**Pandas Guide**

**Note**

* Created using Python-3.6.4 and Pandas-0.22.0
* CSV files can be downloaded from below link,

<https://bitbucket.org/pythondsp/pandasguide/downloads/>

**Contents:**

* [1. Pandas Basic](https://pandasguide.readthedocs.io/en/latest/Pandas/basic.html)
  + [1.1. Introduction](https://pandasguide.readthedocs.io/en/latest/Pandas/basic.html#introduction)
  + [1.2. Data structures](https://pandasguide.readthedocs.io/en/latest/Pandas/basic.html#data-structures)
* [2. Overview](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html)
  + [2.1. Reading files](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#reading-files)
  + [2.2. Data operations](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#data-operations)
  + [2.3. Groupby](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#groupby)
  + [2.4. Unstack](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#unstack)
  + [2.5. Merge](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#merge)
  + [2.6. Index](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#index)
  + [2.7. Implement using Python-CSV library](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#implement-using-python-csv-library)
* [3. Numpy](https://pandasguide.readthedocs.io/en/latest/Pandas/numpy.html)
  + [3.1. Creating Arrays](https://pandasguide.readthedocs.io/en/latest/Pandas/numpy.html#creating-arrays)
  + [3.2. Boolean indexing](https://pandasguide.readthedocs.io/en/latest/Pandas/numpy.html#boolean-indexing)
  + [3.3. Reshaping arrays](https://pandasguide.readthedocs.io/en/latest/Pandas/numpy.html#reshaping-arrays)
  + [3.4. Concatenating the data](https://pandasguide.readthedocs.io/en/latest/Pandas/numpy.html#concatenating-the-data)
* [4. Data processing](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html)
  + [4.1. Hierarchical indexing](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html#hierarchical-indexing)
  + [4.2. File operations](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html#file-operations)
  + [4.3. Merge](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html#merge)
  + [4.4. Data transformation](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html#data-transformation)
  + [4.5. Groupby and data aggregation](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html#groupby-and-data-aggregation)
* [5. Time series](https://pandasguide.readthedocs.io/en/latest/Pandas/timeseries.html)
  + [5.1. Dates and times](https://pandasguide.readthedocs.io/en/latest/Pandas/timeseries.html#dates-and-times)
  + [5.2. Application](https://pandasguide.readthedocs.io/en/latest/Pandas/timeseries.html#application)
* [6. Reading multiple files](https://pandasguide.readthedocs.io/en/latest/Pandas/babyname.html)
  + [6.1. Example: Baby names trend](https://pandasguide.readthedocs.io/en/latest/Pandas/babyname.html#example-baby-names-trend)
  + [6.2. Total boys and girls in year 1880](https://pandasguide.readthedocs.io/en/latest/Pandas/babyname.html#total-boys-and-girls-in-year-1880)
  + [6.3. pivot\_table](https://pandasguide.readthedocs.io/en/latest/Pandas/babyname.html#pivot-table)

# Pandas Basic

## 1.1. Introduction

Data processing is important part of analyzing the data, because data is not always available in desired format. Various processing are required before analyzing the data such as cleaning, restructuring or merging etc. Numpy, Scipy, Cython and Panda are the tools available in python which can be used fast processing of the data. Further, Pandas are built on the top of Numpy.

Pandas provides rich set of functions to process various types of data. Further, working with Panda is fast, easy and more expressive than other tools. Pandas provides fast data processing as Numpy along with flexible data manipulation techniques as spreadsheets and relational databases. Lastly, pandas integrates well with matplotlib library, which makes it very handy tool for analyzing the data.

**Note**

* In chapter 1, two important data structures i.e. Series and DataFrame are discussed.
* Chapter 2 shows the frequently used features of Pandas with example. And later chapters include various other information about Pandas.

## 1.2. Data structures

Pandas provides two very useful data structures to process the data i.e. Series and DataFrame, which are discussed in this section.

### 1.2.1. Series

The Series is a one-dimensional array that can store various data types, including mix data types. The row labels in a Series are called the index. Any list, tuple and dictionary can be converted in to Series using ‘series’ method as shown below,

>>>

**>>> import** **pandas** **as** **pd**

**>>>** *# converting tuple to Series*

**>>>** h = ('AA', '2012-02-01', 100, 10.2)

**>>>** s = pd.Series(h)

**>>>** type(s)

<class 'pandas.core.series.Series'>

**>>> print**(s)

0 AA

1 2012-02-01

2 100

3 10.2

dtype: object

**>>>** *# converting dict to Series*

**>>>** d = {'name' : 'IBM', 'date' : '2010-09-08', 'shares' : 100, 'price' : 10.2}

**>>>** ds = pd.Series(d)

**>>>** type(ds)

<class 'pandas.core.series.Series'>

**>>> print**(ds)

date 2010-09-08

name IBM

price 10.2

shares 100

dtype: object

Note that in the tuple-conversion, the index are set to ‘0, 1, 2 and 3’. We can provide custom index names as follows.

>>>

**>>>** f = ['FB', '2001-08-02', 90, 3.2]

**>>>** f = pd.Series(f, index = ['name', 'date', 'shares', 'price'])

**>>> print**(f)

name FB

date 2001-08-02

shares 90

price 3.2

dtype: object

**>>>** f['shares']

90

**>>>** f[0]

'FB'

>>>

Elements of the Series can be accessed using index name e.g. f[‘shares’] or f[0] in below code. Further, specific elements can be selected by providing the index in the list,

>>>

**>>>** f[['shares', 'price']]

shares 90

price 3.2

dtype: object

### 1.2.2. DataFrame

DataFrame is the widely used data structure of pandas. Note that, Series are used to work with one dimensional array, whereas DataFrame can be used with two dimensional arrays. DataFrame has two different index i.e. column-index and row-index.

The most common way to create a DataFrame is by using the dictionary of equal-length list as shown below. Further, all the spreadsheets and text files are read as DataFrame, therefore it is very important data structure of pandas.

>>>

**>>>** data = { 'name' : ['AA', 'IBM', 'GOOG'],

**...**  'date' : ['2001-12-01', '2012-02-10', '2010-04-09'],

**...**  'shares' : [100, 30, 90],

**...**  'price' : [12.3, 10.3, 32.2]

**...** }

**>>>** df = pd.DataFrame(data)

**>>>** type(df)

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

**>>>** df

date name price shares

0 2001-12-01 AA 12.3 100

1 2012-02-10 IBM 10.3 30

2 2010-04-09 GOOG 32.2 90

Additional columns can be added after defining a DataFrame as below,

>>>

**>>>** df['owner'] = 'Unknown'

**>>>** df

date name price shares owner

0 2001-12-01 AA 12.3 100 Unknown

1 2012-02-10 IBM 10.3 30 Unknown

2 2010-04-09 GOOG 32.2 90 Unknown

Currently, the row index are set to 0, 1 and 2. These can be changed using ‘index’ attribute as below,

>>>

**>>>** df.index = ['one', 'two', 'three']

**>>>** df

date name price shares owner

one 2001-12-01 AA 12.3 100 Unknown

two 2012-02-10 IBM 10.3 30 Unknown

three 2010-04-09 GOOG 32.2 90 Unknown

Further, any column of the DataFrame can be set as index using ‘set\_index()’ attribute, as shown below,

>>>

**>>>** df = df.set\_index(['name'])

**>>>** df

date price shares owner

name

AA 2001-12-01 12.3 100 Unknown

IBM 2012-02-10 10.3 30 Unknown

GOOG 2010-04-09 32.2 90 Unknown

Data can be accessed in two ways i.e. using row and column index,

>>>

**>>>** *# access data using column-index*

**>>>** df['shares']

name

AA 100

IBM 30

GOOG 90

Name: shares, dtype: int64

**>>>** *# access data by row-index*

**>>>** df.ix['AA']

date 2001-12-01

price 12.3

shares 100

owner Unknown

Name: AA, dtype: object

**>>>** *# access all rows for a column*

**>>>** df.ix[:, 'name']

0 AA

1 IBM

2 GOOG

Name: name, dtype: object

**>>>** *# access specific element from the DataFrame,*

**>>>** df.ix[0, 'shares']

100

Any column can be deleted using ‘del’ or ‘drop’ commands,

>>>

**>>> del** df['owner']

**>>>** df

date price shares

name

AA 2001-12-01 12.3 100

IBM 2012-02-10 10.3 30

GOOG 2010-04-09 32.2 90

**>>>** df.drop('shares', axis = 1)

date price

name

AA 2001-12-01 12.3

IBM 2012-02-10 10.3

GOOG 2010-04-09 32.2

# 2. Overview

In this chapter, various functionalities of pandas are shown with examples, which are explained in later chapters as well.

**Note**

CSV files can be downloaded from below link,

<https://bitbucket.org/pythondsp/pandasguide/downloads/>

## 2.1. Reading files

In this section, two data files are used i.e. ‘titles.csv’ and ‘cast.csv’. The ‘titles.csv’ file contains the list of movies with the releasing year; whereas ‘cast.csv’ file has five columns which store the title of movie, releasing year, star-casts, type(actor/actress), characters and ratings for actors, as shown below,

>>>

**>>> import** **pandas** **as** **pd**

**>>>** casts = pd.read\_csv('cast.csv', index\_col=None)

**>>>** casts.head()

title year name type character n

0 Closet Monster 2015 Buffy #1 actor Buffy 4 31.0

1 Suuri illusioni 1985 Homo $ actor Guests 22.0

2 Battle of the Sexes 2017 $hutter actor Bobby Riggs Fan 10.0

3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NaN

4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron NaN

**>>>** titles = pd.read\_csv('titles.csv', index\_col =None)

**>>>** titles.tail()

title year

49995 Rebel 1970

49996 Suzanne 1996

49997 Bomba 2013

49998 Aao Jao Ghar Tumhara 1984

49999 Mrs. Munck 1995

* **read\_csv** : read the data from the csv file.
* **index\_col = None** : there is no index i.e. first column is data
* **head()** : show only first five elements of the DataFrame
* **tail()** : show only last five elements of the DataFrame

If there is some error while reading the file due to encoding, then try for following option as well,

titles = pd.read\_csv('titles.csv', index\_col=None, encoding='utf-8')

If we simply type the name of the DataFrame (i.e. cast in below code), then it will show the first thirty and last twenty rows of the file along with complete list of columns. This can be limited using ‘set\_options’ as below. Further, at the end of the table total number of rows and columns will be displayed.

>>>

**>>>** pd.set\_option('max\_rows', 10, 'max\_columns', 10)

**>>>** titles

title year

0 The Rising Son 1990

1 The Thousand Plane Raid 1969

2 Crucea de piatra 1993

3 Country 2000

4 Gaiking II 2011

**...**  ... ...

49995 Rebel 1970

49996 Suzanne 1996

49997 Bomba 2013

49998 Aao Jao Ghar Tumhara 1984

49999 Mrs. Munck 1995

[50000 rows x 2 columns]

* **len** : ‘len’ commmand can be used to see the total number of rows in the file,

>>>

**>>>** len(titles)

50000

**Note**

head() and tail() commands can be used for remind ourselves about the header and contents of the file. These two commands will show the first and last 5 lines respectively of the file. Further, we can change the total number of lines to be displayed by these commands,

**>>>** titles.head(3)

title year

0 The Rising Son 1990

1 The Thousand Plane Raid 1969

2 Crucea de piatra 1993

## 2.2. Data operations

In this section, various useful data operations for DataFrame are shown.

### 2.2.1. Row and column selection

Any row or column of the DataFrame can be selected by passing the name of the column or rows. After selecting one from DataFrame, it becomes one-dimensional therefore it is considered as Series.

* **ix** : use ‘ix’ command to select a row from the DataFrame.

>>>

**>>>** t = titles['title']

**>>>** type(t)

<class 'pandas.core.series.Series'>

**>>>** t.head()

0 The Rising Son

1 The Thousand Plane Raid

2 Crucea de piatra

3 Country

4 Gaiking II

Name: title, dtype: object

>>>

**>>>** titles.ix[0]

title The Rising Son

year 1990

Name: 0, dtype: object

>>>

### 2.2.2. Filter Data

Data can be filtered by providing some boolean expression in DataFrame. For example, in below code, movies which released after 1985 are filtered out from the DataFrame ‘titles’ and stored in a new DataFrame i.e. after85.

>>>

**>>>** *# movies after 1985*

**>>>** after85 = titles[titles['year'] > 1985]

**>>>** after85.head()

title year

0 The Rising Son 1990

2 Crucea de piatra 1993

3 Country 2000

4 Gaiking II 2011

5 Medusa (IV) 2015

>>>

**Note**

When we pass the boolean results to DataFrame, then panda will show all the results which corresponds to True (rather than displaying True and False), as shown in above code. Further, ‘& (and)’ and ‘| (or)’ can be used for joining two conditions as shown below,\*\*

In below code all the movies in decade 1990 (i.e. 1900-1999) are selected. Also ‘t = titles’ is used for simplicity purpose only.

>>>

**>>>** *# display movie in years 1990 - 1999*

**>>>** t = titles

**>>>** movies90 = t[ (t['year']>=1990) & (t['year']<2000) ]

**>>>** movies90.head()

title year

0 The Rising Son 1990

2 Crucea de piatra 1993

12 Poka Makorer Ghar Bosoti 1996

19 Maa Durga Shakti 1999

24 Conflict of Interest 1993

>>>

### 2.2.3. Sorting

Sorting can be performed using ‘sort\_index’ or ‘sort\_values’ keywords,

>>>

**>>>** *# find all movies named as 'Macbeth'*

**>>>** t = titles

**>>>** macbeth = t[ t['title'] == 'Macbeth']

**>>>** macbeth.head()

title year

4226 Macbeth 1913

9322 Macbeth 2006

11722 Macbeth 2013

17166 Macbeth 1997

25847 Macbeth 1998

Note that in above filtering operation, the data is sorted by index i.e. by default ‘sort\_index’ operation is used as shown below,

>>>

**>>>** *# by default, sort by index i.e. row header*

**>>>** macbeth = t[ t['title'] == 'Macbeth'].sort\_index()

**>>>** macbeth.head()

title year

4226 Macbeth 1913

9322 Macbeth 2006

11722 Macbeth 2013

17166 Macbeth 1997

25847 Macbeth 1998

>>>

To sort the data by values, the ‘sort\_value’ option can be used. In below code, data is sorted by year now,

>>>

**>>>** *# sort by year*

**>>>** macbeth = t[ t['title'] == 'Macbeth'].sort\_values('year')

**>>>** macbeth.head()

title year

4226 Macbeth 1913

17166 Macbeth 1997

25847 Macbeth 1998

9322 Macbeth 2006

11722 Macbeth 2013

>>>

### 2.2.4. Null values

Note that, various columns may contains no values, which are usually filled as NaN. For example, rows 3-4 of casts are NaN as shown below,

>>>

**>>>** casts.ix[3:4]

title year name type character n

3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NaN

4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron NaN

These null values can be easily selected, unselected or contents can be replaced by any other values e.g. empty strings or 0 etc. Various examples of null values are shown in this section.

