Pandas

March 30, 2022

Ejecuta esta linea de código para descargar los datos necesarios para correr el notebook

[1]:
```
!wget https://raw.githubusercontent.com/cosmolejo/dataRepo/master/DJIA_table.csv
```

```
--2022-03-17 21:48:53--
https://raw.githubusercontent.com/cosmolejo/dataRepo/master/DJIA_table.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.111.133, 185.199.109.133, 185.199.108.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.111.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 167083 (163K) [text/plain]
Saving to: 'DJIA_table.csv'

DJIA_table.csv      100%[===================>] 163.17K  --.-KB/s    in 0.04s

2022-03-17 21:48:53 (3.61 MB/s) - 'DJIA_table.csv' saved [167083/167083]
```

#Introducción a Pandas

Este notebook fue tomado de la plataforma kagle una comunidad en línea de científicos de datos y profesionales del aprendizaje automático. En ella encontrarán distintos tutoriales, bancos de datos y competencias remuneradas en ciencias de datos.

Creditos a ***ABDULLAH SAHIN***

1 Pandas Tutorial

2 Overview

Return Contents

Welcome to my Kernel! In this kernel, I show you Pandas functions and how to use pandas. Why do I this? Because everyone who's just starting out or who's a professional is using the pandas.

If you have a question or feedback, do not hesitate to write and if you **like** this kernel, please do not forget to **UPVOTE**.

3 What is the pandas?

Return Contents

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

pandas is a NumFOCUS sponsored project. This will help ensure the success of development of pandas as a world-class open-source project, and makes it possible to donate to the project.

4 Import Library

Return Contents

[2]:
```
import numpy as np # linear algebra
import pandas as pd # import in pandas

import os
```

5 Pandas Data Structure

Return Contents

Pandas has two types of data structures. These are series and dataframe.

5.0.1 Series

The series is a one-dimensional labeled array. It can accommodate any type of data in it.

[3]:
```
mySeries = pd.Series([3,-5,7,4], index=['a','b','c','d'])
type(mySeries)
```

[3]:
```
pandas.core.series.Series
```

5.0.2 DataFrame

The dataframe is a two-dimensional data structure. It contains columns.

[4]:
```
data = {'Country' : ['Belgium', 'India', 'Brazil' ],
        'Capital': ['Brussels', 'New Delhi', 'Brassilia'],
        'Population': [1234,1234,1234]}
datas = pd.DataFrame(data, columns=['Country','Capital','Population'])
print(type(data))
print(type(datas))
```

```
<class 'dict'>
<class 'pandas.core.frame.DataFrame'>
```

6 Import Library

Return Contents

With pandas, we can open CSV, Excel and SQL databases. I will show you how to use this method for CSV and Excel files only.

6.0.1 CSV(comma - separated values)

It is very easy to open and read CSV files and to overwrite the CSV file.

[5]:
```
df = pd.read_csv('DJIA_table.csv')
type(df)
# If your Python file is not in the same folder as your CSV file, you should do␣
 ↪this as follows.
# df = pd.read_csv('/home/desktop/Iris.csv')
```

[5]:
```
pandas.core.frame.DataFrame
```

6.0.2 Excel

When we want to work with Excel files, we need to type the following code.

[ ]:
```
# pd.read_excel('filename')
# pd.to_excel('dir/dataFrame.xlsx', sheet_name='Sheet1')
```

6.0.3 Others(json, SQL, table, html)

[ ]:
```
# pd.read_sql(query,connection_object) -> Reads from a SQL table/database
# pd.read_table(filename) -> From a delimited text file(like TSV)
# pd.read_json(json_string) -> Reads from a json formatted string, URL or file
# pd.read_html(url) -> Parses an html URL, string or file and extracts tables␣
 ↪to a list of dataframes
# pd.read_clipboard() -> Takes the contentes of your clipboard and passes it to␣
 ↪read_table()
# pd.DataFrame(dict) -> From a dict, keys for columns names, values for data as␣
 ↪lists
```

7 Exporting Data

Return Contents

[ ]:
```
# df.to_csv(filename) -> Writes to a CSV file
# df.to_excel(filename) -> Writes on an Excel file
# df.to_sql(table_name, connection_object) -> Writes to a SQL table
# df.to_json(filename) -> Writes to a file in JSON format
# df.to_html(filename) -> Saves as an HTML table
# df.to_clipboard() -> Writes to the clipboard
```

8 Create Test Objects

Return Contents

[ ]:
```
pd.DataFrame(np.random.rand(20,5)) # 5 columns and 20 rows of random floats
```

[ ]:

