*=\*\* ["rank 1", "rank 5", "rank 2"], value \*\*=\*\* ["Rank 1", "Rank 5", "Rank 2"] )

\*\*.\*\*split(" ") - Split where there is a space.

data = data[(data["type"] != "closed")] - Remove row with closed.

astype - converts data types.

df = df.drop('column\_name', 1)

where 1 is the *axis* number (0 for rows and 1 for columns.)

data[['wikipedia', 'search']] = data["wikipedia\_link"].apply(lambda x:pd.Series(str(x).split("wiki/")))

lambda is used to apply it to each and every row. Split is to split it into two columns.

**Joins**

Joins : left, right, inner, & outer

Concatenate two data frames into one.

pd.concat(objs = [ p2\_df, p1\_df])

class\_df \*\*=\*\* pd\*\*.\*\*concat(objs \*\*=\*\* [p2\_df, p1\_df], ignore\_index\*\*=True\*\*)

ignore\_index deletes the index the data frame has. It posts

band\_members\*\*.\*\*merge(band\_instruments, how \*\*=\*\* "left", on \*\*=\*\* "name")

band members is to the left

band instruments is to the right

inner join - intersection

outer join - union

band\_members\*\*.\*\*merge(band\_instruments, how \*\*=\*\* "outer", on \*\*=\*\* "name", indicator \*\*=\*\* \*\*True\*\*)

Indicator tells how each row is merged.

band\_members\*\*.\*\*merge(band\_instruments, how \*\*=\*\* "outer", left\_on \*\*=\*\* "name", right\_on \*\*=\*\* "name", indicator \*\*=\*\* \*\*True\*\*)

**Time Series**

pandas\_datareader

datetime

last\_day\_1 \*\*=\*\* dt\*\*.\*\*date(2021, 11, 30)

[dt.date](http://dt.date) - ( YYYY, MM, DD)

last\_dt \*\*=\*\* dt\*\*.\*\*datetime(year \*\*=\*\* 2021, month\*\*=\*\* 11, day \*\*=\*\* 30, hour\*\*=\*\*16, minute \*\*=\*\* 54, second\*\*=\*\*20)

dt.datetime - (YYYY, MM, DD, Hour, Minute, Second)

last\_ts \*\*=\*\* pd\*\*.\*\*Timestamp(year \*\*=\*\* 2021, month\*\*=\*\* 11, day \*\*=\*\* 30, hour\*\*=\*\*16, minute \*\*=\*\* 54, second\*\*=\*\*20)

pd.Timestamp - (YYYY, MM, DD, Hour, Minute, Second)

Timestamp output - Timestamp('2021-11-30 00:00:00')

file\_name.year - prints year

file\_name.month - prints month

file\_name.day - prints day

file\_name.hour - hour

file\_name.minute - minute

file\_name.second - second

**Convert date strings into Timestamp objects**

x \*\*=\*\* 'Nov-30-2021'

Here, x is a date string.

x\_dt \*\*=\*\* pd\*\*.\*\*to\_datetime(x)

Here x is converted to Timestamp using datetime.

Various Methods:

file\_name.day\_name( ) - gives week of the day as output.

pd.DateOffset( days = ) - x\_dt \*\*+\*\* pd\*\*.\*\*DateOffset(days\*\*=\*\*5)

ts\_df\*\*=\*\*data\*\*.\*\*DataReader("INFY.NS", 'yahoo', '20210101', '20211130')\*\*.\*\*reset\_index()

Calling Infosys data and resetting index.

ts\_df['returns'] \*\*=\*\* (ts\_df['Adj Close'] \*\*\*\* ts\_df['Adj Close']\*\*.\*\*shift(1))\*\*/\*\*ts\_df['Adj Close']\*\*.\*\*shift(1)

Here shift(1) the value in the previous row.

