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(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.

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 - (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 - (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.