

Installing pandas



```
edureka@ubuntu:~$ sudo apt-get install python-pandas  
[sudo] password for edureka:
```

pandas

- [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
- It is built on top of NumPy, SciPy, to some extent matplotlib
- [pandas is well suited for many different kinds of data:](#)
 - » Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
 - » Ordered and unordered (not necessarily fixed-frequency) time series data
 - » Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
 - » Any other form of observational/statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

pandas

→ Purpose

- » Analyzing data
- » Cleaning data
- » Munging data
- » Modeling data
- » Organizing the results of the analysis into a form which is suitable for plotting or tabular display



Refer - <https://pypi.python.org/pypi/pandas/0.13.0/>

pandas - Data Structure Types

→ Primary Data Structures of pandas:

- » Series (1-dimensional) – It is one-dimensional labelled array capable of holding any data type
- » The basic method to create a Series is to call:

```
>>> import pandas  
  
>>> dataFrame = pandas.Series(data = d, index=index) ←
```

- » DataFrame (2-dimensional) – It is a 2-dimensional labelled data structure with columns of potentially different types capable of holding
- » The basic method to create a DataFrame is to call:

```
>>> import pandas  
  
>>> dataFrame = pandas.DataFrame(data = d, index=index) ←
```



Note: d and index can be any Dict of 1D ndarrays, lists, dicts, or Series etc

pandas - Series

→ Series

» One-dimensional array like object containing data and labels (or index)

```
>>> import pandas as pd
>>> series = pd.Series(list('98765')) ←
>>> series
0    9
1    8
2    7
3    6
4    5
dtype: object
```

```
>>> import pandas as pd
>>> series = pd.Series(tuple('abcdef')) ←
>>> series
0    a
1    b
2    c
3    d
4    e
5    f
dtype: object
```

pandas - Traversing Series

→ [Working with Index:](#)

- » Index in a series can be specified and using the index we can fetch its corresponding value

```
>>> import pandas as pd
>>> series= pd.Series([9,8,7,6], index = ['*', '**', '***', '****'])
>>> series
*      9
**     8
***    7
****   6
dtype: int64
>>> # Fetching value with index '****'
>>> series['****']
6
```

- » Multiple values can be fetched with multiple indexes

```
>>> series[ ['*', '***']]
*      9
***    7
dtype: int64
```

pandas - Traversing Series (Contd.)

→ Series is like a fixed-length, ordered dictionary

```
>>> import pandas as pd
>>> series = pd.Series(range(5), index = list('xyzxy')) ←
>>> series
x    0
y    1
z    2
x    3
y    4
dtype: int64
```

→ Unlike dictionary, index items don't have to be unique

```
>>> # Fetching the values with index x
>>> series['x'] ←
x    0
x    3
dtype: int64
>>> # Fetching the value at index [x, 1]
>>> series['x'][1] ←
3
```


pandas - Incomplete Data in Series

- Data can be filtered using conditions.
- pandas can accommodate incomplete data. Incomplete data is replaced with "NaN" and "NaN" value is not an issue in arithmetic operations. Unlike in NumPy ndarray, data is automatically aligned

```
>>> import pandas as pd
>>> series = pd.Series({1:10, 2:20, 3:30}, index = [1,2,3,4]) ←
>>> series
1    10
2    20
3    30
4    NaN
dtype: float64
```

- We can use numpy-operations on data for filtering

```
>>> series * 4 ←
1    40
2    80
3   120
4    NaN
dtype: float64
```


pandas - DataFrame

→ DataFrame

- » Spreadsheet-like data structure containing an ordered collection of columns
- » Has both row and column index