	0	1	2	3	4
0	0.396958	0.660761	0.866063	0.300435	0.233622
1	0.814769	0.186867	0.151222	0.807853	0.448494
2	0.864380	0.316361	0.761057	0.620408	0.516316
3	0.137959	0.888772	0.039402	0.979863	0.337795
4	0.686843	0.683374	0.834539	0.188544	0.461206
5	0.033928	0.813040	0.112714	0.934681	0.431551
6	0.696093	0.730487	0.989352	0.450823	0.518951
7	0.559077	0.629547	0.716996	0.399144	0.628025
8	0.866333	0.581066	0.033455	0.881053	0.487403
9	0.042143	0.420162	0.832912	0.788419	0.794355
10	0.001675	0.461602	0.719204	0.336960	0.064070
11	0.701142	0.274169	0.566095	0.025223	0.262044
12	0.859836	0.330400	0.100965	0.480195	0.522275
13	0.983663	0.432736	0.782356	0.523563	0.791688
14	0.904935	0.478968	0.425037	0.564771	0.271104
15	0.238865	0.150874	0.154631	0.984328	0.710985
16	0.133146	0.440752	0.305782	0.312823	0.020796
17	0.919955	0.951449	0.575893	0.256070	0.994723
18	0.811380	0.073797	0.903966	0.607590	0.583512
19	0.438171	0.574421	0.968110	0.464921	0.636542

9 Summarize Data

Return Contents

It's easy to get information about data with pandas. It makes it easier for us. Let's examine the existing functions one by one

9.0.1 df.info()

This Code provides detailed information about our data.

- **RangeIndex:** Specifies how many data there is.
- **Data Columns:** Specifies how many columns are found.
- **Columns:** Gives information about Columns.
- **dtypes:** It says what kind of data you have and how many of these data you have.
- **Memory Usage:** It says how much memory usage is.

Return Contents

[ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1989 entries, 0 to 1988
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       1989 non-null   object
 1   Open       1989 non-null   float64
 2   High       1989 non-null   float64
 3   Low        1989 non-null   float64
 4   Close      1989 non-null   float64
 5   Volume     1989 non-null   int64
 6   Adj Close  1989 non-null   float64
dtypes: float64(5), int64(1), object(1)
memory usage: 108.9+ KB
```

9.0.2 df.shape()

This code shows us the number of rows and columns.

Return Contents

[ ]:
```
df.shape
```

[ ]:
```
(1989, 7)
```

9.0.3 df.index

This code shows the total number of index found.

Return Contents

[ ]:
```
df.index
```

[ ]:
```
RangeIndex(start=0, stop=1989, step=1)
```

9.0.4 df.columns

This code shows all the columns contained in the data we have examined.

Return Contents

[ ]:
```
df.columns
```

[ ]:
```
Index(['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'],
dtype='object')
```

9.0.5 df.count()

This code shows us how many pieces of data are in each column.

Return Contents

[ ]:
```
df.count()
```

[ ]:
```
Date         1989
Open         1989
High         1989
Low          1989
Close        1989
Volume       1989
Adj Close    1989
dtype: int64
```

9.0.6 df.sum()

This code shows us the sum of the data in each column.

Return Contents

[ ]:
```
df.sum()
```

[ ]:
```
Date         2016-07-012016-06-302016-06-292016-06-282016-0…
Open                                          26770181.820295
High                                          26933652.010742
Low                                           26598761.206535
Close                                         26777971.154794
Volume                                           323831020000
Adj Close                                     26777971.154794
dtype: object
```

9.0.7 df.cumsum()

This code gives us cumulative sum of the data.

Return Contents

[ ]:
```
df.cumsum().head()
```

[ ]:
```
                                                Date          Open  \
0                                         2016-07-01  17924.240234
1                                  2016-07-012016-06-30  35637.000000
2                           2016-07-012016-06-302016-06-29  53093.019531
3                    2016-07-012016-06-302016-06-292016-06-28  70283.529297
4  2016-07-012016-06-302016-06-292016-06-282016-0…  87638.740235

           High           Low         Close     Volume     Adj Close
0  18002.380859  17916.910156  17949.369141   82160000  17949.369141
1  35932.990234  35628.710937  35879.359375  215190000  35879.359375
2  53637.500000  53084.730468  53574.039063  321570000  53574.039063
3  71047.220703  70275.240234  70983.759766  433760000  70983.759766
4  88402.431641  87338.320312  88124.000000  572500000  88124.000000
```

9.0.8 df.min()

This code brings us the smallest of the data.

Return Contents

```
[ ]: df.min()
```

```
[ ]: Date          2008-08-08
     Open         6547.009766
     High         6709.609863
     Low          6469.950195
     Close        6547.049805
     Volume          8410000
     Adj Close    6547.049805
     dtype: object
```

9.0.9 df.max()

This code brings up the largest among the data.

Return Contents

```
[ ]: df.max()
```

```
[ ]: Date          2016-07-01
     Open         18315.060547
     High         18351.359375
     Low          18272.560547
     Close        18312.390625
     Volume         674920000
     Adj Close    18312.390625
     dtype: object
```

9.0.10 idxmin()

This code fetches the smallest value in the data. The use on series and dataframe is different.

Return Contents

```
[ ]: print("df: ",df['Open'].idxmin())
     print("series", mySeries.idxmin())
```

```
df:  1842
series b
```

9.0.11 idxmax()

This code returns the largest value in the data.

Return Contents

[ ]:
```
print("df: ",df['Open'].idxmax())
print("series: ",mySeries.idxmax())
```