* **‘isnull’** command returns the true value if any row of has null values. Since the rows 3-4 has NaN value, therefore, these are displayed as True.

>>>

**>>>** c = casts

**>>>** c['n'].isnull().head()

0 False

1 False

2 False

3 True

4 True

Name: n, dtype: bool

* **‘notnull’** is opposite of isnull, it returns true for not null values,

>>>

**>>>** c['n'].notnull().head()

0 True

1 True

2 True

3 False

4 False

Name: n, dtype: bool

* To display the rows with null values, the condition must be passed in the DataFrame,

>>>

**>>>** c[c['n'].isnull()].head(3)

title year name type character n

3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NaN

4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron NaN

5 Straight Outta Compton 2015 $hutter actor Club Patron NaN

>>>

* NaN values can be fill by using **fillna**, **ffill(forward fill)**, and **bfill(backward fill)** etc. In below code, ‘NaN’ values are replace by NA. Further, example of ffill and bfill are shown in later part of the tutorial,

>>>

**>>>** c\_fill = c[c['n'].isnull()].fillna('NA')

**>>>** c\_fill.head(2)

title year name type character n

3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NA

4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron NA

### 2.2.5. String operations

Various string operations can be performed using **‘.str.’** option. Let’s search for the movie “Maa” first,

>>>

**>>>** t = titles

**>>>** t[t['title'] == 'Maa']

title year

38880 Maa 1968

>>>

There is only one movie in the list. Now, we want to search all the movies which starts with ‘Maa’. The ‘.str.’ option is required for such queries as shown below,

>>>

**>>>** t[t['title'].str.startswith("Maa ")].head(3)

title year

19 Maa Durga Shakti 1999

3046 Maa Aur Mamta 1970

7470 Maa Vaibhav Laxmi 1989

>>>

### 2.2.6. Count Values

Total number of occurrences can be counted using **‘value\_counts()’** option. In following code, total number of movies are displayed base on years.

>>>

**>>>** t['year'].value\_counts().head()

2016 2363

2017 2138

2015 1849

2014 1701

2013 1609

Name: year, dtype: int64

### 2.2.7. Plots

Pandas supports the matplotlib library and can be used to plot the data as well. In previous section, the total numbers of movies/year were filtered out from the DataFrame. In the below code, those values are saved in new DataFrame and then plotted using panda,

>>>

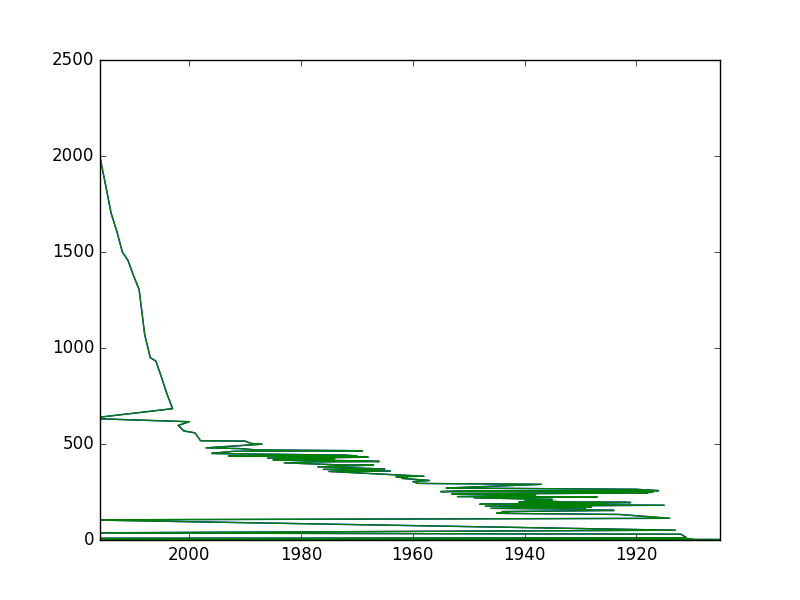
**>>> import** **matplotlib.pyplot** **as** **plt**

**>>>** t = titles

**>>>** p = t['year'].value\_counts()

**>>>** p.plot()

<matplotlib.axes.\_subplots.AxesSubplot object at 0xaf18df6c>

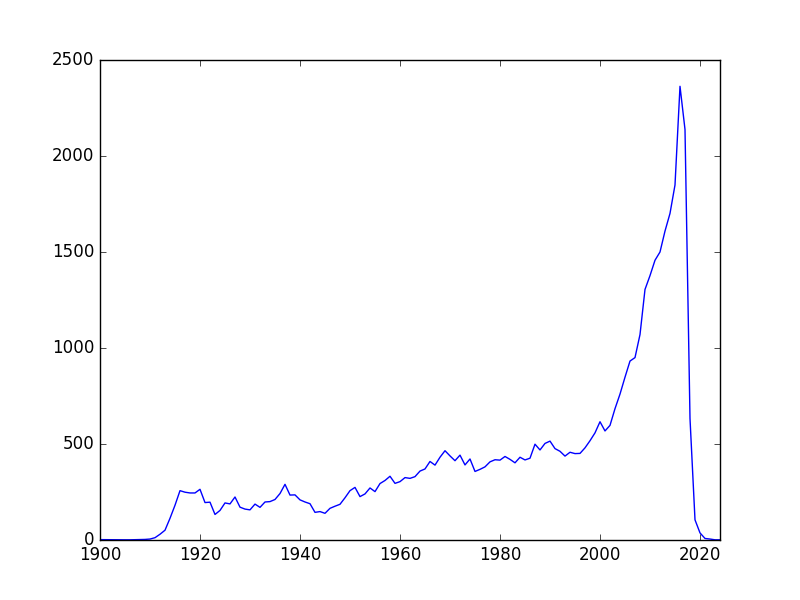
[](https://pandasguide.readthedocs.io/en/latest/_images/plot11.png)**>>>** plt.show()

Following plot will be generated from above code, which does not provide any useful information.

It’s better to sort the years (i.e. index) first and then plot the data as below. Here, the plot shows that number of movies are increasing every year.

>>>

**>>>** p.sort\_index().plot()

[](https://pandasguide.readthedocs.io/en/latest/_images/plot2.png)<matplotlib.axes.\_subplots.AxesSubplot object at 0xa9cd134c>

**>>>** plt.show()

Now, the graph provide some useful information i.e. number of movies are increasing each year.

## 2.3. Groupby

Data can be grouped by columns-headers. Further, custom formats can be defined to group the various elements of the DataFrame.

### 2.3.1. Groupby with column-names

In Section [Count Values](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#countvalues), the value of movies/year were counted using ‘count\_values()’ method. Same can be achieve by ‘groupby’ method as well. The ‘groupby’ command return an object, and we need to an additional functionality to it to get some results. For example, in below code, data is grouped by ‘year’ and then size() command is used. The **size()** option counts the total number for rows for each year; therefore the result of below code is same as ‘count\_values()’ command.

>>>

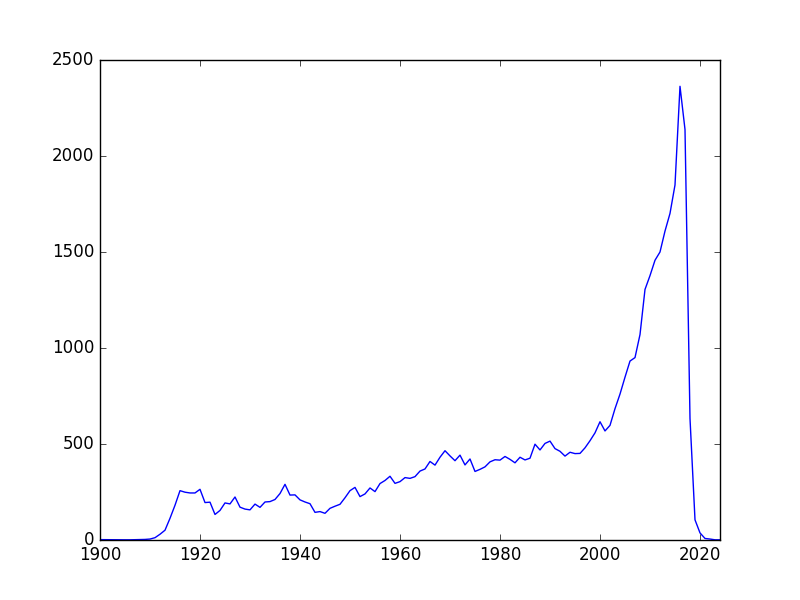
**>>>** cg = c.groupby(['year']).size()

**>>>** cg.plot()

<matplotlib.axes.\_subplots.AxesSubplot object at 0xa9f14b4c>

**>>>** plt.show()

>>>

[](https://pandasguide.readthedocs.io/en/latest/_images/plot2.png)

* Further, groupby option can take multiple parameters for grouping. For example, we want to group the movies of the actor ‘Aaron Abrams’ based on year,

>>>

**>>>** c = casts

**>>>** cf = c[c['name'] == 'Aaron Abrams']

**>>>** cf.groupby(['year']).size().head()

year

2003 2

2004 2

2005 2

2006 1

2007 2

dtype: int64

>>>

Above list shows that year-2003 is found in two rows with name-entry as ‘Aaron Abrams’. In the other word, he did 2 movies in 2003.

* Next, we want to see the list of movies as well, then we can pass two parameters in the list as shown below,

>>>

**>>>** cf.groupby(['year', 'title']).size().head()

year title

2003 The In-Laws 1

The Visual Bible: The Gospel of John 1

2004 Resident Evil: Apocalypse 1

Siblings 1

2005 Cinderella Man 1

dtype: int64

>>>

In above code, the groupby operation is performed on the ‘year’ first and then on ‘title’. In the other word, first all the movies are grouped by year. After that, the result of this groupby is again grouped based on titles. Note that, first group command arranged the year in order i.e. 2003, 2004 and 2005 etc.; then next group command arranged the title in alphabetical order.

* Next, we want to do grouping based on maximum ratings in a year; i.e. we want to group the items by year and see the maximum rating in those years,

>>>

**>>>** c.groupby(['year']).n.max().head()

year

1912 6.0

1913 14.0

1914 39.0

1915 14.0

1916 35.0

Name: n, dtype: float64

Above results show that the maximum rating in year 1912 is 6 for Aaron Abrams.

* Similarly, we can check for the minimum rating,

>>>

**>>>** c.groupby(['year']).n.min().head()

year

1912 6.0

1913 1.0

1914 1.0

1915 1.0

1916 1.0

Name: n, dtype: float64

* Lastly, we want to check the mean rating each year,

>>>

**>>>** c.groupby(['year']).n.mean().head()

year

1912 6.000000

1913 4.142857

1914 7.085106

1915 4.236111

1916 5.037736

Name: n, dtype: float64

### 2.3.2. Groupby with custom field

Suppose we want to group the data based on decades, then we need to create a custom groupby field,

>>>

**>>>** *# decade conversion : 1985//10 = 198, 198\*10 = 1980*

**>>>** decade = c['year']//10\*10

**>>>** c\_dec = c.groupby(decade).n.size()

>>>

**>>>** c\_dec.head()

year

1910 669

1920 1121

1930 3448

1940 3997

1950 3892

dtype: int64

Above results shows the total number of movies in each decade.

## 2.4. Unstack

Before understanding the unstack, let’s consider one case from cast.csv file. In following code, the data is grouped by decade and type i.e. actor and actress.

>>>

**>>>** c = casts

**>>>** c.groupby( [c['year']//10\*10, 'type'] ).size().head(8)

year type

1910 actor 384

actress 285

1920 actor 710

actress 411

1930 actor 2628

actress 820

1940 actor 3014

actress 983

dtype: int64

>>>

**Note**

Unstack is discussed in Section [Unstack the data](https://pandasguide.readthedocs.io/en/latest/Pandas/dataprocessing.html#unstackdatamorepanda) in detail.

Now we want to compare and plot the total number of actors and actresses in each decade. One solution to this problem is to grab even and odd rows separately and plot the data, which is quite complicated operation if types has more varieties e.g. new-actor, new-actress and teen-actors etc. A simple solution to such problem is the **‘unstack’**, which allows to create a new DataFrame based on the grouped Dataframe, as shown below.

* Since we want a plot based on actors and actress, therefore first we need to group the data based on ‘type’ as below,

>>>

**>>>** c = casts

**>>>** c\_decade = c.groupby( ['type', c['year']//10\*10] ).size()

**>>>** c\_decade

type year

actor 1910 384

1920 710

1930 2628

[...]

actress 1910 285

1920 411

1930 820

[...]

dtype: int64

>>>

* Now we can create a new DataFrame using ‘unstack’ command. The ‘unstack’ command creates a new DataFrame based on index,

>>>

**>>>** c\_decade.unstack()

year 1910 1920 1930 1940 1950 1960 1970 1980 1990 [...]

type

actor 384 710 2628 3014 2877 2775 3044 3565 5108 [...]

actress 285 411 820 983 1015 968 1299 1989 2544 [...]