# calculate the mean over the trailing three elements

ts\_df.rolling(3).mean().head()

3-month moving average

Exponential Weighted

ts\_df\*\*.\*\*ewm(3)\*\*.\*\*mean()\*\*.\*\*head()

We use np.NaN to create NaN values.

pd.Series( list\_1, list\_2) - Merges series.

temp\_df\*\*.\*\*resample('1D')\*\*.\*\*mean()\*\*.\*\*ffill()

fill the 'NaN' value with forward fill i.e. the prior value.

temp\_df\*\*.\*\*resample('1D')\*\*.\*\*mean()\*\*.\*\*bfill()

fill the 'NaN' value with backward fill i.e. the next value

temp\_df\*\*.\*\*resample('1D')\*\*.\*\*mean()\*\*.\*\*interpolate()

fills it with mean of non interpolate values.

**[DataFrame.dropna](<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html#pandas.DataFrame.dropna>)**

Omit axes labels with missing values.

**OPERATIONS ON TEXT**

pandas\_datareader

datetime

last\_day\_1 \*\*=\*\* dt\*\*.\*\*date(2021, 11, 30)

[dt.date](http://dt.date) - ( YYYY, MM, DD)

last\_dt \*\*=\*\* dt\*\*.\*\*datetime(year \*\*=\*\* 2021, month\*\*=\*\* 11, day \*\*=\*\* 30, hour\*\*=\*\*16, minute \*\*=\*\* 54, second\*\*=\*\*20)

dt.datetime - (YYYY, MM, DD, Hour, Minute, Second)

last\_ts \*\*=\*\* pd\*\*.\*\*Timestamp(year \*\*=\*\* 2021, month\*\*=\*\* 11, day \*\*=\*\* 30, hour\*\*=\*\*16, minute \*\*=\*\* 54, second\*\*=\*\*20)

pd.Timestamp - (YYYY, MM, DD, Hour, Minute, Second)

Timestamp output - Timestamp('2021-11-30 00:00:00')

file\_name.year - prints year

file\_name.month - prints month

file\_name.day - prints day

file\_name.hour - hour

file\_name.minute - minute

file\_name.second - second

**Convert date strings into Timestamp objects**

x \*\*=\*\* 'Nov-30-2021'

Here, x is a date string.

x\_dt \*\*=\*\* pd\*\*.\*\*to\_datetime(x)

Here x is converted to Timestamp using datetime.

Various Methods:

file\_name.day\_name( ) - gives week of the day as output.

pd.DateOffset( days = ) - x\_dt \*\*+\*\* pd\*\*.\*\*DateOffset(days\*\*=\*\*5)

ts\_df\*\*=\*\*data\*\*.\*\*DataReader("INFY.NS", 'yahoo', '20210101', '20211130')\*\*.\*\*reset\_index()

Calling Infosys data and resetting index.

ts\_df['returns'] \*\*=\*\* (ts\_df['Adj Close'] \*\*\*\* ts\_df['Adj Close']\*\*.\*\*shift(1))\*\*/\*\*ts\_df['Adj Close']\*\*.\*\*shift(1)

Here shift(1) the value in the previous row.

# calculate the mean over the trailing three elements

ts\_df.rolling(3).mean().head()

3-month moving average

Exponential Weighted

ts\_df\*\*.\*\*ewm(3)\*\*.\*\*mean()\*\*.\*\*head()

We use np.NaN to create NaN values.

pd.Series( list\_1, list\_2) - Merges series.

temp\_df\*\*.\*\*resample('1D')\*\*.\*\*mean()\*\*.\*\*ffill()

fill the 'NaN' value with forward fill i.e. the prior value.

temp\_df\*\*.\*\*resample('1D')\*\*.\*\*mean()\*\*.\*\*bfill()

fill the 'NaN' value with backward fill i.e. the next value

temp\_df\*\*.\*\*resample('1D')\*\*.\*\*mean()\*\*.\*\*interpolate()

fills it with mean of non interpolate values.

**[DataFrame.dropna](<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html#pandas.DataFrame.dropna>)**

Omit axes labels with missing values.