```
df:  282
series:  c
```

9.0.12 df.describe()

This Code provides basic statistical information about the data. The numerical column is based.

- **count:** vnumber of entries
- **mean:** average of entries
- **std:** standart deviation
- **min:** minimum entry
- **25%:** first quantile
- **50%:** median or second quantile
- **75%:** third quantile
- **max:** maximum entry

Return Contents

[ ]:
```
df.describe()
```

[ ]:
```
                Open          High           Low         Close        Volume  \
count    1989.000000   1989.000000   1989.000000   1989.000000  1.989000e+03
mean    13459.116048  13541.303173  13372.931728  13463.032255  1.628110e+08
std      3143.281634   3136.271725   3150.420934   3144.006996  9.392343e+07
min      6547.009766   6709.609863   6469.950195   6547.049805  8.410000e+06
25%     10907.339844  11000.980469  10824.759766  10913.379883  1.000000e+08
50%     13022.049805  13088.110352  12953.129883  13025.580078  1.351700e+08
75%     16477.699219  16550.070312  16392.769531  16478.410156  1.926000e+08
max     18315.060547  18351.359375  18272.560547  18312.390625  6.749200e+08

           Adj Close
count    1989.000000
mean    13463.032255
std      3144.006996
min      6547.049805
25%     10913.379883
50%     13025.580078
75%     16478.410156
max     18312.390625
```

9.0.13 df.mean()

This code returns the mean value for the numeric column.

Return Contents

[ ]:
```
df.mean()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: FutureWarning:
Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None')
is deprecated; in a future version this will raise TypeError.  Select only valid
columns before calling the reduction.
  """Entry point for launching an IPython kernel.
```

[ ]:
```
Open         1.345912e+04
High         1.354130e+04
Low          1.337293e+04
Close        1.346303e+04
Volume       1.628110e+08
Adj Close    1.346303e+04
dtype: float64
```

9.0.14 df.median()

This code returns median for columns with numeric values.

Return Contents

[ ]:
```
df.median()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: FutureWarning:
Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None')
is deprecated; in a future version this will raise TypeError.  Select only valid
columns before calling the reduction.
  """Entry point for launching an IPython kernel.
```

[ ]:
```
Open         1.302205e+04
High         1.308811e+04
Low          1.295313e+04
Close        1.302558e+04
Volume       1.351700e+08
Adj Close    1.302558e+04
dtype: float64
```

9.0.15 df.quantile([0.25,0.75])

This code calculates the values 0.25 and 0.75 of the columns for each column.

Return Contents

[ ]:
```
df.quantile([0.25,0.75])
```

[ ]:
```
              Open          High           Low         Close       Volume  \
0.25  10907.339844  11000.980469  10824.759766  10913.379883  100000000.0
0.75  16477.699219  16550.070312  16392.769531  16478.410156  192600000.0

          Adj Close
```

```
0.25  10913.379883
0.75  16478.410156
```

9.0.16 df.var()

This code calculates the variance value for each column with a numeric value.

Return Contents

[ ]:
```
df.var()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: FutureWarning:
Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None')
is deprecated; in a future version this will raise TypeError.  Select only valid
columns before calling the reduction.
  """Entry point for launching an IPython kernel.
```

[ ]:
```
Open         9.880219e+06
High         9.836200e+06
Low          9.925152e+06
Close        9.884780e+06
Volume       8.821610e+15
Adj Close    9.884780e+06
dtype: float64
```

9.0.17 df.std()

This code calculates the standard deviation value for each column with numeric value.

Return Contents

[ ]:
```
df.std()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: FutureWarning:
Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=None')
is deprecated; in a future version this will raise TypeError.  Select only valid
columns before calling the reduction.
  """Entry point for launching an IPython kernel.
```

[ ]:
```
Open         3.143282e+03
High         3.136272e+03
Low          3.150421e+03
Close        3.144007e+03
Volume       9.392343e+07
Adj Close    3.144007e+03
dtype: float64
```

9.0.18 df.cummax()

This code calculates the cumulative max value between the data.

Return Contents

[ ]:
```
df.cummax()
```

[ ]:
```
            Date          Open          High           Low         Close  \
0     2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
1     2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
2     2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
3     2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
4     2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
…            …             …             …             …             …
1984  2016-07-01  18315.060547  18351.359375  18272.560547  18312.390625
1985  2016-07-01  18315.060547  18351.359375  18272.560547  18312.390625
1986  2016-07-01  18315.060547  18351.359375  18272.560547  18312.390625
1987  2016-07-01  18315.060547  18351.359375  18272.560547  18312.390625
1988  2016-07-01  18315.060547  18351.359375  18272.560547  18312.390625

          Volume     Adj Close
0       82160000  17949.369141
1      133030000  17949.369141
2      133030000  17949.369141
3      133030000  17949.369141
4      138740000  17949.369141
…            …             …
1984   674920000  18312.390625
1985   674920000  18312.390625
1986   674920000  18312.390625
1987   674920000  18312.390625
1988   674920000  18312.390625

[1989 rows x 7 columns]
```

9.0.19 df.cummin()

This code returns the cumulative min value of the data.

Return Contents

[ ]:
```
df.cummin()
```

[ ]:
```
            Date          Open          High           Low         Close  \
0     2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
1     2016-06-30  17712.759766  17930.609375  17711.800781  17929.990234
2     2016-06-29  17456.019531  17704.509766  17456.019531  17694.679688
3     2016-06-28  17190.509766  17409.720703  17190.509766  17409.720703
4     2016-06-27  17190.509766  17355.210938  17063.080078  17140.240234
…            …             …             …             …             …
1984  2008-08-14   6547.009766   6709.609863   6469.950195   6547.049805
1985  2008-08-13   6547.009766   6709.609863   6469.950195   6547.049805
```