* Use following commands to plot the above data,

>>>

**>>>** c\_decade.unstack().plot()

<matplotlib.axes.\_subplots.AxesSubplot object at 0xb1cec56c>

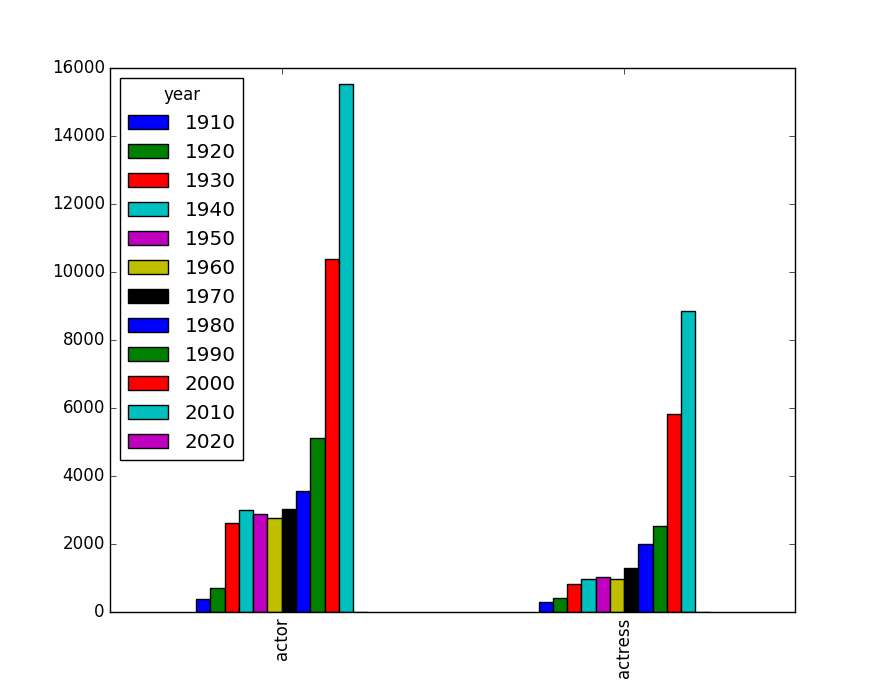
**>>>** plt.show()

**>>>** c\_decade.unstack().plot(kind='bar')

<matplotlib.axes.\_subplots.AxesSubplot object at 0xa8bf778c>

**>>>** plt.show()

Below figure will be generated from above command. Note that in the plot, actor and actress are plot separately in the groups.

[](https://pandasguide.readthedocs.io/en/latest/_images/plot4.png)

* To plot the data side by side, use unstack(0) option as shown below (by default unstack(-1) is used),

>>>

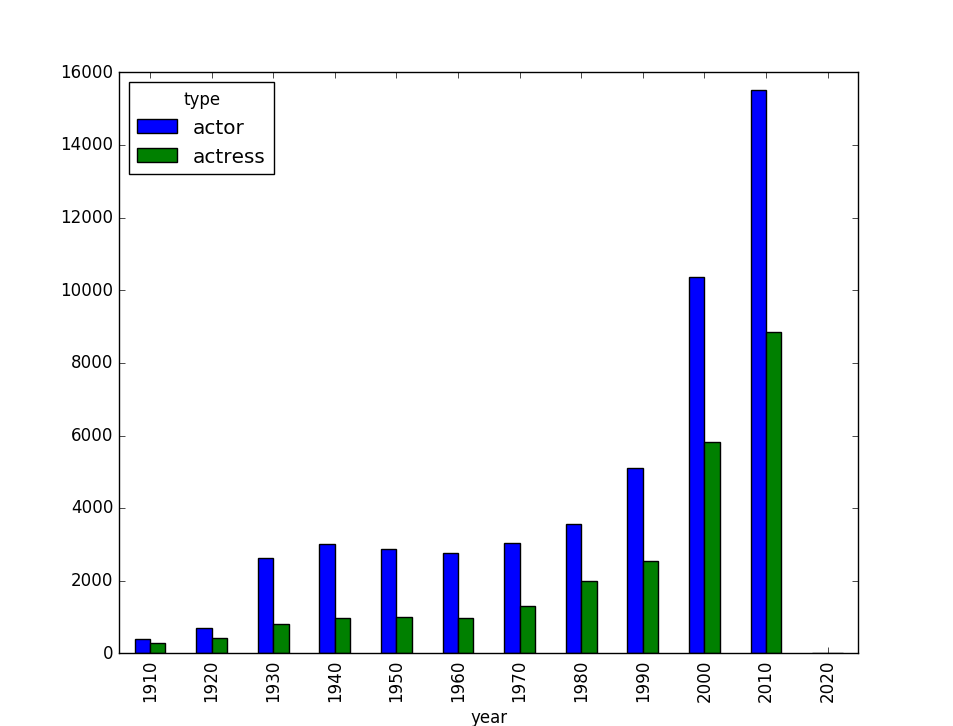
**>>>** c\_decade.unstack(0)

type actor actress

year

1910 384 285

1920 710 411

[](https://pandasguide.readthedocs.io/en/latest/_images/plot5.png)1930 2628 820

1940 3014 983

1950 2877 1015

1960 2775 968

1970 3044 1299

1980 3565 1989

1990 5108 2544

2000 10368 5831

2010 15523 8853

2020 4 3

**>>>** c\_decade.unstack(0).plot(kind='bar')

<matplotlib.axes.\_subplots.AxesSubplot object at 0xb1d218cc>

**>>>** plt.show()

## 2.5. Merge

Usually, different data of same project are available in various files. To get the useful information from these files, we need to combine these files. Also, we need to merge to different data in the same file to get some specific information. In this section, we will understand these two merges i.e. merge with different file and merge with same file.

### 2.5.1. Merge with different files

In this section, we will merge the data of two table i.e. ‘release\_dates.csv’ and ‘cast.csv’. The ‘release\_dates.csv’ file contains the release date of movies in different countries.

* First, load the ‘release\_dates.csv’ file, which contains the release dates of some of the movies, listed in ‘cast.csv’. Following are the content of ‘release\_dates.csv’ file,

>>>

**>>>** release = pd.read\_csv('release\_dates.csv', index\_col=None)

**>>>** release.head()

title year country date

0 #73, Shaanthi Nivaasa 2007 India 2007-06-15

1 #Beings 2015 Romania 2015-01-29

2 #Declimax 2018 Netherlands 2018-01-21

3 #Ewankosau saranghaeyo 2015 Philippines 2015-01-21

4 #Horror 2015 USA 2015-11-20

**>>>** casts.head()

title year name type character n

0 Closet Monster 2015 Buffy #1 actor Buffy 4 31.0

1 Suuri illusioni 1985 Homo $ actor Guests 22.0

2 Battle of the Sexes 2017 $hutter actor Bobby Riggs Fan 10.0

3 Secret in Their Eyes 2015 $hutter actor 2002 Dodger Fan NaN

4 Steve Jobs 2015 $hutter actor 1988 Opera House Patron NaN

* Let’s we want to see the release date of the movie ‘Amelia’. For this first, filter out the Amelia from the DataFrame ‘cast’ as below. There are only two entries for the movie Amelia.

>>>

**>>>** c\_amelia = casts[ casts['title'] == 'Amelia']

**>>>** c\_amelia.head()

title year name type character n

5767 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0

23319 Amelia 2009 Jeremy Akerman actor Sheriff 19.0

>>>

* Next, we will see the entries of movie ‘Amelia’ in release dates as below. In the below result, we can see that there are two different release years for the movie i.e. 1966 and 2009.

>>>

**>>>** release [ release['title'] == 'Amelia' ].head()

title year country date

20543 Amelia 1966 Mexico 1966-03-10

20544 Amelia 2009 Canada 2009-10-23

20545 Amelia 2009 USA 2009-10-23

20546 Amelia 2009 Australia 2009-11-12

20547 Amelia 2009 Singapore 2009-11-12

>>>

* Since there is not entry for Amelia-1966 in casts DataFrame, therefore merge command will not merge the Amelia-1966 release dates. In following results, we can see that only Amelia 2009 release dates are merges with casts DataFrame.

>>>

**>>>** c\_amelia.merge(release).head()

title year name type character n country date

0 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Canada 2009-10-23

1 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 USA 2009-10-23

2 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Australia 2009-11-12

3 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Singapore 2009-11-12

4 Amelia 2009 Aaron Abrams actor Slim Gordon 8.0 Ireland 2009-11-13

### 2.5.2. Merge table with itself

Suppose, we want see the list of co-actors in the movies. For this, we need to merge the table with itself based on the title and year, as shown below. In the below code, co-star for actor ‘Aaron Abrams’ are displayed,

* First, filter out the results for ‘Aaron Abrams’,

>>>

**>>>** c = casts[ casts['name']=='Aaron Abrams' ]

**>>>** c.head(2)

title year name type character n

5765 #FromJennifer 2017 Aaron Abrams actor Ralph Sinclair NaN

5766 388 Arletta Avenue 2011 Aaron Abrams actor Alex 4.0

>>>

* Next, to find the co-stars, merge the DataFrame with itself based on ‘title’ and ‘year’ i.e. for being a co-star, the name of the movie and the year must be same,
* Note that ‘casts’ is used inside the bracket instead of c.

c.merge(casts, on=['title', 'year']).head()

|  | **title** | **year** | **name\_x** | **type\_x** | **character\_x** | **n\_x** | **name\_y** | **type\_y** | **character\_y** | **n\_y** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | #FromJennifer | 2017 | Aaron Abrams | actor | Ralph Sinclair | NaN | Aaron Abrams | actor | Ralph Sinclair | NaN |
| **1** | #FromJennifer | 2017 | Aaron Abrams | actor | Ralph Sinclair | NaN | Christian Ackerman | actor | Simon | NaN |
| **2** | 388 Arletta Avenue | 2011 | Aaron Abrams | actor | Alex | 4.0 | Graham Abbey | actor | Officer #2 | 8.0 |
| **3** | 388 Arletta Avenue | 2011 | Aaron Abrams | actor | Alex | 4.0 | Aaron Abrams | actor | Alex | 4.0 |
| **4** | Amelia | 2009 | Aaron Abrams | actor | Slim Gordon | 8.0 | Aaron Abrams | actor | Slim Gordon | 8.0 |

The problem with above joining is that it displays the ‘Aaron Abrams’ as his co-actor as well (see first row). This problem can be avoided as below,

c\_costar = c.merge (casts, on=['title', 'year'])

c\_costar = c\_costar[c\_costar['name\_y'] != 'Aaron Abrams']

c\_costar.head()

|  | **title** | **year** | **name\_x** | **type\_x** | **character\_x** | **n\_x** | **name\_y** | **type\_y** | **character\_y** | **n\_y** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | #FromJennifer | 2017 | Aaron Abrams | actor | Ralph Sinclair | NaN | Christian Ackerman | actor | Simon | NaN |
| **2** | 388 Arletta Avenue | 2011 | Aaron Abrams | actor | Alex | 4.0 | Graham Abbey | actor | Officer #2 | 8.0 |
| **5** | Amelia | 2009 | Aaron Abrams | actor | Slim Gordon | 8.0 | Jeremy Akerman | actor | Sheriff | 19.0 |
| **8** | Cinderella Man | 2005 | Aaron Abrams | actor | 1928 Fan | 67.0 | Nick Alachiotis | actor | Baer Cornerman | 38.0 |
| **9** | Cinderella Man | 2005 | Aaron Abrams | actor | 1928 Fan | 67.0 | Nick Alachiotis | actor | Undercard Boxer - Feldman | 38.0 |

## 2.6. Index

In the previous section, we saw some uses of index for sorting and plotting the data. In this section, index are discussed in detail.

Index is very important tool in pandas. It is used to organize the data and to provide us fast access to data. In this section, time for data-access are compared for the data with and without indexing. For this section, Jupyter notebook is used as ‘%%timeit’ is very easy to use in it to compare the time required for various access-operations.

### 2.6.1. Creating index

**import** **pandas** **as** **pd**

cast = pd.read\_csv('cast.csv', index\_col=None)

cast.head()

|  | **title** | **year** | **name** | **type** | **character** | **n** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Closet Monster | 2015 | Buffy #1 | actor | Buffy 4 | 31.0 |
| **1** | Suuri illusioni | 1985 | Homo $ | actor | Guests | 22.0 |
| **2** | Macbeth | 1916 | USA | 1916-06-04 | NaN | NaN |
| **3** | Macbeth | 1916 | Japan | 1917-02-26 | NaN | NaN |
| **4** | Macbeth | 1948 | France | 1950-06-23 | NaN | NaN |

%%time

*# data access without indexing*

cast[cast['title']=='Macbeth']

CPU times: user 8 ms, sys: 4 ms, total: 12 ms

Wall time: 13.8 ms

|  | **title** | **year** | **name** | **type** | **character** | **n** |
| --- | --- | --- | --- | --- | --- | --- |
| **2** | Macbeth | 1916 | USA | 1916-06-04 | NaN | NaN |
| **3** | Macbeth | 1916 | Japan | 1917-02-26 | NaN | NaN |
| **4** | Macbeth | 1948 | France | 1950-06-23 | NaN | NaN |
| **5** | Macbeth | 1948 | West Germany | 1950-06-28 | NaN | NaN |
| **6** | Macbeth | 1948 | Finland | 1950-09-22 | NaN | NaN |
| **...** | ... | ... | ... | ... | ... | ... |
| **27046** | Macbeth | 2016 | John Albasiny | actor | Doctor | NaN |
| **38146** | Macbeth | 1948 | William Alland | actor | Second Murderer | 18.0 |
| **40695** | Macbeth | 1997 | Stevie Allen | actor | Murderer | 21.0 |
| **60599** | Macbeth | 2014 | Moyo Akand? | actress | Witch | NaN |
| **63832** | Macbeth | 1916 | Mary Alden | actress | Lady Macduff | 6.0 |

63 rows × 6 columns

‘%%timeit’ can be used for more precise results as it run the shell various times and display the average time; but it will not show the output of the shell,

%%timeit

*# data access without indexing*

cast[cast['title']=='Macbeth']

100 loops, best of 3: 9.85 ms per loop

**‘set\_index’** can be used to create an index for the data. Note that, in below code, ‘title’ is set at index, therefore index-numbers are replaced by ‘title’ (see the first column).