→ DataFrame Creation

- » Creation with dictionary of equal-length lists

```
>>> import pandas as pd
>>> data = {'Country': ['US', 'US', 'INDIA', 'INDIA'], ←
...         'Year': [2012, 2013, 2012, 2013], ←
...         'Population': [20, 27, 30, 35] } ←
>>>
>>> dataframe = pd.DataFrame(data)
>>> dataframe
   Country  Population  Year
0       US           20  2012
1       US           27  2013
2    INDIA           30  2012
3    INDIA           35  2013

[4 rows x 3 columns]
```

pandas - DataFrame using dicts

→ Creation with dict of dicts

```
>>> import pandas as pd
>>> data = { 'US': {2012:30, 2013:35}, ←
...         'INDIA': {2012:20, 2013:27, 2014:28}} ←
...
>>> dataframe = pd.DataFrame(data)
>>> dataframe
   INDIA  US
2012    20  30
2013    27  35
2014    28 NaN
[3 rows x 2 columns]
```



→ Note: Incomplete data is filled with 'NaN' values

pandas - DataFrame

→ Columns can be retrieved as Series

- » dict notation
- » attribute notation

→ Rows can be retrieved by position or by name (using `ix` attribute)

```
>>> dataframe['Country']  
0      US  
1      US  
2    INDIA  
3    INDIA  
Name: Country, dtype: object  
>>> dataframe.describe()  
      Population      Year  
count      4.000000      4.00000  
mean      28.000000  2012.50000  
std        6.271629      0.57735  
min       20.000000  2012.00000  
25%       25.250000  2012.00000  
50%       28.500000  2012.50000  
75%       31.250000  2013.00000  
max       35.000000  2013.00000  
  
[8 rows x 2 columns]
```

pandas - DataFrame

→ New Columns can be added by computation or direct assignment

```
>>> dataframe
  Country  Population  Year
0      US          20  2012
1      US          27  2013
2  INDIA          30  2012
3  INDIA          35  2013

[4 rows x 3 columns]
>>> # Adding a new column
>>> dataframe['newCol'] = dataframe['Population'] * 2
>>> dataframe
  Country  Population  Year  newCol
0      US          20  2012      40
1      US          27  2013      54
2  INDIA          30  2012      60
3  INDIA          35  2013      70

[4 rows x 4 columns]
```

pandas - DataFrame

→ Functions on DataFrame

» We can apply functions like `sum()`, `mean()`, `max()`, `head()`, `count()`, `tail()` etc on DataFrame

```
>>> dataframe
      INDIA  US
2012     20  30
2013     27  35
2014     28 NaN
```

```
[3 rows x 2 columns]
```

```
>>> # Calling sum()
```

```
>>> dataframe.sum() ←
```

```
INDIA    75
```

```
US        65
```

```
dtype: float64
```

```
>>> # Calling mean()
```

```
>>> dataframe.mean() ← ←
```

```
INDIA    25.0
```

```
US       32.5
```

```
dtype: float64
```

pandas - DataFrame

» `describe()` functions will give the summary of DataFrame

```
>>> # Calling describe()
>>> dataframe.describe() ←
```

	INDIA	US
count	3.000000	2.000000
mean	25.000000	32.500000
std	4.358899	3.535534
min	20.000000	30.000000
25%	23.500000	31.250000
50%	27.000000	32.500000
75%	27.500000	33.750000
max	28.000000	35.000000

[8 rows x 2 columns]

pandas - Data Loading

→ pandas supports several ways to handle data loading

→ [Text file data](#)

- » `read_csv`
- » `read_table`

→ [Structured data](#) (JSON, XML, HTML)

- » It works fine with existing libraries

→ [Excel](#) (depends upon `xlrd` and `openpyxl` packages)

→ [Database](#)

- » `pandas.io.sql` module (`read_frame`)

pandas - Loading CSV data

→ `pandas.read_csv` will load the csv data

```
>>> from pandas import *  
>>> # Reading csv file named custs  
>>> data = read_csv('/home/edureka/Desktop/custs.csv') ←
```