```
1986  2008-08-12   6547.009766   6709.609863   6469.950195   6547.049805
1987  2008-08-11   6547.009766   6709.609863   6469.950195   6547.049805
1988  2008-08-08   6547.009766   6709.609863   6469.950195   6547.049805

         Volume     Adj Close
0      82160000  17949.369141
1      82160000  17929.990234
2      82160000  17694.679688
3      82160000  17409.720703
4      82160000  17140.240234
…          …           …
1984    8410000   6547.049805
1985    8410000   6547.049805
1986    8410000   6547.049805
1987    8410000   6547.049805
1988    8410000   6547.049805

[1989 rows x 7 columns]
```

9.0.20 df['columnName'].cumproad()

This code returns the cumulative production of the data.

Return Contents

[ ]:
```
df['Open'].cumprod().head()
```

[ ]:
```
0    1.792424e+04
1    3.174878e+08
2    5.542073e+12
3    9.527105e+16
4    1.653449e+21
Name: Open, dtype: float64
```

9.0.21 len(df)

This code gives you how many data there is.

Return Contents

[ ]:
```
len(df)
```

[ ]:
```
1989
```

9.0.22 df.isnull()

Checks for null values, returns boolean.

Return Contents

[ ]:
```
df.isnull().head()
```

[ ]:

	Date	Open	High	Low	Close	Volume	Adj Close
0	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False

9.0.23 df.corr()

it gives information about the correlation between the data.

Return Contents

[ ]:
```
df.corr()
```

[ ]:

	Open	High	Low	Close	Volume	Adj Close
Open	1.000000	0.999592	0.999436	0.998991	-0.691621	0.998991
High	0.999592	1.000000	0.999373	0.999546	-0.686997	0.999546
Low	0.999436	0.999373	1.000000	0.999595	-0.699572	0.999595
Close	0.998991	0.999546	0.999595	1.000000	-0.694281	1.000000
Volume	-0.691621	-0.686997	-0.699572	-0.694281	1.000000	-0.694281
Adj Close	0.998991	0.999546	0.999595	1.000000	-0.694281	1.000000

10 Selection & Filtering

Return Contents

This is how we can choose the data we want with pandas, how we can bring unique data.

10.0.1 mySeries['b']

This code returns data with a value of B in series.

Return Contents

[ ]:
```
mySeries['b']
```

[ ]:
```
-5
```

10.0.2 df[n:n]

This code fetches data from N to N.

Return Contents

[ ]:
```
df[1982:]
#Or
#df[5:7]
```

[ ]:

	Date	Open	High	Low	Close	\
1982	2008-08-18	11659.650391	11690.429688	11434.120117	11479.389648	
1983	2008-08-15	11611.209961	11709.889648	11599.730469	11659.900391	
1984	2008-08-14	11532.070312	11718.280273	11450.889648	11615.929688	
1985	2008-08-13	11632.809570	11633.780273	11453.339844	11532.959961	
1986	2008-08-12	11781.700195	11782.349609	11601.519531	11642.469727	
1987	2008-08-11	11729.669922	11867.110352	11675.530273	11782.349609	
1988	2008-08-08	11432.089844	11759.959961	11388.040039	11734.320312	

	Volume	Adj Close
1982	156290000	11479.389648
1983	215040000	11659.900391
1984	159790000	11615.929688
1985	182550000	11532.959961
1986	173590000	11642.469727
1987	183190000	11782.349609
1988	212830000	11734.320312

10.0.3 df.iloc[[n],[n]]

This code brings the data in the N row and N column in the DataFrame.

Return Contents

[ ]:
```
df.iloc[[0],[3]]
```

[ ]:

	Low
0	17916.910156

10.0.4 df.loc[n:n]

This code allows us to fetch the data in the range we specify.

Return Contents

[ ]:
```
#df.loc[n:]
# OR
df.loc[5:7]
```

[ ]:

	Date	Open	High	Low	Close	\
5	2016-06-24	17946.630859	17946.630859	17356.339844	17400.750000	
6	2016-06-23	17844.109375	18011.070312	17844.109375	18011.070312	
7	2016-06-22	17832.669922	17920.160156	17770.359375	17780.830078	

	Volume	Adj Close
5	239000000	17400.750000
6	98070000	18011.070312
7	89440000	17780.830078

10.0.5 df['columnName']

With this code, we can select and bring any column we want.

Return Contents

[ ]:
```
df['Open'].head()
# OR
# df.Open
```

[ ]:
```
0    17924.240234
1    17712.759766
2    17456.019531
3    17190.509766
4    17355.210938
Name: Open, dtype: float64
```

10.0.6 df['columnName'][n]

With this code, we can select and return any value of the column we want.

Return Contents

[ ]:
```
df['Open'][0]
# OR
# df.Open[0]
# df["Open"][1]
# df.loc[1,["Open"]]
```

[ ]:
```
17924.240234
```

10.0.7 df['columnName'].nunique()

This code shows how many of the data that is in the selected column and does not repeat.

Return Contents

[ ]:
```
df['Open'].nunique()
```

[ ]:
```
1980
```

10.0.8 df['columnName'].unique()

This code shows which of the data in the selected column repeats.

Return Contents

[ ]:
```
df['Open'].unique()
# We can write the above code as follows:: df.Open.unique()
```