*# below line will not work for multiple index*

*# c = cast.set\_index('title')*

c = cast.set\_index(['title'])

c.head(4)

|  | **year** | **name** | **type** | **character** | **n** |
| --- | --- | --- | --- | --- | --- |
| **title** |  |  |  |  |  |
| **Closet Monster** | 2015 | Buffy #1 | actor | Buffy 4 | 31.0 |
| **Suuri illusioni** | 1985 | Homo $ | actor | Guests | 22.0 |
| **Macbeth** | 1916 | USA | 1916-06-04 | NaN | NaN |
| **Macbeth** | 1916 | Japan | 1917-02-26 | NaN | NaN |

To use the above indexing, **‘.loc’** should be used for fast operations,

%%time

*# data access with indexing*

*# note that there is minor performance improvement*

c.loc['Macbeth']

CPU times: user 36 ms, sys: 0 ns, total: 36 ms

Wall time: 36.2 ms

|  | **year** | **name** | **type** | **character** | **n** |
| --- | --- | --- | --- | --- | --- |
| **title** |  |  |  |  |  |
| **Macbeth** | 1916 | USA | 1916-06-04 | NaN | NaN |
| **Macbeth** | 1916 | Japan | 1917-02-26 | NaN | NaN |
| **Macbeth** | 1948 | France | 1950-06-23 | NaN | NaN |
| **Macbeth** | 1948 | West Germany | 1950-06-28 | NaN | NaN |
| **Macbeth** | 1948 | Finland | 1950-09-22 | NaN | NaN |
| **...** | ... | ... | ... | ... | ... |
| **Macbeth** | 2016 | John Albasiny | actor | Doctor | NaN |
| **Macbeth** | 1948 | William Alland | actor | Second Murderer | 18.0 |
| **Macbeth** | 1997 | Stevie Allen | actor | Murderer | 21.0 |
| **Macbeth** | 2014 | Moyo Akand? | actress | Witch | NaN |
| **Macbeth** | 1916 | Mary Alden | actress | Lady Macduff | 6.0 |

63 rows × 5 columns

%%timeit

*# data access with indexing*

*# note that there is minor performance improvement*

c.loc['Macbeth']

100 loops, best of 3: 5.64 ms per loop

\*\* We can see that, there is performance improvement (i.e. 11ms to 6ms) using indexing, because speed will increase further if the index are in sorted order. \*\*

Next, we will sort the index and perform the filter operation,

cs = cast.set\_index(['title']).sort\_index()

cs.tail(4)

|  | **year** | **name** | **type** | **character** | **n** |
| --- | --- | --- | --- | --- | --- |
| **title** |  |  |  |  |  |
| **xXx: Return of Xander Cage** | 2017 | Julie Abcede | actor | Catwalk Partiers | 84.0 |
| **xXx: Return of Xander Cage** | 2017 | Jeimi Abila | actress | Lazarus' Girls | 64.0 |
| **xXx: Return of Xander Cage** | 2017 | Wayne Ambrose | actor | Choir Members | 34.0 |
| **xXx: State of the Union** | 2005 | Robert Alonzo | actor | Guard | NaN |

%%time

*# data access with indexing*

*# note that there is huge performance improvement*

cs.loc['Macbeth']

CPU times: user 36 ms, sys: 0 ns, total: 36 ms

Wall time: 38.8 ms

|  | **year** | **name** | **type** | **character** | **n** |
| --- | --- | --- | --- | --- | --- |
| **title** |  |  |  |  |  |
| **Macbeth** | 2015 | Darren Adamson | actor | Soldier | NaN |
| **Macbeth** | 2015 | Estonia | 2015-12-25 | NaN | NaN |
| **Macbeth** | 1948 | Robert Alan | actor | Third Murderer | NaN |
| **Macbeth** | 1948 | West Germany | 1950-06-28 | NaN | NaN |
| **Macbeth** | 2015 | Bulgaria | 2015-12-11 | NaN | NaN |
| **...** | ... | ... | ... | ... | ... |
| **Macbeth** | 2015 | Argentina | 2015-12-17 | NaN | NaN |
| **Macbeth** | 2009 | USA | 2009-11-17 | NaN | NaN |
| **Macbeth** | 1997 | UK | 1997-05-16 | NaN | NaN |
| **Macbeth** | 2015 | Germany | 2015-10-29 | NaN | NaN |
| **Macbeth** | 1916 | Mary Alden | actress | Lady Macduff | 6.0 |

63 rows × 5 columns

Now, filtering is completing in around ‘0.5 ms’ (rather than 4 ms), as shown by below results,

%%timeit

*# data access with indexing*

*# note that there huge performance improvement*

cs.loc['Macbeth']

1000 loops, best of 3: 480 µs per loop

### 2.6.2. Multiple index

Further, we can have multiple indexes in the data,

*# data with two index i.e. title and n*

cm = cast.set\_index(['title', 'n']).sort\_index()

cm.tail(30)

|  |  | **year** | **name** | **type** | **character** |
| --- | --- | --- | --- | --- | --- |
| **title** | **n** |  |  |  |  |
| **Zwei in einem Anzug** | **2.0** | 1950 | Wolf Albach-Retty | actor | Otto Vogel |
| **Zwei in einem Auto** | **2.0** | 1951 | Wolf Albach-Retty | actor | Georg Schmittlein |
| **Zweimal zwei im Himmelbett** | **1.0** | 1937 | Georg Alexander | actor | Arnd Krusemark |
| **Zwischen Lachen und Weinen** | **NaN** | 1919 | Georg Alexander | actor | Hans |
| **Zwischen Pankow und Zehlendorf** | **NaN** | 1991 | Eugen Albert | actor | Soldat |
| **...** | **...** | ... | ... | ... | ... |
| **w Delta z** | **8.0** | 2007 | Barbara Adair | actress | Alice Jackson |
| **xXx: Return of Xander Cage** | **34.0** | 2017 | Wayne Ambrose | actor | Choir Members |
| **64.0** | 2017 | Jeimi Abila | actress | Lazarus' Girls |
| **84.0** | 2017 | Julie Abcede | actor | Catwalk Partiers |
| **xXx: State of the Union** | **NaN** | 2005 | Robert Alonzo | actor | Guard |

30 rows × 4 columns

>>>

**>>>** cm.loc['Macbeth']

year name type character

n

4.0 1916 Spottiswoode Aitken actor Duncan

6.0 1916 Mary Alden actress Lady Macduff

18.0 1948 William Alland actor Second Murderer

21.0 1997 Stevie Allen actor Murderer

NaN 2015 Darren Adamson actor Soldier

NaN 1948 Robert Alan actor Third Murderer

NaN 2016 John Albasiny actor Doctor

NaN 2014 Moyo Akand? actress Witch

In above result, ‘title’ is removed from the index list, which represents that there is one more level of index, which can be used for filtering. Lets filter the data again with second index as well,

*# show Macbeth with ranking 4-18*

cm.loc['Macbeth'].loc[4:18]

|  | **year** | **name** | **type** | **character** |
| --- | --- | --- | --- | --- |
| **n** |  |  |  |  |
| **4.0** | 1916 | Spottiswoode Aitken | actor | Duncan |
| **6.0** | 1916 | Mary Alden | actress | Lady Macduff |
| **18.0** | 1948 | William Alland | actor | Second Murderer |

If there is only one match data, then Series will return (instead of DataFrame),

*# show Macbeth with ranking 4*

cm.loc['Macbeth'].loc[4]

year 1916

name Spottiswoode Aitken

type actor

character Duncan

Name: 4.0, dtype: object

### 2.6.3. Reset index

Index can be reset using **‘reset\_index**’ command. Let’s look at the ‘cm’ DataFrame again.

cm.head(2)

|  |  | **year** | **name** | **type** | **character** |
| --- | --- | --- | --- | --- | --- |
| **title** | **n** |  |  |  |  |
| **#1 Serial Killer** | **17.0** | 2013 | Michael Alton | actor | Detective Roberts |
| **#DigitalLivesMatter** | **NaN** | 2016 | Rashan Ali | actress | News Reporter |

In ‘cm’ DataFrame, there are two index; and one of these i.e. n is removed using ‘reset\_index’ command.

*# remove 'n' from index*

cm = cm.reset\_index('n')

cm.head(2)

|  | **n** | **year** | **name** | **type** | **character** |
| --- | --- | --- | --- | --- | --- |
| **title** |  |  |  |  |  |
| **#1 Serial Killer** | 17.0 | 2013 | Michael Alton | actor | Detective Roberts |
| **#DigitalLivesMatter** | NaN | 2016 | Rashan Ali | actress | News Reporter |

## 2.7. Implement using Python-CSV library

Note that, all the above logic can be implemented using python-csv library as well. In this section, some of the logics of above sections are re-implemented using python-csv library. By looking at following examples, we can see that how easy is it to work with pandas as compare to python-csv library. However, we have more fun with python built-in libraries,

### 2.7.1. Read the file

**import** **csv**

titles = list(csv.DictReader(open('titles.csv')))

titles[0:5] *# display first 5 rows*

[OrderedDict([('title', 'The Rising Son'), ('year', '1990')]),

OrderedDict([('title', 'The Thousand Plane Raid'), ('year', '1969')]),

OrderedDict([('title', 'Crucea de piatra'), ('year', '1993')]),

OrderedDict([('title', 'Country'), ('year', '2000')]),

OrderedDict([('title', 'Gaiking II'), ('year', '2011')])]

*# display last 5 rows*

titles[-5:]

[OrderedDict([('title', 'Rebel'), ('year', '1970')]),

OrderedDict([('title', 'Suzanne'), ('year', '1996')]),

OrderedDict([('title', 'Bomba'), ('year', '2013')]),

OrderedDict([('title', 'Aao Jao Ghar Tumhara'), ('year', '1984')]),

OrderedDict([('title', 'Mrs. Munck'), ('year', '1995')])]

* Display title and year in separate row,

**for** k, v **in** titles[0].items():

**print**(k, ':', v)

title : The Rising Son

year : 1990

### 2.7.2. Display movies according to year

* Display all movies in year 1985

year85 = [a **for** a **in** titles **if** a['year'] == '1985']

year85[:5]

[OrderedDict([('title', 'Insaaf Main Karoonga'), ('year', '1985')]),

OrderedDict([('title', 'Vivre pour survivre'), ('year', '1985')]),

OrderedDict([('title', 'Water'), ('year', '1985')]),

OrderedDict([('title', 'Doea tanda mata'), ('year', '1985')]),

OrderedDict([('title', 'Koritsia gia tsibima'), ('year', '1985')])]

* Movies in years 1990 - 1999,

*# movies from 1990 to 1999*

movies90 = [m **for** m **in** titles **if** (int(m['year']) < int('2000')) **and** (int(m['year']) > int('1989'))]

movies90[:5]

[OrderedDict([('title', 'The Rising Son'), ('year', '1990')]),

OrderedDict([('title', 'Crucea de piatra'), ('year', '1993')]),

OrderedDict([('title', 'Poka Makorer Ghar Bosoti'), ('year', '1996')]),

OrderedDict([('title', 'Maa Durga Shakti'), ('year', '1999')]),

OrderedDict([('title', 'Conflict of Interest'), ('year', '1993')])]

* Find all movies ‘Macbeth’,

*# find Macbeth movies*

macbeth = [m **for** m **in** titles **if** m['title']=='Macbeth']

macbeth[:3]

[OrderedDict([('title', 'Macbeth'), ('year', '1913')]),

OrderedDict([('title', 'Macbeth'), ('year', '2006')]),

OrderedDict([('title', 'Macbeth'), ('year', '2013')])]

### 2.7.3. operator.iemgetter

* Sort movies by year,

*# sort based on year and display 3*

**from** **operator** **import** itemgetter

sorted(macbeth, key=itemgetter('year'))[:3]

[OrderedDict([('title', 'Macbeth'), ('year', '1913')]),

OrderedDict([('title', 'Macbeth'), ('year', '1997')]),

OrderedDict([('title', 'Macbeth'), ('year', '1998')])]

### 2.7.4. Replace empty string with 0

casts = list(csv.DictReader(open('cast.csv')))

casts[3:5]

[OrderedDict([('title', 'Secret in Their Eyes'),

('year', '2015'),

('name', '$hutter'),

('type', 'actor'),

('character', '2002 Dodger Fan'),

('n', '')]),

OrderedDict([('title', 'Steve Jobs'),

('year', '2015'),

('name', '$hutter'),

('type', 'actor'),

('character', '1988 Opera House Patron'),

('n', '')])]

*# replace '' with 0*

cast0 = [{\*\*c, 'n':c['n'].replace('', '0')} **for** c **in** casts]

cast0[3:5]

[{'title': 'Secret in Their Eyes',

'year': '2015', 'name': '$hutter',

'type': 'actor', 'character': '2002 Dodger Fan',

'n': '0'},

{'title': 'Steve Jobs',

'year': '2015', 'name': '$hutter',

'type': 'actor', 'character': '1988 Opera House Patron',

'n': '0'}]

* Movies starts with ‘Maa’

*# Movies starts with Maa*

maa = [m **for** m **in** titles **if** m['title'].startswith('Maa')]

maa[:3]

[OrderedDict([('title', 'Maa Durga Shakti'), ('year', '1999')]),

OrderedDict([('title', 'Maarek hob'), ('year', '2004')]),

OrderedDict([('title', 'Maa Aur Mamta'), ('year', '1970')])]

### 2.7.5. collections.Counter

* Count movies by year,

*# Most release movies*

**from** **collections** **import** Counter

by\_year = Counter(t['year'] **for** t **in** titles)

by\_year.most\_common(3)

*# by\_year.elements # to see the complete dictionary*

['1990', '1969', '1993', '2000', '2011']

* plot the data

**import** **matplotlib.pyplot** **as** **plt**

data = by\_year.most\_common(len(titles))

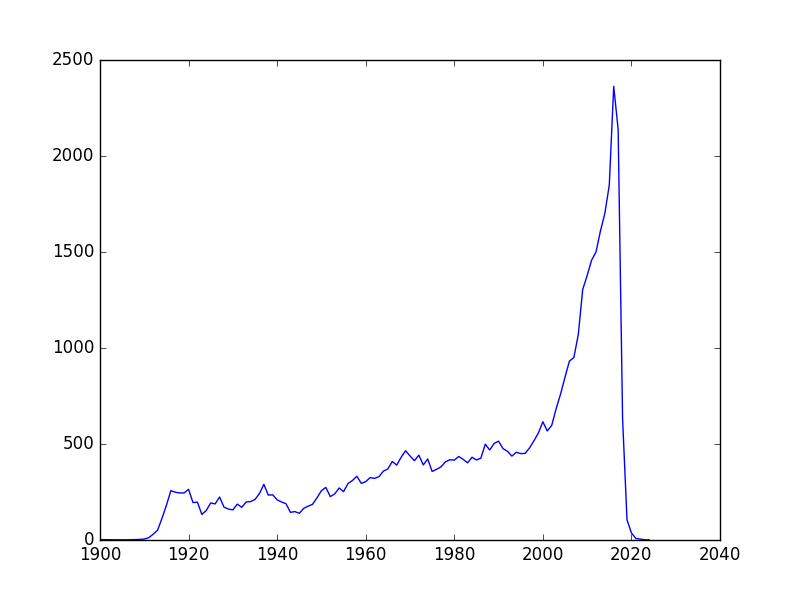
data = sorted(data) *# sort the data for proper axis*

x = [c[0] **for** c **in** data] *# extract year*

y = [c[1] **for** c **in** data] *# extract total number of movies*

plt.plot(x, y)

plt.show()

[](https://pandasguide.readthedocs.io/en/latest/_images/plot6.png)

### 2.7.6. collections.defaultdict

* append movies in dictionary by year,

**from** **collections** **import** defaultdict

d = defaultdict(list)

**for** row **in** titles:

d[row['year']].append(row['title'])

xx=[]

yy=[]