Import pandas

```
data = pandas.read_csv("aapl_data.csv")
```

```
>>> data ←  
   ID First Name  Last Name  Age  Occupation  
0  4000001  Kristina    Chung   55      Pilot  
1  4000002    Paige     Chen   74      Teacher  
2  4000003   Sherri    Melton  34  Firefighter  
3  4000004  Gretchen     Hill  66  Computer hardware engineer  
4  4000005    Karen    Puckett  74      Lawyer  
5  4000006  Patrick     Song  42  Veterinarian  
6  4000007    Elsie   Hamilton  43      Pilot  
7  4000008    Hazel     Bender  63    Carpenter  
8  4000009  Malcolm     Wagner  39      Artist  
9  4000010  Dolores  McLaughlin  60      Writer  
  
[10 rows x 5 columns]
```

→ `Data` created above is a DataFrame

pandas - Loading CSV data

→ Using len on a DataFrame will give you the number of rows

```
>>> # Calling len()
>>> len(data) ←
10
```

→ We can get the column names using the columns property

```
>>> # Getting the column names
>>> data.columns ←
Index([u'ID', u'First Name', u'Last Name', u'Age', u'Occupation'], dtype='object')
```

pandas - Loading CSV data

→ Columns can be accessed in two ways

» The first is using the DataFrame like a dictionary with string keys. Multiple columns can be accessed by passing multiple column names

» `>>>data["Open"]`

» The second way to access columns is using the dot syntax. This only works if your column name could also be a Python variable name (i.e., no spaces), and if it doesn't collide with another DataFrame property or function name (e.g., count, sum)

» `>>>data.Open`

```
>>> data['ID']  
0    4000001  
1    4000002  
2    4000003  
3    4000004  
4    4000005  
5    4000006  
6    4000007  
7    4000008  
8    4000009  
9    4000010  
Name: ID, dtype: int64
```

pandas - Functions

→ `head()` function lists the first 5 rows as default. If we want to display first n rows. Use `.head(n)`

```
>>> data.head(2) ←
      ID First Name Last Name Age Occupation
0  4000001   Kristina   Chung   55      Pilot
1  4000002    Paige    Chen   74      Teacher

[2 rows x 5 columns]
```

→ `head()` function can be applied on columns also

```
>>> data.ID.head(2) ←
0    4000001
1    4000002
Name: ID, dtype: int64
```

→ We can use other functions like `tail()`, `max()`, `min()`, `std()`, `mean()` etc

```
>>> data.ID.max() ←
4000010
```

pandas - Accessing Rows

→ Accessing Individual Rows

- » Sometimes you need to access individual rows in your DataFrame. The `irow()` function lets you grab the *i*th row from a DataFrame (starting from 0)

```
>>> data.irow(1) ←  
ID          4000002  
First Name   Paige  
Last Name    Chen  
Age          74  
Occupation   Teacher  
Name: 1, dtype: object
```

```
>>> data.irow(2) ←  
ID          4000003  
First Name   Sherri  
Last Name    Melton  
Age          34  
Occupation   Firefighter  
Name: 2, dtype: object
```

pandas - Accessing Rows

→ Filtering

- » Selecting rows of interest from a DataFrame. In addition to strings, the dictionary syntax accepts things like this:
- » We can apply filters again on Hi_Volume to filter further if Hi_Volume gives more than one row as output. Multiple conditions can also be checked at a time

```
>>> Volume = data[ data.ID == 4000002 ] ←
>>> Volume
      ID First Name Last Name  Age Occupation
1  4000002      Paige      Chen   74     Teacher

[1 rows x 5 columns]
```

```
>>> Volume = data[ data.Occupation == 'Writer' ] ←
>>> Volume
      ID First Name  Last Name  Age Occupation
9  4000010    Dolores McLaughlin   60      Writer

[1 rows x 5 columns]
```

Reading and Writing Data

- We'll see three commonly used file formats: csv, text file, and Excel.