```
[ ]: array([17924.240234, 17712.759766, 17456.019531, …, 11781.700195,
            11729.669922, 11432.089844])
```

10.0.9 df.columnName

This code is another way to select the column we want.

Return Contents

```
[ ]: df.Open.head()
```

```
[ ]: 0    17924.240234
     1    17712.759766
     2    17456.019531
     3    17190.509766
     4    17355.210938
     Name: Open, dtype: float64
```

10.0.10 df['columnName'].value_counts(dropna =False)

This code counts all of the data in the column we have specified, but does not count the null/none values.

Return Contents

```
[ ]: print(df.Open.value_counts(dropna =True).head())
     # OR
     # print(df['Item'].value_counts(dropna =False))
```

```
12266.750000    2
17374.779297    2
18033.330078    2
10309.389648    2
10780.000000    2
Name: Open, dtype: int64
```

10.0.11 df.head(n)

This code optionally brings in the first 5 data. returns the number of data that you type instead of N.

Return Contents

```
[ ]: df.head()
     # OR
     # df.head(15)
```

```
[ ]:          Date          Open          High           Low         Close  \
     0  2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
     1  2016-06-30  17712.759766  17930.609375  17711.800781  17929.990234
```

```
2  2016-06-29  17456.019531  17704.509766  17456.019531  17694.679688
3  2016-06-28  17190.509766  17409.720703  17190.509766  17409.720703
4  2016-06-27  17355.210938  17355.210938  17063.080078  17140.240234

      Volume     Adj Close
0   82160000  17949.369141
1  133030000  17929.990234
2  106380000  17694.679688
3  112190000  17409.720703
4  138740000  17140.240234
```

10.0.12 df.tail(n)

This code optionally brings 5 data at the end. returns the number of data that you type instead of N.

Return Contents

[ ]:
```
df.tail()
# OR
# df.tail(20)
```

[ ]:
```
            Date          Open          High           Low         Close  \
1984  2008-08-14  11532.070312  11718.280273  11450.889648  11615.929688
1985  2008-08-13  11632.809570  11633.780273  11453.339844  11532.959961
1986  2008-08-12  11781.700195  11782.349609  11601.519531  11642.469727
1987  2008-08-11  11729.669922  11867.110352  11675.530273  11782.349609
1988  2008-08-08  11432.089844  11759.959961  11388.040039  11734.320312

         Volume     Adj Close
1984  159790000  11615.929688
1985  182550000  11532.959961
1986  173590000  11642.469727
1987  183190000  11782.349609
1988  212830000  11734.320312
```

10.0.13 df.sample(n)

This code fetches random n data from the data.

Return Contents

[ ]:
```
df.sample(5)
```

[ ]:
```
            Date          Open          High           Low         Close  \
1018  2012-06-14  12497.889648  12698.679688  12497.660156  12651.910156
347   2015-02-17  18019.800781  18052.009766  17951.410156  18047.580078
1980  2008-08-20  11345.940430  11454.150391  11290.580078  11417.429688
1231  2011-08-10  11228.000000  11228.000000  10686.490234  10719.940430
```

```
596   2014-02-20  16044.150391  16161.639648  16006.589844  16133.230469

         Volume     Adj Close
1018  128640000  12651.910156
347    98760000  18047.580078
1980  144880000  11417.429688
1231  396300000  10719.940430
596    77720000  16133.230469
```

10.0.14 df.sample(frac=0.5)

This code selects the fractions of random rows and fetches the data to that extent.

Return Contents

```
[ ]: df.sample(frac=0.5).head()
```

```
[ ]:            Date          Open          High           Low         Close  \
1770  2009-06-22   8538.519531   8538.830078   8334.549805   8339.009766
918   2012-11-07  13228.240234  13228.320312  12876.599609  12932.730469
1254  2011-07-08  12717.900391  12717.900391  12567.410156  12657.200195
635   2013-12-23  16225.250000  16318.110352  16225.250000  16294.610352
278   2015-05-27  18045.080078  18190.349609  18045.080078  18162.990234

         Volume     Adj Close
1770  291240000   8339.009766
918   164250000  12932.730469
1254  131150000  12657.200195
635    78930000  16294.610352
278    96400000  18162.990234
```

10.0.15 df.nlargest(n,'columnName')

This code brings N from the column where we have specified the largest data.

Return Contents

```
[ ]: df.nlargest(5,'Open')
```

```
[ ]:            Date          Open          High           Low         Close  \
282   2015-05-20  18315.060547  18350.130859  18272.560547  18285.400391
283   2015-05-19  18300.480469  18351.359375  18261.349609  18312.390625
280   2015-05-22  18286.869141  18286.869141  18217.140625  18232.019531
281   2015-05-21  18285.869141  18314.890625  18249.900391  18285.740234
337   2015-03-03  18281.949219  18281.949219  18136.880859  18203.369141

        Volume     Adj Close
282   80190000  18285.400391
283   87200000  18312.390625
```