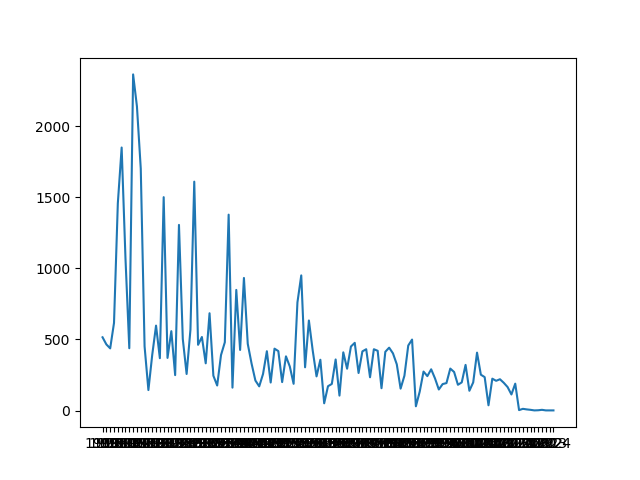
**for** k, v **in** d.items():

xx.append(k)*# = k*

yy.append(len(v))*# = len(v)*

plt.plot(sorted(xx), yy)

plt.show()

[](https://pandasguide.readthedocs.io/en/latest/_images/plot7.png)

xx[:5] *# display content of xx*

['1976', '1964', '1914', '1934', '1952']

yy[:5] *# display content of yy*

[515, 465, 437, 616, 1457]

* show all movies of Aaron Abrams

*# show all movies of Aaron Abrams*

cf = [c **for** c **in** casts **if** c['name']=='Aaron Abrams']

cf[:3]

[OrderedDict([('title', '#FromJennifer'), ('year', '2017'),

('name', 'Aaron Abrams'), ('type', 'actor'),

('character', 'Ralph Sinclair'), ('n', '')]),

OrderedDict([('title', '388 Arletta Avenue'), ('year', '2011'),

('name', 'Aaron Abrams'), ('type', 'actor'),

('character', 'Alex'), ('n', '4')]),

OrderedDict([('title', 'Amelia'), ('year', '2009'),

('name', 'Aaron Abrams'), ('type', 'actor'),

('character', 'Slim Gordon'), ('n', '8')])]

* Collect all movies of Aaron Abrams by year,

*# Display movies of Aaron Abrams by year*

dcf = defaultdict(list)

**for** row **in** cf:

dcf[row['year']].append(row['title'])

dcf

defaultdict(<class 'list'>, {

'2017': ['#FromJennifer', 'The Go-Getters'],

'2011': ['388 Arletta Avenue', 'Jesus Henry Christ', 'Jesus Henry Christ', 'Take This Waltz', 'The Chicago 8'], '2009': ['Amelia', 'At Home by Myself... with You'],

'2005': ['Cinderella Man', 'Sabah'],

'2015': ['Closet Monster', 'Regression'],

'2018': ['Code 8'], '2007': ['Firehouse Dog', 'Young People Fucking'],

'2008': ['Flash of Genius'], '2013': ['It Was You Charlie'],

'2004': ['Resident Evil: Apocalypse', 'Siblings'],

'2003': ['The In-Laws', 'The Visual Bible: The Gospel of John'],

'2006': ['Zoom']})

# 3. Numpy

Numerical Python (Numpy) is used for performing various numerical computation in python. Calculations using Numpy arrays are faster than the normal python array. Further, pandas are build over numpy array, therefore better understanding of python can help us to use pandas more effectively.

## 3.1. Creating Arrays

Defining multidimensional arrays are very easy in numpy as shown in below examples,

>>>

**>>> import** **numpy** **as** **np**

**>>>** *# 1-D array*

**>>>** d = np.array([1, 2, 3])

**>>>** type(d)

<class 'numpy.ndarray'>

**>>>** d

array([1, 2, 3])

>>>

**>>>** *# multi dimensional array*

**>>>** nd = np.array([[1, 2, 3], [3, 4, 5], [10, 11, 12]])

**>>>** type(nd)

<class 'numpy.ndarray'>

**>>>** nd

array([[ 1, 2, 3],

[ 3, 4, 5],

[10, 11, 12]])

**>>>** nd.shape *# shape of array*

(3, 3)

**>>>** nd.dtype *# data type*

dtype('int32')

>>>

**>>>** *# define zero matrix*

**>>>** np.zeros(3)

array([ 0., 0., 0.])

**>>>** np.zeros([3, 2])

array([[ 0., 0.],

[ 0., 0.],

[ 0., 0.]])

**>>>** *# diagonal matrix*

**>>>** e = np.eye(3)

array([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]])

**>>>** *# add 2 to e*

**>>>** e2 = e + 2

**>>>** e2

array([[ 3., 2., 2.],

[ 2., 3., 2.],

[ 2., 2., 3.]])

**>>>** *# create matrix with all entries as 1 and size as 'e2'*

**>>>** o = np.ones\_like(e2)

**>>>** o

array([[ 1., 1., 1.],

[ 1., 1., 1.],

[ 1., 1., 1.]])

**>>>** *# changing data type*

**>>>** o = np.ones\_like(e2)

**>>>** o.dtype

dtype('float64')

**>>>** oi = o.astype(np.int32)

**>>>** oi

array([[1, 1, 1],

[1, 1, 1],

[1, 1, 1]])

**>>>** oi.dtype

dtype('int32')

>>>

**>>>** *# convert string-list to float*

**>>>** a = ['1', '2', '3']

**>>>** a\_arr = np.array(a, dtype=np.string\_) *# convert list to ndarray*

**>>>** af = a\_arr.astype(float) *# change ndarray type*

**>>>** af

array([ 1., 2., 3.])

**>>>** af.dtype

dtype('float64')

## 3.2. Boolean indexing

Boolean indexing is very important feature of numpy, which is frequently used in pandas,

>>>

**>>>** *# accessing data with boolean indexing*

**>>>** data = np.random.randn(5, 3)

**>>>** data

array([[ 0.96174001, 1.49352768, -0.31277422],

[ 0.25044202, 2.35367396, 0.5697222 ],

[-1.21536074, 0.82088599, -1.85503026],

[-1.31492648, 1.24546252, 0.27972961],

[ 0.23487862, -0.20627825, 0.41470205]])

**>>>** name = np.array(['a', 'b', 'c', 'a', 'b'])

**>>>** name=='a'

array([ True, False, False, True, False], dtype=bool)

**>>>** data[name=='a']

array([[ 0.96174001, 1.49352768, -0.31277422],

[-1.31492648, 1.24546252, 0.27972961]])

**>>>** data[name != 'a']

array([[ 0.25044202, 2.35367396, 0.5697222 ],

[-1.21536074, 0.82088599, -1.85503026],

[ 0.23487862, -0.20627825, 0.41470205]])

**>>>** data[(name == 'b') | (name=='c')]

array([[ 0.25044202, 2.35367396, 0.5697222 ],

[-1.21536074, 0.82088599, -1.85503026],

[ 0.23487862, -0.20627825, 0.41470205]])

**>>>** data[ (data > 1) & (data < 2) ]

array([ 1.49352768, 1.24546252])

## 3.3. Reshaping arrays

>>>

**>>>** a = np.arange(0, 20)

**>>>** a

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,

17, 18, 19])

**>>>** *# reshape array a*

**>>>** a45 = a.reshape(4, 5)

**>>>** a45

array([[ 0, 1, 2, 3, 4],

[ 5, 6, 7, 8, 9],

[10, 11, 12, 13, 14],

[15, 16, 17, 18, 19]])

**>>>** *# select row 2, 0 and 1 from a45 and store in b*

**>>>** b = a45[ [2, 0, 1] ]

**>>>** b

array([[10, 11, 12, 13, 14],

[ 0, 1, 2, 3, 4],

[ 5, 6, 7, 8, 9]])

**>>>** *# transpose array b*

**>>>** b.T

array([[10, 0, 5],

[11, 1, 6],

[12, 2, 7],

[13, 3, 8],

[14, 4, 9]])

## 3.4. Concatenating the data

We can combine the data to two arrays using ‘concatenate’ command,

>>>

**>>>** arr = np.arange(12).reshape(3,4)

**>>>** rn = np.random.randn(3, 4)

**>>>** arr

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

**>>>** rn

array([[-0.25178434, 0.98443663, -0.99723191, -0.64737102],

[ 1.29179768, -0.88437251, -1.25608884, -1.60265896],

[-0.60085171, 0.8569506 , 0.62657649, 1.43647342]])

**>>>** *# merge data of rn below the arr*

**>>>** np.concatenate([arr, rn])

array([[ 0. , 1. , 2. , 3. ],

[ 4. , 5. , 6. , 7. ],

[ 8. , 9. , 10. , 11. ],

[ -0.25178434, 0.98443663, -0.99723191, -0.64737102],

[ 1.29179768, -0.88437251, -1.25608884, -1.60265896],

[ -0.60085171, 0.8569506 , 0.62657649, 1.43647342]])

**>>>** *# merge dataof rn on the right side of the arr*

**>>>** np.concatenate([arr, rn], axis=1)

array([[ 0. , 1. , 2. , 3. ,

-0.25178434, 0.98443663, -0.99723191, -0.64737102],

[ 4. , 5. , 6. , 7. ,

1.29179768, -0.88437251, -1.25608884, -1.60265896],

[ 8. , 9. , 10. , 11. ,

-0.60085171, 0.8569506 , 0.62657649, 1.43647342]])

>>>

# 4. Data processing

Most of programming work in data analysis and modeling is spent on data preparation e.g. loading, cleaning and rearranging the data etc. Pandas along with python libraries gives us provide us a high performance, flexible and high level environment for processing the data.

In chapter 1, we saw basics of pandas; then various examples are shown in chapter 2 for better understanding of pandas; whereas chapter 3 presented some basics of numpy. In this chapter, we will see some more functionality of pandas to process the data effectively.

## 4.1. Hierarchical indexing

Hierarchical indexing is an important feature of pandas that enable us to have multiple index levels. We already see an example of it in Section [Multiple index](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#multipleindex). In this section, we will learn more about indexing and access to data with these indexing.

### 4.1.1. Creating multiple index

* Following is an example of series with multiple index,

>>>

**>>> import** **pandas** **as** **pd**

**>>>** data = pd.Series([10, 20, 30, 40, 15, 25, 35, 25], index = [['a', 'a',

**...** 'a', 'a', 'b', 'b', 'b', 'b'], ['obj1', 'obj2', 'obj3', 'obj4', 'obj1',

**...** 'obj2', 'obj3', 'obj4']])

**>>>** data

a obj1 10

obj2 20

obj3 30

obj4 40

b obj1 15

obj2 25

obj3 35

obj4 25

dtype: int64

* There are two level of index here i.e. (a, b) and (obj1, …, obj4). The index can be seen using ‘index’ command as shown below,

>>>

**>>>** data.index

MultiIndex(levels=[['a', 'b'], ['obj1', 'obj2', 'obj3', 'obj4']],

labels=[[0, 0, 0, 0, 1, 1, 1, 1], [0, 1, 2, 3, 0, 1, 2, 3]])

### 4.1.2. Partial indexing

Choosing a particular index from a hierarchical indexing is known as partial indexing.

* In the below code, index ‘b’ is extracted from the data,

>>>

**>>>** data['b']

obj1 15

obj2 25

obj3 35

obj4 25

dtype: int64

* Further, the data can be extracted based on inner level i.e. ‘obj’. Below result shows the two available values for ‘obj2’ in the Series.

>>>

**>>>** data[:, 'obj2']

a 20

b 25

dtype: int64

>>>

### 4.1.3. Unstack the data

We saw the use of unstack operation in the Section [Unstack](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#unstackex1). Unstack changes the row header to column header. Since the row index is changed to column index, therefore the Series will become the DataFrame in this case. Following are the some more example of unstacking the data,

>>>

**>>>** *# unstack based on first level i.e. a, b*

**>>>** *# note that data row-labels are a and b*

**>>>** data.unstack(0)

a b

obj1 10 15

obj2 20 25

obj3 30 35

obj4 40 25

**>>>** *# unstack based on second level i.e. 'obj'*

**>>>** data.unstack(1)

obj1 obj2 obj3 obj4

a 10 20 30 40

b 15 25 35 25

>>>

**>>>** *# by default innermost level is used for unstacking*

**>>>** d = data.unstack()

**>>>** d

obj1 obj2 obj3 obj4

a 10 20 30 40

b 15 25 35 25

* ‘stack()’ operation converts the column index to row index again. In above code, DataFrame ‘d’ has ‘obj’ as column index, this can be converted into row index using ‘stack’ operation,

>>>

**>>>** d.stack()

a obj1 10

obj2 20

obj3 30

obj4 40

b obj1 15

obj2 25

obj3 35

obj4 25

dtype: int64

### 4.1.4. Column indexing

Remember that, the column indexing is possible for DataFrame only (not for Series), because column-indexing require two dimensional data. Let’s create a new DataFrame as below for understanding the columns with multiple index,

>>>

**>>> import** **numpy** **as** **np**

**>>>** df = pd.DataFrame(np.arange(12).reshape(4, 3),

**...**  index = [['a', 'a', 'b', 'b'], ['one', 'two', 'three', 'four']],

**...**  columns = [['num1', 'num2', 'num3'], ['red', 'green', 'red']]

**...** )

>>>

**>>>** df

num1 num2 num3

red green red

a one 0 1 2

two 3 4 5

b three 6 7 8

four 9 10 11

>>>

**>>>** *# display row index*

**>>>** df.index

MultiIndex(levels=[['a', 'b'], ['four', 'one', 'three', 'two']],

labels=[[0, 0, 1, 1], [1, 3, 2, 0]])

**>>>** *# display column index*

**>>>** df.columns

MultiIndex(levels=[['num1', 'num2', 'num3'], ['green', 'red']],

labels=[[0, 1, 2], [1, 0, 1]])

* Note that, in previous section, we used the numbers for stack and unstack operation e.g. unstack(0) etc. We can give name to index as well as below,

>>>

**>>>** df.index.names=['key1', 'key2']

**>>>** df.columns.names=['n', 'color']

**>>>** df

n num1 num2 num3

color red green red

key1 key2

a one 0 1 2

two 3 4 5

b three 6 7 8

four 9 10 11

* Now, we can perform the partial indexing operations. In following code, various ways to access the data in a DataFrame are shown,

>>>

**>>>** *# accessing the column for num1*

**>>>** df['num1'] *# df.ix[:, 'num1']*

color red

key1 key2

a one 0

two 3

b three 6

four 9

**>>>** *# accessing the column for a*

**>>>** df.ix['a']

n num1 num2 num3

color red green red

key2

one 0 1 2

two 3 4 5

**>>>** *# access row 0 only*

**>>>** df.ix[0]

n color

num1 red 0

num2 green 1

num3 red 2

Name: (a, one), dtype: int32

### 4.1.5. Swap and sort level

We can swap the index level using ‘swaplevel’ command, which takes two level-numbers as input,

>>>

**>>>** df.swaplevel('key1', 'key2')

n num1 num2 num3

color red green red

key2 key1

one a 0 1 2

two a 3 4 5

three b 6 7 8

four b 9 10 11

>>>

Levels can be sorted using ‘sort\_index’ command. In below code, data is sorted by ‘key2’ names i.e. key2 is arranged alphabatically,

>>>

**>>>** df.sort\_index(level='key2')

n num1 num2 num3

color red green red

key1 key2

b four 9 10 11

a one 0 1 2

b three 6 7 8

a two 3 4 5

>>>

### 4.1.6. Summary statistics by level

We saw the example of groupby command in Section [Groupby](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#groupbyex1). Pandas provides some easier ways to perform those operations using ‘level’ shown below,

>>>

**>>>** *# add all rows with similar key1 name*

**>>>** df.sum(level = 'key1')

n num1 num2 num3

color red green red

key1

a 3 5 7

b 15 17 19

>>>

**>>>** *# add all the columns based on similar color*

**>>>** df.sum(level= 'color', axis=1)

color green red

key1 key2

a one 1 2

two 4 8

b three 7 14

four 10 20

## 4.2. File operations

In this section, various methods for reading and writing the files are discussed.