Reading

```
df=pd.read_csv('Data/mtcars.csv') # from csv
```

```
df=pd.read_csv('Data/mtcars.txt', sep='\t') # from text file
```

```
df=pd.read_excel('Data/mtcars.xlsx','Sheet2') # from Excel
```

reading from multiple sheets of same Excel into different data frames

```
xlsx = pd.ExcelFile('file_name.xls')
```

```
sheet1_df = pd.read_excel(xlsx, 'Sheet1')
```

```
sheet2_df = pd.read_excel(xlsx, 'Sheet2')
```



```
# writing
```

```
# index = False parameter will not write the index values, default is  
True
```

```
df.to_csv('Data/mtcars_new.csv', index=False)
```

```
df.to_csv('Data/mtcars_new.txt', sep='\t', index=False)
```

```
df.to_excel('Data/mtcars_new.xlsx', sheet_name='Sheet1', index =  
False)
```

Basic Operations

- Convert string to date series

```
pd.to_datetime(pd.Series(['2017-04-01','2017-04-02','2017-04-03']))
```

- Rename a specific column name

```
df.rename(columns={'old_columnname':'new_columnname'})
```

- Flag duplicates

```
df.duplicated()
```

- Drop duplicates

```
df = df.drop_duplicates()
```

Basic Operations

- Drop missing rows and columns having missing values

`df.dropna()`

- Replaces all missing values with 0 (or you can use any int or str)

`df.fillna(value=0)`

- Check missing value condition and return Boolean value of true or false for each cell

`pd.isnull(df)`

Viewing Data

- The Pandas dataframe comes with built-in functions to view the contained data.

Looking at the top `n` records default
`n` value is 5 if not specified

`df.head(n=2)`

Looking at the bottom `n` records

`df.tail()`

Get column names

`df.columns`

Get column datatypes

`df.dtypes`

Get dataframe index

`df.index`

Get unique values

`df[column_name].unique()`

Get values

`df.values`

Sort DataFrame

`df.sort_values(by=['Column1', 'Column2'],
ascending=[True, True])`

Viewing Data

select/view by column name

```
df[column_name]
```

select/view by row number

```
df[0:3]
```

selection by index

```
df.loc[0:3] # index 0 to 3
```

```
df.loc[0:3,['column1', 'column2']] # index 0 to 3  
for specific columns
```

selection by position

```
df.iloc[0:2] # using range, first 2 rows
```

```
df.iloc[2,3,6] # specific position
```

```
df.iloc[0:2,0:2] # first 2 rows and first 2 columns
```

Viewing Data

selection without it being in the index

Faster alternative to `iloc` to get scalar values

Transpose DataFrame

Filter DataFrame based on value condition for one column

Filter DataFrame based on a value condition on one column

Filter based on multiple conditions on multiple columns using AND operator

Filter based on multiple conditions on multiple columns using OR operator

```
print df.ix[1,1] # value from first row and first column
```

```
print df.ix[:,2] # all rows of column at 2nd position
```

```
print df.iat[1,1]
```

```
df.T
```

```
df[df['column_name'] > 7.5]
```

```
df[df['column_name'].isin(['condition_value1', 'condition_value2'])]
```

```
df[(df['column1'] > 7.5) & (df['column2'] > 3)]
```

```
df[(df['column1'] > 7.5) | (df['column2'] > 3)]
```

Merge/Join

- Pandas provide various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join merge-type operations

Viewing Data

- Concat or append operation

```
df = pd.concat([df_1, df_2])
```

```
print df_1.append(df_2)
```

- Join the two dataframes along columns

```
pd.concat([df_1, df_2], axis=1)
```

- Merge two dataframes

```
pd.merge(df_1, df_2, on='columnid')
```

Join

- Pandas offer SQL style merges as well.
- Left join two dataframes

```
print pd.merge(df_1, df_2, on='columnid', how='left')
```

- Merge while adding a suffix to duplicate column names of both table

```
print pd.merge(df_1, df_2, on='emp_id', how='left', suffixes=('_left', '_right'))
```

Join

- Inner Join - Inner join produces only the set of records that match in both Table A and Table B

```
pd.merge(df_1, df_2, on='emp_id', how='inner')
```

- Outer Join - Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null

```
pd.merge(df_1, df_2, on='emp_id', how='outer')
```

Grouping

- Grouping involves one or more of the following steps:
 - Splitting the data into groups based on some criteria
 - Applying a function to each group independently
 - Combining the results into a data structure