```
280   78890000   18232.019531
281   84270000   18285.740234
337   83830000   18203.369141
```

10.0.16 df.nsmallest(n,'columnName')

This code brings N from the column where we have specified the smallest data.

Return Contents

[ ]:
```
df.nsmallest(3,'Open')
```

[ ]:
```
            Date         Open         High          Low        Close  \
1842  2009-03-10  6547.009766  6926.490234  6546.609863  6926.490234
1844  2009-03-06  6595.160156  6755.169922  6469.950195  6626.939941
1843  2009-03-09  6625.740234  6709.609863  6516.859863  6547.049805

          Volume    Adj Close
1842  640020000  6926.490234
1844  425170000  6626.939941
1843  365990000  6547.049805
```

10.0.17 df[df.columnName < 5]

This code returns the column name we have specified, which is less than 5.

Return Contents

[ ]:
```
df[df.Open > 18281.949219]
```

[ ]:
```
            Date          Open          High           Low         Close  \
280   2015-05-22  18286.869141  18286.869141  18217.140625  18232.019531
281   2015-05-21  18285.869141  18314.890625  18249.900391  18285.740234
282   2015-05-20  18315.060547  18350.130859  18272.560547  18285.400391
283   2015-05-19  18300.480469  18351.359375  18261.349609  18312.390625

       Volume     Adj Close
280  78890000  18232.019531
281  84270000  18285.740234
282  80190000  18285.400391
283  87200000  18312.390625
```

10.0.18 df[['columnName','columnName']]

This code helps us pick and bring any columns we want.

Return Contents

[ ]:
```
df[['High','Low']].head()
# df.loc[:,["High","Low"]]
```

[ ]:

	High	Low
0	18002.380859	17916.910156
1	17930.609375	17711.800781
2	17704.509766	17456.019531
3	17409.720703	17190.509766
4	17355.210938	17063.080078

10.0.19 df.loc[:,"columnName1":"columnName2"]

This code returns columns from columnname1 to columnname2.

Return Contents

[ ]:
```
df.loc[:,"Date":"Close"].head()
# OR
# data.loc[:3,"Date":"Close"]
```

[ ]:

	Date	Open	High	Low	Close
0	2016-07-01	17924.240234	18002.380859	17916.910156	17949.369141
1	2016-06-30	17712.759766	17930.609375	17711.800781	17929.990234
2	2016-06-29	17456.019531	17704.509766	17456.019531	17694.679688
3	2016-06-28	17190.509766	17409.720703	17190.509766	17409.720703
4	2016-06-27	17355.210938	17355.210938	17063.080078	17140.240234

10.0.20 Create Filter

Return Contents

[ ]:
```
filters = df.Date > '2016-06-27'
df[filters]
```

[ ]:

	Date	Open	High	Low	Close	\
0	2016-07-01	17924.240234	18002.380859	17916.910156	17949.369141	
1	2016-06-30	17712.759766	17930.609375	17711.800781	17929.990234	
2	2016-06-29	17456.019531	17704.509766	17456.019531	17694.679688	
3	2016-06-28	17190.509766	17409.720703	17190.509766	17409.720703	

	Volume	Adj Close
0	82160000	17949.369141
1	133030000	17929.990234
2	106380000	17694.679688
3	112190000	17409.720703

10.0.21 df.filter(regex = 'code')

This code allows regex to filter any data we want.

Return Contents

[ ]:
```
df.filter(regex='^L').head()
```

[ ]:
```
            Low
0  17916.910156
1  17711.800781
2  17456.019531
3  17190.509766
4  17063.080078
```

10.0.22 np.logical_and

Filtering with logical_and. Lets look at the example.

Return Contents

[ ]:
```
df[np.logical_and(df['Open']>18281.949219, df['Date']>'2015-05-20' )]
```

[ ]:
```
           Date          Open          High           Low         Close  \
280  2015-05-22  18286.869141  18286.869141  18217.140625  18232.019531
281  2015-05-21  18285.869141  18314.890625  18249.900391  18285.740234

       Volume     Adj Close
280  78890000  18232.019531
281  84270000  18285.740234
```

10.0.23 Filtering with &

Return Contents

[ ]:
```
df[(df['Open']>18281.949219) & (df['Date']>'2015-05-20')]
```

[ ]:
```
           Date          Open          High           Low         Close  \
280  2015-05-22  18286.869141  18286.869141  18217.140625  18232.019531
281  2015-05-21  18285.869141  18314.890625  18249.900391  18285.740234