### 4.2.1. Reading files

Pandas supports various types of file format e.g. csv, text, excel and different database etc. Files are often stored in different formats as well e.g. files may or may not contain header, footer and comments etc.; therefore we need to process the content of file. Pandas provides various features which can process some of the common processing while reading the file. Some of these processing are shown in this section.

* Files can be read using ‘read\_csv’, ‘read\_table’ or ‘DataFrame.from\_csv’ options, as shown below. Note that, the output of all these methods are same, but we need to provide different parameters to read the file correctly.

Following are the contents of ‘ex1.csv’ file,

$ cat ex1.csv

a,b,c,d,message

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foo

Below are the outputs of different file reading methods. ‘read\_csv’ is general purpose method for reading the files, hence this method is used for rest of the tutorial,

>>>

**>>> import** **pandas** **as** **pd**

**>>>** *# DataFrame.from\_csv*

**>>>** df = pd.DataFrame.from\_csv('ex1.csv', index\_col=None)

**>>>** df

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

**>>>** *# read\_csv*

**>>>** df = pd.read\_csv('ex1.csv')

**>>>** df

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

**>>>** *# read\_table*

**>>>** df = pd.read\_table('ex1.csv', sep=',')

**>>>** df

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

>>>

* Note that, in above outputs, the headers are added from the file; but not all the files contain header. In this case, we need to explicitly define the header as below,

Following are the contents of ‘ex2.csv’ file,

$ cat ex2.csv

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,food

Since header is not present in above file, therefore we need to provide the “header” argument explicitly.

>>>

**>>> import** **pandas** **as** **pd**

**>>>** *# set header as none, default values will be used as header*

**>>>** pd.read\_csv('ex2.csv', header=None)

0 1 2 3 4

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

**>>>** *# specify the header using 'names'*

**>>>** pd.read\_csv('ex2.csv', names=['a', 'b', 'c', 'd', 'message'])

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

**>>>** *# specify the row and column header both*

**>>>** pd.read\_csv('ex2.csv', names=['a', 'b', 'c', 'd', 'message'], index\_col='message')

a b c d

message

hello 1 2 3 4

world 5 6 7 8

foo 9 10 11 12

>>>

* Hierarchical index can be created by providing a list to ‘index\_col’ argument,

Following are the contents of ‘csv\_mindex.csv’ file,

$ cat csv\_mindex.csv

key1,key2,value1,value2

one,a,1,2

one,b,3,4

one,c,5,6

one,d,7,8

two,a,9,10

two,b,11,12

two,c,13,14

two,d,15,16

The hierarchical index can be created with ‘key’ values as below,

>>>

**>>>** pd.read\_csv('csv\_mindex.csv', index\_col=['key1', 'key2'])

value1 value2

key1 key2

one a 1 2

b 3 4

c 5 6

d 7 8

two a 9 10

b 11 12

c 13 14

d 15 16

>>>

* Some files may contain additional information or comments, therefore we need to remove these information for processing the data. This can be done by using ‘skiprows’ command,

Following are the content of ‘ex4.csv’ file,

$ cat ex4.csv

*# hey!*

a,b,c,d,message

*# just wanted to make things more difficult for you*

*# who reads CSV files with computers, anyway?*

1,2,3,4,hello

5,6,7,8,world

9,10,11,12,foodh

In above results, lines 0, 2 and 3 contains some comments. These can be removed as follows,

>>>

**>>>** d = pd.read\_csv('ex4.csv', skiprows=[0,2,3])

**>>>** d

a b c d message

0 1 2 3 4 hello

1 5 6 7 8 world

2 9 10 11 12 foo

### 4.2.2. Writing data to a file

The ‘to\_csv’ command is used to save the file. In following code, previous data ‘d’ is saved in two files i.e. d\_out.csv and d\_out2.csv with and without index respectively,

>>>

**>>>** d.to\_csv('d\_out.csv')

**>>>** *# save without headers*

**>>>** d.to\_csv('d\_out2.csv', header=False, index=False)

Contents of above two files are shown below,

$ cat d\_out.csv

,a,b,c,d,message

0,1,2,3,4,hello

1,5,6,7,8,world

2,9,10,11,12,foo

$ cat d\_out2.csv

0,1,2,3,4,hello

1,5,6,7,8,world

2,9,10,11,12,foo

## 4.3. Merge

Merge or joins operations combine the data sets by liking rows using one or more keys. The ‘merge’ function is the main entry point for using these algorithms on the data. Let’s understand this by following examples,

>>>

**>>>** df1 = pd.DataFrame({ 'key' : ['b', 'b', 'a', 'c', 'a', 'a', 'b'],

**...**  'data1' : range(7)})

**>>>** df2 = pd.DataFrame({ 'key' : ['a', 'b', 'd'],

**...**  'data2' : range(3)})

**>>>** df1

data1 key

0 0 b

1 1 b

2 2 a

3 3 c

4 4 a

5 5 a

6 6 b

**>>>** df2 = pd.DataFrame({ 'key' : ['a', 'b', 'd', 'b'],

**...**  'data2' : range(4)})

**>>>** df2

data2 key

0 0 a

1 1 b

2 2 d

3 3 b

>>>

### 4.3.1. Many to one

* ‘Many to one’ merge joins the Cartesian product of the rows, e.g. df1 and df2 has total 3 and 2 rows of ‘b’ respectively, therefore join will result in total 6 rows. Further, it is better to define ‘on’ keyword while using the joins, as it makes code more readable,

>>>

**>>>** pd.merge(df1, df2) *# or pd.merge(df1, df2, on='key')*

data1 key data2

0 0 b 1

1 0 b 3

2 1 b 1

3 1 b 3

4 6 b 1

5 6 b 3

6 2 a 0

7 4 a 0

8 5 a 0

>>>

* In previous case, both the DataFrame have the same header ‘key’. In the following example data are joined based on different keys using ‘left\_on’ and ‘right\_on’ keywords,

>>>

**>>>** *# data is same as previous, only 'key' is replaces with 'key1' and 'key2'*

**>>>** df1 = pd.DataFrame({ 'key1' : ['b', 'b', 'a', 'c', 'a', 'a', 'b'],

**...**  'data1' : range(7)})

**>>>** df2 = pd.DataFrame({ 'key2' : ['a', 'b', 'd', 'b'],

**...**  'data1' : range(4)})

**>>>** pd.merge(df1, df2, left\_on='key1', right\_on='key2')

data1\_x key1 data1\_y key2

0 0 b 1 b

1 0 b 3 b

2 1 b 1 b

3 1 b 3 b

4 6 b 1 b

5 6 b 3 b

6 2 a 0 a

7 4 a 0 a

8 5 a 0 a

>>>

### 4.3.2. Inner and outer join

In previous example, we can see that uncommon entries in DataFrame ‘df1’ and ‘df2’ are missing from the merge e.g. ‘d’ is not in the merged data. This is an example of ‘inner join’ where only common keys are merged together. By default, pandas perform the inner join. To perform outer join, we need to use ‘how’ keyword which can have 3 different values i.e. ‘left’, ‘right’ and ‘outer’. ‘left’ option takes the left DataFrame and merge all it’s entries with other DataFrame. Similarly, ‘right’ option merge the entries of the right DataFrame with left DataFrame. Lastly, the ‘outer’ option merge all the entries from both the DataFrame, as shown below. Note that, the missing entries after joining the table are represented as ‘NaN’.

>>>

**>>>** *# left join*

**>>>** pd.merge(df1, df2, left\_on='key1', right\_on='key2', how="left")

data1\_x key1 data1\_y key2

0 0 b 1.0 b

1 0 b 3.0 b

2 1 b 1.0 b

3 1 b 3.0 b

4 2 a 0.0 a

5 3 c NaN NaN

6 4 a 0.0 a

7 5 a 0.0 a

8 6 b 1.0 b

9 6 b 3.0 b

**>>>** *# right join*

**>>>** pd.merge(df1, df2, left\_on='key1', right\_on='key2', how="right")

data1\_x key1 data1\_y key2

0 0.0 b 1 b

1 1.0 b 1 b

2 6.0 b 1 b

3 0.0 b 3 b

4 1.0 b 3 b

5 6.0 b 3 b

6 2.0 a 0 a

7 4.0 a 0 a

8 5.0 a 0 a

9 NaN NaN 2 d

**>>>** *# outer join*

**>>>** pd.merge(df1, df2, left\_on='key1', right\_on='key2', how="outer")

data1\_x key1 data1\_y key2

0 0.0 b 1.0 b

1 0.0 b 3.0 b

2 1.0 b 1.0 b

3 1.0 b 3.0 b

4 6.0 b 1.0 b

5 6.0 b 3.0 b

6 2.0 a 0.0 a

7 4.0 a 0.0 a

8 5.0 a 0.0 a

9 3.0 c NaN NaN

10 NaN NaN 2.0 d

### 4.3.3. Concatenating the data

We saw concatenation of data in Numpy. Pandas concatenation is more generalized than Numpy. It allows concatenation based on union or intersection of data along with labeling to visualize the grouping as shown in this section,

>>>

**>>>** s1 = pd.Series([0, 1], index=['a', 'b'])

**>>>** s2 = pd.Series([2, 1, 3], index=['c', 'd', 'e'])

**>>>** s3 = pd.Series([4, 7], index=['a', 'e'])

**>>>** s1

a 0

b 1

dtype: int64

**>>>** s2

c 2

d 1

e 3

dtype: int64

**>>>** s3

a 4

e 7

dtype: int64

**>>>** *# concatenate s1 and s2*

**>>>** pd.concat([s1, s2])

a 0

b 1

c 2

d 1

e 3

dtype: int64

**>>>** *# join on axis 1*

**>>>** pd.concat([s1, s2], axis=1)

0 1

a 0.0 NaN

b 1.0 NaN

c NaN 2.0

d NaN 1.0

e NaN 3.0

* In above results , it is difficult to identify the different pieces of concatenate operation. We can provide ‘keys’ to make the operation identifiable,

>>>

**>>>** pd.concat([s1, s2, s3], keys=['one', 'two', 'three'])

one a 0

b 1

two c 2

d 1

e 3

three a 4

e 7

dtype: int64

**Note**

Above concatenate operation are the union of two data set i.e. it is outer join. We can use “join=’inner’” for intersection of data.

>>>

**>>>** pd.concat([s1, s3], join='inner', axis=1)

0 1

a 0 4

* Concatenating the DataFrame is same as above. Following is the example of the concatenation of DataFrame. Note that ‘df1’ and ‘df2’ are defined at the beginning of this section.

>>>

**>>>** pd.concat([df1, df2], join='inner', axis=1, keys=['one', 'two'])

one two

data1 key1 data1 key2

0 0 b 0 a

1 1 b 1 b

2 2 a 2 d

3 3 c 3 b

* We can pass the DataFrame as dictionary as well for the concatenation operation. In this case, the keys of the dictionary will be used as ‘keys’ for the operation,

>>>

**>>>** pd.concat({ 'level1':df1, 'level2':df2}, axis=1, join='inner')

level1 level2

data1 key1 data1 key2

0 0 b 0 a

1 1 b 1 b

2 2 a 2 d

3 3 c 3 b

>>>

## 4.4. Data transformation

In previous section, we saw various operations to join the various data. Next, important step is the data transformation i.e. cleaning and filtering the data e.g. removing the duplicate entries and replacing the NaN values etc.

### 4.4.1. Removing duplicates

* Removing duplicate entries are quite easy with ‘drop\_duplicates’ command. Also, ‘duplicate()’ command can be used to check the duplicate entries as shown below,

>>>

**>>>** *# create DataFrame with duplicate entries*

**>>>** df = pd.DataFrame({'k1':['one']\*3 + ['two']\*4,

**...**  'k2':[1,1,2,3,3,4,4]})

**>>>** df

k1 k2

0 one 1

1 one 1

2 one 2

3 two 3

4 two 3

5 two 4

6 two 4

**>>>** *# see the duplicate entries*

**>>>** df.duplicated()

0 false

1 true

2 false

3 false

4 true

5 false

6 true

dtype: bool

**>>>** *# drop the duplicate entries*

**>>>** df.drop\_duplicates()

k1 k2

0 one 1

2 one 2

3 two 3

5 two 4

* Currently, last entry is removed by drop\_duplicates commnad. If we want to keep the last entry, then ‘keep’ keyword can be used,

>>>

**>>>** df.drop\_duplicates(keep="last")

k1 k2

1 one 1

2 one 2

4 two 3

6 two 4

>>>

* We can drop all the duplicate values from based on the specific columns as well,

>>>

**>>>** *# drop duplicate entries based on k1 only*

**>>>** df.drop\_duplicates(['k1'])

k1 k2

0 one 1

3 two 3

**>>>** *# drop if k1 and k2 column matched*

**>>>** df.drop\_duplicates(['k1', 'k2'])

k1 k2

0 one 1

2 one 2

3 two 3

5 two 4

>>>

### 4.4.2. Replacing values

Replacing value is very easy using pandas as below,

>>>

**>>>** *# replace 'one' with 'One'*

**>>>** df.replace('one', 'One')

k1 k2

0 One 1

1 One 1

2 One 2

3 two 3

4 two 3

5 two 4

6 two 4

**>>>** *# replace 'one'->'One' and 3->30*

**>>>** df.replace(['one', 3], ['One', '30'])

k1 k2

0 One 1

1 One 1

2 One 2

3 two 30

4 two 30

5 two 4

6 two 4

>>>

* Arguments can be passed as dictionary as well,

>>>

**>>>** df.replace({'one':'One', 3:30})

k1 k2

0 One 1

1 One 1

2 One 2

3 two 30

4 two 30

5 two 4

6 two 4

## 4.5. Groupby and data aggregation

### 4.5.1. Basics

We saw various groupby operation in Section [Groupby](https://pandasguide.readthedocs.io/en/latest/Pandas/overview.html#groupbyex1). Here, some more features of gropby operations are discussed.

Let’s create a DataFrame first,

>>>

**>>>** df = pd.DataFrame({'k1':['a', 'a', 'b', 'b', 'a'],

**...**  'k2':['one', 'two', 'one', 'two', 'one'],

**...**  'data1': [2, 3, 3, 2, 4],

**...**  'data2': [5, 5, 5, 5, 10]})

**>>>** df

data1 data2 k1 k2

0 2 5 a one

1 3 5 a two

2 3 5 b one

3 2 5 b two

4 4 10 a one

* Now, create a group based on ‘k1’ and find the mean value as below. In the following code, rows (0, 1, 4) and (2, 3) are grouped together. Therefore mean values are 3 and 2.5.

>>>

**>>>** gp1 = df['data1'].groupby(df['k1'])

**>>>** gp1

<pandas.core.groupby.SeriesGroupBy object at 0xb21f6bcc>

**>>>** gp1.mean()

k1

a 3.0

b 2.5

Name: data1, dtype: float64

* We can pass multiple parameters for grouping as well,

>>>

**>>>** gp2 = df['data1'].groupby([df['k1'], df['k2']])

**>>>** mean = gp2.mean()

**>>>** mean

k1 k2

a one 3

two 3

b one 3

two 2

Name: data1, dtype: int64

>>>

### 4.5.2. Iterating over group

* The groupby operation supports iteration which generates the tuple with two values i.e. group-name and data.

>>>

**>>> for** name, group **in** gp1:

**...**  **print**(name)

**...**  **print**(group)

**...**

a

0 2

1 3

4 4

Name: data1, dtype: int64

b

2 3

3 2

Name: data1, dtype: int64

* If groupby operation is performed based on multiple keys, then it will generate a tuple for keys as well,

>>>

**>>> for** name, group **in** gp2:

**...**  **print**(name)

**...**  **print**(group)

**...**

('a', 'one')

0 2

4 4

Name: data1, dtype: int64

('a', 'two')

1 3

Name: data1, dtype: int64

('b', 'one')

2 3

Name: data1, dtype: int64

('b', 'two')

3 2

Name: data1, dtype: int64

**>>>** *# seperate key values as well*

**>>> for** (k1, k2), group **in** gp2:

**...**  **print**(k1, k2)

**...**  **print**(group)

**...**

a one

0 2

4 4

Name: data1, dtype: int64

a two

1 3

Name: data1, dtype: int64

b one

2 3

Name: data1, dtype: int64

b two

3 2

Name: data1, dtype: int64

>>>

### 4.5.3. Data aggregation

We can perform various aggregation operation on the grouped data as well,

>>>

**>>>** gp1.max()

k1

a 4

b 3

Name: data1, dtype: int64

**>>>** gp2.min()

k1 k2

a one 2

two 3

b one 3

two 2

Name: data1, dtype: int64

# 5. Time series

## 5.1. Dates and times

### 5.1.1. Generate series of time

A series of time can be generated using ‘date\_range’ command. In below code, ‘periods’ is the total number of samples; whereas freq = ‘M’ represents that series must be generated based on ‘Month’.

* By default, pandas consider ‘M’ as end of the month. Use ‘MS’ for start of the month. Similarly, other options are also available for day (‘D’), business days (‘B’) and hours (‘H’) etc.

>>>

**>>> import** **pandas** **as** **pd**

**>>> import** **numpy** **as** **np**

**>>>** rng = pd.date\_range('2011-03-01 10:15', periods = 10, freq = 'M')

**>>>** rng

DatetimeIndex(['2011-03-31 10:15:00', '2011-04-30 10:15:00',

'2011-05-31 10:15:00', '2011-06-30 10:15:00',

'2011-07-31 10:15:00', '2011-08-31 10:15:00',

'2011-09-30 10:15:00', '2011-10-31 10:15:00',

'2011-11-30 10:15:00', '2011-12-31 10:15:00'],

dtype='datetime64[ns]', freq='M')

**>>>** rng = pd.date\_range('2015 Jul 2 10:15', periods = 10, freq = 'M')

**>>>** rng

DatetimeIndex(['2015-07-31 10:15:00', '2015-08-31 10:15:00',

'2015-09-30 10:15:00', '2015-10-31 10:15:00',

'2015-11-30 10:15:00', '2015-12-31 10:15:00',

'2016-01-31 10:15:00', '2016-02-29 10:15:00',

'2016-03-31 10:15:00', '2016-04-30 10:15:00'],

dtype='datetime64[ns]', freq='M')

* Similarly, we can generate the time series using ‘start’ and ‘end’ parameters as below,

>>>

**>>>** rng = pd.date\_range(start = '2015 Jul 2 10:15', end = '2015 July 12', freq = '12H')

**>>>** rng

DatetimeIndex(['2015-07-02 10:15:00', '2015-07-02 22:15:00',

'2015-07-03 10:15:00', '2015-07-03 22:15:00',

'2015-07-04 10:15:00', '2015-07-04 22:15:00',

'2015-07-05 10:15:00', '2015-07-05 22:15:00',

'2015-07-06 10:15:00', '2015-07-06 22:15:00',

'2015-07-07 10:15:00', '2015-07-07 22:15:00',

'2015-07-08 10:15:00', '2015-07-08 22:15:00',

'2015-07-09 10:15:00', '2015-07-09 22:15:00',

'2015-07-10 10:15:00', '2015-07-10 22:15:00',

'2015-07-11 10:15:00', '2015-07-11 22:15:00'],

dtype='datetime64[ns]', freq='12H')

**>>>** len(rng)

20

* Time zone can be specified for generating the series,

>>>

**>>>** rng = pd.date\_range(start = '2015 Jul 2 10:15', end = '2015 July 12', freq = '12H', tz='Asia/Kolkata')

**>>>** rng

DatetimeIndex(['2015-07-02 10:15:00+05:30', '2015-07-02 22:15:00+05:30',

'2015-07-03 10:15:00+05:30', '2015-07-03 22:15:00+05:30',

'2015-07-04 10:15:00+05:30', '2015-07-04 22:15:00+05:30',

'2015-07-05 10:15:00+05:30', '2015-07-05 22:15:00+05:30',

'2015-07-06 10:15:00+05:30', '2015-07-06 22:15:00+05:30',

'2015-07-07 10:15:00+05:30', '2015-07-07 22:15:00+05:30',

'2015-07-08 10:15:00+05:30', '2015-07-08 22:15:00+05:30',

'2015-07-09 10:15:00+05:30', '2015-07-09 22:15:00+05:30',

'2015-07-10 10:15:00+05:30', '2015-07-10 22:15:00+05:30',

'2015-07-11 10:15:00+05:30', '2015-07-11 22:15:00+05:30'],

dtype='datetime64[ns, Asia/Kolkata]', freq='12H')

>>>

* Further, we can change the time zone of the data for various comparison,

>>>

**>>>** rng.tz\_convert('Australia/Sydney')DatetimeIndex(['2015-07-02 14:45:00+10:00', '2015-07-03 02:45:00+10:00',

'2015-07-03 14:45:00+10:00', '2015-07-04 02:45:00+10:00',

'2015-07-04 14:45:00+10:00', '2015-07-05 02:45:00+10:00',

'2015-07-05 14:45:00+10:00', '2015-07-06 02:45:00+10:00',

'2015-07-06 14:45:00+10:00', '2015-07-07 02:45:00+10:00',

'2015-07-07 14:45:00+10:00', '2015-07-08 02:45:00+10:00',

'2015-07-08 14:45:00+10:00', '2015-07-09 02:45:00+10:00',

'2015-07-09 14:45:00+10:00', '2015-07-10 02:45:00+10:00',

'2015-07-10 14:45:00+10:00', '2015-07-11 02:45:00+10:00',

'2015-07-11 14:45:00+10:00', '2015-07-12 02:45:00+10:00'],

dtype='datetime64[ns, Australia/Sydney]', freq='12H')

* Note that types of these dates are Timestamp,

>>>

**>>>** type(rng[0])

<class 'pandas.tslib.Timestamp'>

>>>

### 5.1.2. Convert string to dates

Dates in string formats can be converted into time stamp using ‘to\_datetime’ option as below,

>>>

**>>>** dd = ['07/07/2015', '08/12/2015', '12/04/2015']

**>>>** dd

['07/07/2015', '08/12/2015', '12/04/2015']

**>>>** type(dd[0])

<class 'str'>

**>>>** *# American style*

**>>>** list(pd.to\_datetime(dd))

[Timestamp('2015-07-07 00:00:00'), Timestamp('2015-08-12 00:00:00'), Timestamp('2015-12-04 00:00:00')]

**>>>** *# European format*

**>>>** d = list(pd.to\_datetime(dd, dayfirst=True))

**>>>** d

[Timestamp('2015-07-07 00:00:00'), Timestamp('2015-12-08 00:00:00'), Timestamp('2015-04-12 00:00:00')]

**>>>** type(d[0])

<class 'pandas.tslib.Timestamp'>

>>>

### 5.1.3. Periods

Periods represents the time span e.g. days, years, quarter or month etc. Period class in pandas allows us to convert the frequency easily.

#### 5.1.3.1. Generating periods and frequency conversion

In following code, period is generated using ‘Period’ command with frequency ‘M’. Note that, when we use ‘asfreq’ operation with ‘start’ operation the date is ‘01’ where as it is ‘31’ with ‘end’ option.

>>>

**>>>** pr = pd.Period('2012', freq='M')

**>>>** pr.asfreq('D', 'start')

Period('2012-01-01', 'D')

**>>>** pr.asfreq('D', 'end')

Period('2012-01-31', 'D')

>>>

#### 5.1.3.2. Period arithmetic

We can perform various arithmetic operation on periods. All the operations will be performed based on ‘freq’,

>>>

**>>>** pr = pd.Period('2012', freq='A') *# Annual*

**>>>** pr

Period('2012', 'A-DEC')

**>>>** pr + 1

Period('2013', 'A-DEC')

**>>>** *# Year to month conversion*

**>>>** prMonth = pr.asfreq('M')

**>>>** prMonth

Period('2012-12', 'M')

**>>>** prMonth - 1

Period('2012-11', 'M')

>>>

#### 5.1.3.3. Creating period range

A range of periods can be created using ‘period\_range’ command,

>>>

**>>>** prg = pd.period\_range('2010', '2015', freq='A')

**>>>** prg

PeriodIndex(['2010', '2011', '2012', '2013', '2014', '2015'], dtype='int64', freq='A-DEC')

**>>>** *# create a series with index as 'prg'*

**>>>** data = pd.Series(np.random.rand(len(prg)), index=prg)

**>>>** data

2010 0.785453

2011 0.606939

2012 0.558619

2013 0.321185

2014 0.224793

2015 0.561374

Freq: A-DEC, dtype: float64

>>>

#### 5.1.3.4. Converting string-dates to period

Conversion of string-dates to period is the two step process, i.e. first we need to convert the string to date format and then convert the dates in periods as shown below,

>>>

**>>>** *# dates as string*

**>>>** dates = ['2013-02-02', '2012-02-02', '2013-02-02']

**>>>** *# convert string to date format*

**>>>** d = pd.to\_datetime(dates)

**>>>** d

DatetimeIndex(['2013-02-02', '2012-02-02', '2013-02-02'], dtype='datetime64[ns]', freq=None)

**>>>** *# create PeriodIndex from DatetimeIndex*

**>>>** prd = d.to\_period(freq='M')

**>>>** prd

PeriodIndex(['2013-02', '2012-02', '2013-02'], dtype='int64', freq='M')

**>>>** *# change frequency type*

**>>>** prd.asfreq('D')

PeriodIndex(['2013-02-28', '2012-02-29', '2013-02-28'], dtype='int64', freq='D')

**>>>** prd.asfreq('Y')

PeriodIndex(['2013', '2012', '2013'], dtype='int64', freq='A-DEC')

#### 5.1.3.5. Convert periods to timestamps

Periods can be converted back to timestamps using ‘to\_timestamp’ command,

>>>

**>>>** prd

PeriodIndex(['2013-02', '2012-02', '2013-02'], dtype='int64', freq='M')

**>>>** prd.to\_timestamp()

DatetimeIndex(['2013-02-01', '2012-02-01', '2013-02-01'], dtype='datetime64[ns]', freq=None)

**>>>** prd.to\_timestamp(how='end')

DatetimeIndex(['2013-02-28', '2012-02-29', '2013-02-28'], dtype='datetime64[ns]', freq=None)

>>>

### 5.1.4. Time offsets

Time offset can be defined as follows. Further we can perform various operations on time as as well e.g. adding and subtracting etc.

>>>

**>>>** *# generate time offset*

**>>>** pd.Timedelta('3 days')

Timedelta('3 days 00:00:00')

**>>>** pd.Timedelta('3M')

Timedelta('0 days 00:03:00')

**>>>** pd.Timedelta('4 days 3M')

Timedelta('4 days 00:03:00')

>>>

**>>>** *# adding Timedelta to time*

**>>>** pd.Timestamp('9 July 2016 12:00') + pd.Timedelta('1 day 3 min')

Timestamp('2016-07-10 12:03:00')

>>>

**>>>** *# add Timedelta to complete rng*

**>>>** rng + pd.Timedelta('1 day')

DatetimeIndex(['2015-07-03 10:15:00+05:30', '2015-07-03 22:15:00+05:30',

'2015-07-04 10:15:00+05:30', '2015-07-04 22:15:00+05:30',

'2015-07-05 10:15:00+05:30', '2015-07-05 22:15:00+05:30',

'2015-07-06 10:15:00+05:30', '2015-07-06 22:15:00+05:30',

'2015-07-07 10:15:00+05:30', '2015-07-07 22:15:00+05:30',

'2015-07-08 10:15:00+05:30', '2015-07-08 22:15:00+05:30',

'2015-07-09 10:15:00+05:30', '2015-07-09 22:15:00+05:30',

'2015-07-10 10:15:00+05:30', '2015-07-10 22:15:00+05:30',

'2015-07-11 10:15:00+05:30', '2015-07-11 22:15:00+05:30',

'2015-07-12 10:15:00+05:30', '2015-07-12 22:15:00+05:30'],

dtype='datetime64[ns, Asia/Kolkata]', freq='12H')

>>>

### 5.1.5. Index data with time

In this section, time is used as index for Series and DataFrame; and then various operations are performed on these data structures.