       Volume     Adj Close
280  78890000  18232.019531
281  84270000  18285.740234
```

11 Sort Data

Return Contents

11.0.1 df.sort_values('columnName')

This code sorts the column we specify in the form of low to high.

[ ]:
```
df.sort_values('Open').head()
```

[ ]:

	Date	Open	High	Low	Close	Volume	Adj Close
1842	2009-03-10	6547.009766	6926.490234	6546.609863	6926.490234	640020000	6926.490234
1844	2009-03-06	6595.160156	6755.169922	6469.950195	6626.939941	425170000	6626.939941
1843	2009-03-09	6625.740234	6709.609863	6516.859863	6547.049805	365990000	6547.049805
1846	2009-03-04	6726.500000	6979.220215	6726.419922	6875.839844	464830000	6875.839844
1847	2009-03-03	6764.810059	6855.290039	6705.629883	6726.020020	445280000	6726.020020

11.0.2 df.sort_values('columnName', ascending=False)

This code is the column we specify in the form of high to low.

Return Contents

[ ]:
```
df.sort_values('Date', ascending=False).head()
```

[ ]:

	Date	Open	High	Low	Close	Volume	Adj Close
0	2016-07-01	17924.240234	18002.380859	17916.910156	17949.369141	82160000	17949.369141
1	2016-06-30	17712.759766	17930.609375	17711.800781	17929.990234	133030000	17929.990234
2	2016-06-29	17456.019531	17704.509766	17456.019531	17694.679688	106380000	17694.679688
3	2016-06-28	17190.509766	17409.720703	17190.509766	17409.720703	112190000	17409.720703
4	2016-06-27	17355.210938	17355.210938	17063.080078	17140.240234	138740000	17140.240234

11.0.3 df.sort_index()

This code sorts from small to large according to the DataFrame index.

Return Contents

[ ]:
```
df.sort_index().head()
```

[ ]:

	Date	Open	High	Low	Close
0	2016-07-01	17924.240234	18002.380859	17916.910156	17949.369141
1	2016-06-30	17712.759766	17930.609375	17711.800781	17929.990234
2	2016-06-29	17456.019531	17704.509766	17456.019531	17694.679688
3	2016-06-28	17190.509766	17409.720703	17190.509766	17409.720703

```
4  2016-06-27  17355.210938  17355.210938  17063.080078  17140.240234

      Volume     Adj Close
0   82160000  17949.369141
1  133030000  17929.990234
2  106380000  17694.679688
3  112190000  17409.720703
4  138740000  17140.240234
```

12 Rename & Defining New & Change Columns

Return Contents

12.0.1 df.rename(columns= {'columnName' : 'newColumnName'})

This code helps us change the column name. The code I wrote below changes the ID value, but as we did not assign the change to the variable DF, it seems to be unchanged as you see below.

[ ]:
```
df.rename(columns= {'Adj Close' : 'Adjclose'}).head()
# df = df.rename(columns= {'Id' : 'Identif'}, inplace=True) -> True way
# inplace= True or False; This meaning, overwrite the data set.
# Other Way
# df.columns = ['date', 'open', 'high', 'low', 'close', 'volume', 'adjclose']
```

[ ]:
```
         Date          Open          High           Low         Close  \
0  2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
1  2016-06-30  17712.759766  17930.609375  17711.800781  17929.990234
2  2016-06-29  17456.019531  17704.509766  17456.019531  17694.679688
3  2016-06-28  17190.509766  17409.720703  17190.509766  17409.720703
4  2016-06-27  17355.210938  17355.210938  17063.080078  17140.240234

      Volume      Adjclose
0   82160000  17949.369141
1  133030000  17929.990234
2  106380000  17694.679688
3  112190000  17409.720703
4  138740000  17140.240234
```

12.0.2 Defining New Column

Create a new column

Return Contents

[ ]:
```
df["Difference"] = df.High - df.Low
df.head()
```

[ ]:
```
         Date          Open          High           Low         Close  \
0  2016-07-01  17924.240234  18002.380859  17916.910156  17949.369141
1  2016-06-30  17712.759766  17930.609375  17711.800781  17929.990234
2  2016-06-29  17456.019531  17704.509766  17456.019531  17694.679688
3  2016-06-28  17190.509766  17409.720703  17190.509766  17409.720703
4  2016-06-27  17355.210938  17355.210938  17063.080078  17140.240234

      Volume     Adj Close  Difference
0   82160000  17949.369141   85.470703
1  133030000  17929.990234  218.808594
2  106380000  17694.679688  248.490235
3  112190000  17409.720703  219.210937
4  138740000  17140.240234  292.130860
```

12.0.3 Change Index Name

Change index name to new index name

Return Contents

[ ]:
```
print(df.index.name)
df.index.name = "index_name"
df.head()
```

```
None
```

[ ]:
```
                 Date          Open          High           Low  \
index_name
0          2016-07-01  17924.240234  18002.380859  17916.910156
1          2016-06-30  17712.759766  17930.609375  17711.800781
2          2016-06-29  17456.019531  17704.509766  17456.019531
3          2016-06-28  17190.509766  17409.720703  17190.509766
4          2016-06-27  17355.210938  17355.210938  17063.080078

                  Close     Volume     Adj Close  Difference
index_name
0          17949.369141   82160000  17949.369141   85.470703
1          17929.990234  133030000  17929.990234  218.808594
2          17694.679688  106380000  17694.679688  248.490235
3          17409.720703  112190000  17409.720703  219.210937
4          17140.240234  138740000  17140.240234  292.130860
```

12.0.4 Make all columns lowercase

Return Contents

[ ]:
```
#df.columns = map(str.lower(), df.columns)
```

12.0.5 Make all columns uppercase

Return Contents

[ ]:
```
#df.columns = map(str.upper(), df.columns)
```

13 Drop Data

Return Contents

13.0.1 df.drop(columns=[‘columnName’])

This code deletes the column we have specified. But as above, I have to reset the delete to the df variable again.

[ ]:
```
df.drop(columns=['Adj Close']).head()
# df = df.drop(columns=['Id']) -> True way
# OR
# df = df.drop('col', axis=1)
# axis = 1 is meaning delete columns
# axis = 0 is meaning delete rows
```

[ ]:
index_name	Date	Open	High	Low	Close	Volume	Difference
0	2016-07-01	17924.240234	18002.380859	17916.910156	17949.369141	82160000	85.470703
1	2016-06-30	17712.759766	17930.609375	17711.800781	17929.990234	133030000	218.808594
2	2016-06-29	17456.019531	17704.509766	17456.019531	17694.679688	106380000	248.490235
3	2016-06-28	17190.509766	17409.720703	17190.509766	17409.720703	112190000	219.210937
4	2016-06-27	17355.210938	17355.210938	17063.080078	17140.240234	138740000	292.130860

13.0.2 mySeries.drop([‘a’])

This code allows us to delete the value specified in the series.

Return Contents

[ ]:
```
mySeries.drop(['a'])
```

[ ]:
```
b   -5
c    7
d    4
```

```
dtype: int64
```

13.0.3 Drop an observation (row)

Return Contents

[ ]:
```
# df.drop(['2016-07-01', '2016-06-27'])
```

13.0.4 Drop a variable (column)

Return Contents

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

[ ]:
```
# df.drop('Volume', axis=1)
```

14 Convert Data Types

Return Contents

14.0.1 df.dtypes

This code shows what data type of columns are. Boolean, int, float, object(String), date and categorical.

[ ]:
```
df.dtypes
```

[ ]:
```
Date           object
Open          float64
High          float64
Low           float64
Close         float64
Volume          int64
Adj Close     float64
Difference    float64
dtype: object
```

14.0.2 df['columnName'] = df['columnName'].astype('dataType')

This code convert the column we specify into the data type we specify.

Return Contents

[ ]:
```
df.Date.astype('category').dtypes
# OR Convert Datetime
# df.Date= pd.to_datetime(df.Date)
```

[ ]:
```
CategoricalDtype(categories=['2008-08-08', '2008-08-11', '2008-08-12',
'2008-08-13',
```

```
                    '2008-08-14', '2008-08-15', '2008-08-18', '2008-08-19',
                    '2008-08-20', '2008-08-21',
                    …
                    '2016-06-20', '2016-06-21', '2016-06-22', '2016-06-23',
                    '2016-06-24', '2016-06-27', '2016-06-28', '2016-06-29',
                    '2016-06-30', '2016-07-01'],
, ordered=False)
```

14.0.3 pd.melt(frame=dataFrameName,id_vars = 'columnName', value_vars= ['columnName'])

This code is confusing, so lets look at the example.

Return Contents

[ ]:
```
df_new = df.head()
melted = pd.melt(frame=df_new,id_vars = 'Date', value_vars= ['Low'])
melted
```

[ ]:
```
         Date variable         value
0  2016-07-01      Low  17916.910156
1  2016-06-30      Low  17711.800781
2  2016-06-29      Low  17456.019531
3  2016-06-28      Low  17190.509766
4  2016-06-27      Low  17063.080078
```

15 Apply Function

Return Contents

15.0.1 Method 1

[ ]:
```
def examples(x):   #create a function
    return x*2

df.Open.apply(examples).head()  #use the function with apply()
```

[ ]:
```
index_name
0    35848.480468
1    35425.519532
2    34912.039062
3    34381.019532
4    34710.421876
Name: Open, dtype: float64
```

15.0.2 Method 2

[ ]:
```
df.Open.apply(lambda x: x*2).head()
```

[ ]:
```
index_name
0    35848.480468
1    35425.519532
2    34912.039062
3    34381.019532
4    34710.421876
Name: Open, dtype: float64
```

16 Utilities Code

Return Contents

[ ]:
```
# pd.get_option OR pd.set_option
# pd.reset_option("^display")

# pd.reset_option("display.max_rows")
# pd.get_option("display.max_rows")
# pd.set_option("max_r",102)                  -> specifies the maximum number of␣
 ↪rows to display.
# pd.options.display.max_rows = 999           -> specifies the maximum number of␣
 ↪rows to display.

# pd.get_option("display.max_columns")
# pd.options.display.max_columns = 999        -> specifies the maximum number of␣
 ↪columns to display.

# pd.set_option('display.width', 300)

# pd.set_option('display.max_columns', 300)   -> specifies the maximum number of␣
 ↪rows to display.
# pd.set_option('display.max_colwidth', 500) -> specifies the maximum number of␣
 ↪columns to display.

# pd.get_option('max_colwidth')
# pd.set_option('max_colwidth',40)
# pd.reset_option('max_colwidth')

# pd.get_option('max_info_columns')
# pd.set_option('max_info_columns', 11)
# pd.reset_option('max_info_columns')

# pd.get_option('max_info_rows')
# pd.set_option('max_info_rows', 11)
```

```
# pd.reset_option('max_info_rows')

# pd.set_option('precision',7) -> sets the output display precision in terms of␣
 ↪decimal places. This is only a suggestion.
# OR
# pd.set_option('display.precision',3)

# pd.set_option('chop_threshold', 0) -> sets at what level pandas rounds to␣
 ↪zero when it displays a Series of DataFrame. This setting does not change␣
 ↪the precision at which the number is stored.
# pd.reset_option('chop_threshold')
```