* First, create a time series using ‘date\_range’ option as below.

>>>

**>>>** dates = pd.date\_range('2015-01-12', '2015-06-14', freq = 'M')

**>>>** dates

DatetimeIndex(['2015-01-31', '2015-02-28', '2015-03-31', '2015-04-30',

'2015-05-31'],

dtype='datetime64[ns]', freq='M')

**>>>** len(dates)

5

* Next, create a Series of temperature of length same as dates,

>>>

**>>>** atemp = pd.Series([100.2, 98, 93, 98, 100], index=dates)

**>>>** atemp

2015-01-31 100.2

2015-02-28 98.0

2015-03-31 93.0

2015-04-30 98.0

2015-05-31 100.0

Freq: M, dtype: float64

>>>

* Now, time index can be used to access the temperatures as below,

>>>

**>>>** idx = atemp.index[3]

**>>>** idx

Timestamp('2015-04-30 00:00:00', offset='M')

**>>>** atemp[idx]

98.0

>>>

* Next, make another temperature series ‘stemp’ and create a DataFrame using ‘stemp’ and ‘atemp’ as below,

>>>

**>>>** stemp = pd.Series([89, 98, 100, 88, 89], index=dates)

**>>>** stemp

2015-01-31 89

2015-02-28 98

2015-03-31 100

2015-04-30 88

2015-05-31 89

Freq: M, dtype: int64

>>>

**>>>** *# create DataFrame*

**>>>** temps = pd.DataFrame({'Auckland':atemp, 'Delhi':stemp})

**>>>** temps

Auckland Delhi

2015-01-31 100.2 89

2015-02-28 98.0 98

2015-03-31 93.0 100

2015-04-30 98.0 88

2015-05-31 100.0 89

>>>

**>>>** *# check the temperature of Auckland*

**>>>** temps['Auckland'] *# or temps.Auckland*

2015-01-31 100.2

2015-02-28 98.0

2015-03-31 93.0

2015-04-30 98.0

2015-05-31 100.0

Freq: M, Name: Auckland, dtype: float64

>>>

* We can add one more column to DataFrame ‘temp’ which shows the temperature differences between these two cities,

>>>

**>>>** temps['Diff'] = temps['Auckland'] - temps['Delhi']

**>>>** temps

Auckland Delhi Diff

2015-01-31 100.2 89 11.2

2015-02-28 98.0 98 0.0

2015-03-31 93.0 100 -7.0

2015-04-30 98.0 88 10.0

2015-05-31 100.0 89 11.0

>>>

**>>>** *# delete the temp['Diff']*

**>>> del** temps['Diff']

**>>>** temps

Auckland Delhi

2015-01-31 100.2 89

2015-02-28 98.0 98

2015-03-31 93.0 100

2015-04-30 98.0 88

2015-05-31 100.0 89

>>>

## 5.2. Application

In previous section, we saw some basics of time series. In this section, we will learn some usage of time series with an example,

### 5.2.1. Basics

* First, load the stocks.csv file as below,

>>>

**>>> import** **pandas** **as** **pd**

**>>>** df = pd.read\_csv('stocks.csv')

**>>>** df.head()

date AA GE IBM MSFT

0 1990-02-01 00:00:00 4.98 2.87 16.79 0.51

1 1990-02-02 00:00:00 5.04 2.87 16.89 0.51

2 1990-02-05 00:00:00 5.07 2.87 17.32 0.51

3 1990-02-06 00:00:00 5.01 2.88 17.56 0.51

4 1990-02-07 00:00:00 5.04 2.91 17.93 0.51

>>>

* If we check the format of ‘date’ column, we will find that it is string (not the date),

>>>

**>>>** d = df.date[0]

**>>>** d

'1990-02-01 00:00:00'

**>>>** type(d)

<class 'str'>

>>>

* To import ‘date’ as time stamp, ‘parse\_dates’ option can be used as below,

>>>

**>>>** df = pd.DataFrame.from\_csv('stocks.csv', parse\_dates=['date'])

**>>>** d = df.date[0]

**>>>** d

Timestamp('1990-02-01 00:00:00')

**>>>** type(d)

<class 'pandas.tslib.Timestamp'>

>>>

**>>>** df.head()

date AA GE IBM MSFT

0 1990-02-01 4.98 2.87 16.79 0.51

1 1990-02-02 5.04 2.87 16.89 0.51

2 1990-02-05 5.07 2.87 17.32 0.51

3 1990-02-06 5.01 2.88 17.56 0.51

4 1990-02-07 5.04 2.91 17.93 0.51

* Since, we want to used the date as index, therefore load it as index,

>>>

**>>>** df = pd.DataFrame.from\_csv('stocks.csv', parse\_dates=['date'], index\_col='date')

**>>>** df.head()

Unnamed: 0 AA GE IBM MSFT

date

1990-02-01 0 4.98 2.87 16.79 0.51

1990-02-02 1 5.04 2.87 16.89 0.51

1990-02-05 2 5.07 2.87 17.32 0.51

1990-02-06 3 5.01 2.88 17.56 0.51

1990-02-07 4 5.04 2.91 17.93 0.51

* Since, ‘Unnamed: 0’ is not a useful column, therefore we can remove it as below,

>>>

**>>> del** df['Unnamed: 0']

**>>>** df.head()

AA GE IBM MSFT

date

1990-02-01 4.98 2.87 16.79 0.51

1990-02-02 5.04 2.87 16.89 0.51

1990-02-05 5.07 2.87 17.32 0.51

1990-02-06 5.01 2.88 17.56 0.51

1990-02-07 5.04 2.91 17.93 0.51

>>>

* Before going further, let’s check the name of the index as it will be used at various places along with plotting the data, where index will be used automatically for plots. **Note that, data is used as columns as well as index by using ‘drop’ keyword.**

>>>

**>>>** *# check the name of the index*

**>>>** df.index.name

'date'

>>>

* Let’s redo all the above steps in different ways,

>>>

**>>>** *# load and display first file line of the file*

**>>>** stocks = pd.DataFrame.from\_csv('stocks.csv', parse\_dates=['date'])

**>>>** stocks.head()

date AA GE IBM MSFT

0 1990-02-01 4.98 2.87 16.79 0.51

1 1990-02-02 5.04 2.87 16.89 0.51

2 1990-02-05 5.07 2.87 17.32 0.51

3 1990-02-06 5.01 2.88 17.56 0.51

4 1990-02-07 5.04 2.91 17.93 0.51

**>>>** stocks.index.name *# nothing is set as index*

**>>>** *# set date as index but do not remove it from column*

**>>>** stocks = stocks.set\_index('date', drop=False)

**>>>** stocks.index.name

'date'

**>>>** stocks.head()

date AA GE IBM MSFT

date

1990-02-01 1990-02-01 4.98 2.87 16.79 0.51

1990-02-02 1990-02-02 5.04 2.87 16.89 0.51

1990-02-05 1990-02-05 5.07 2.87 17.32 0.51

1990-02-06 1990-02-06 5.01 2.88 17.56 0.51

1990-02-07 1990-02-07 5.04 2.91 17.93 0.51

>>>

**>>>** *# check the type of date*

**>>>** type(stocks.date[0])

<class 'pandas.tslib.Timestamp'>

>>>

* Data can be accessed by providing the date in any valid format, as shown below,

>>>

**>>>** *# all four commands have same results*

**>>>** *# stocks.ix['1990, 02, 01']*

**>>>** *# stocks.ix['1990-02-01']*

**>>>** *# stocks.ix['1990/02/01']*

**>>>** stocks.ix['1990-Feb-01']

date 1990-02-01 00:00:00

AA 4.98

GE 2.87

IBM 16.79

MSFT 0.51

Name: 1990-02-01 00:00:00, dtype: object

>>>

* We can display the results in between some range with slice operation e.g. from 01/Feb/90 to 06/Feb/90. Note that, last date of the slice is included in the results,

>>>

**>>>** stocks.ix['1990-Feb-01':'1990-Feb-06']

date AA GE IBM MSFT

date

1990-02-01 1990-02-01 4.98 2.87 16.79 0.51

1990-02-02 1990-02-02 5.04 2.87 16.89 0.51

1990-02-05 1990-02-05 5.07 2.87 17.32 0.51

1990-02-06 1990-02-06 5.01 2.88 17.56 0.51

>>>

**>>>** *# select all from Feb-1990 and display first 5*

**>>>** stocks.ix['1990-Feb'].head()

date AA GE IBM MSFT

date

1990-02-01 1990-02-01 4.98 2.87 16.79 0.51

1990-02-02 1990-02-02 5.04 2.87 16.89 0.51

1990-02-05 1990-02-05 5.07 2.87 17.32 0.51

1990-02-06 1990-02-06 5.01 2.88 17.56 0.51

1990-02-07 1990-02-07 5.04 2.91 17.93 0.51

>>>

**>>>** *# use python-timedelta or pandas-offset for defining range*

**>>> from** **datetime** **import** datetime, timedelta

**>>>** start = datetime(1990, 2, 1)

**>>>** *# stocks.ix[start:start+timedelta(days=5)] # python-timedelta*

**>>>** stocks.ix[start:start+pd.offsets.Day(5)] *# pandas-offset*

date AA GE IBM MSFT

date

1990-02-01 1990-02-01 4.98 2.87 16.79 0.51

1990-02-02 1990-02-02 5.04 2.87 16.89 0.51

1990-02-05 1990-02-05 5.07 2.87 17.32 0.51

1990-02-06 1990-02-06 5.01 2.88 17.56 0.51

>>>

**Note**

Above slice operation works only if the dates are in sorted order. If dates are not sorted then we need to sort them first by using sort\_index() command i.e. stocks.sort\_index()

### 5.2.2. Resampling

Resampling is the conversion of time series from one frequency to another. If we convert higher frequency data to lower frequency, then it is known as down-sampling; whereas if data is converted to low frequency to higher frequency, then it is called up-sampling.

* Suppose, we want to see the data at the end of each month only (not on daily basis), then we can use following resampling code,

>>>

**>>>** stocks.ix[pd.date\_range(stocks.index[0], stocks.index[-1], freq='M')].head()

date AA GE IBM MSFT

1990-02-28 1990-02-28 5.22 2.89 18.06 0.54

1990-03-31 NaT NaN NaN NaN NaN # it is not business day i.e. sat/sun

1990-04-30 1990-04-30 5.07 2.99 18.95 0.63

1990-05-31 1990-05-31 5.39 3.24 21.10 0.80

1990-06-30 NaT NaN NaN NaN NaN

**>>>** *# 'BM' can be used for 'business month'*

**>>>** stocks.ix[pd.date\_range(stocks.index[0], stocks.index[-1], freq='BM')].head()

date AA GE IBM MSFT

1990-02-28 1990-02-28 5.22 2.89 18.06 0.54

1990-03-30 1990-03-30 5.26 3.01 18.45 0.60

1990-04-30 1990-04-30 5.07 2.99 18.95 0.63

1990-05-31 1990-05-31 5.39 3.24 21.10 0.80

1990-06-29 1990-06-29 5.21 3.26 20.66 0.83

>>>

**>>>** *# confirm the entry on 1990-03-30*

**>>>** stocks.ix['1990-Mar-30']

date 1990-03-30 00:00:00

AA 5.26

GE 3.01

IBM 18.45

MSFT 0.6

Name: 1990-03-30 00:00:00, dtype: object

* Pandas provides easier way to write the above code i.e. using ‘resampling’. Further, resampling provides various features e.g. resample the data and show the mean value of the resampled data or maximum value of the data etc., as shown below,

**Downsampling**

Following is the example of downsampling.

>>>

**>>>** *# resample and find mean of each bin*

**>>>** stocks.resample('BM').mean().head()

AA GE IBM MSFT

date

1990-02-28 5.043684 2.873158 17.781579 0.523158

1990-03-30 5.362273 2.963636 18.466818 0.595000

1990-04-30 5.141000 3.037500 18.767500 0.638500

1990-05-31 5.278182 3.160000 20.121818 0.731364

1990-06-29 5.399048 3.275714 20.933810 0.821429

**>>>** *# size() : total number of rows in each bin*

**>>>** stocks.resample('BM').size().head(3)

date

1990-02-28 19 # total 19 business days in Feb-90

1990-03-30 22

1990-04-30 20

Freq: BM, dtype: int64

**>>>** *# count total number of rows in each bin for each column*

**>>>** stocks.resample('BM').count().head(3)

date AA GE IBM MSFT

date

1990-02-28 19 19 19 19 19

1990-03-30 22 22 22 22 22

1990-04-30 20 20 20 20 20

>>>

**>>>** *# display last resample value from each bin*

**>>>** ds = stocks.resample('BM').asfreq().head()

**>>>** ds

date AA GE IBM MSFT

date

1990-02-28 1990-02-28 5.22 2.89 18.06 0.54

1990-03-30 1990-03-30 5.26 3.01 18.45 0.60

1990-04-30 1990-04-30 5.07 2.99 18.95 0.63

1990-05-31 1990-05-31 5.39 3.24 21.10 0.80

1990-06-29 1990-06-29 5.21 3.26 20.66 0.83

>>>

**Upsampling**

* When we upsample the data, the values are filled by NaN; therefore we need to use ‘fillna’ method to replace the NaN value with some other values,as shown below,

>>>

**>>>** *# blank places are filled by NaN*

**>>>** rs = ds.resample('B').asfreq()

**>>>** rs.head()

date AA GE IBM MSFT

date

1990-02-28 1990-02-28 5.22 2.89 18.06 0.54

1990-03-01 NaT NaN NaN NaN NaN

1990-03-02 NaT NaN NaN NaN NaN

1990-03-05 NaT NaN NaN NaN NaN

1990-03-06 NaT NaN NaN NaN NaN

**>>>** *# forward fill the NaN*

**>>>** rs = ds.resample('B').asfreq().fillna(method='ffill')

**>>>** rs.head()

date AA GE IBM MSFT

date

1990-02-28 1990-02-28 5.22 2.89 18.06 0.54

1990-03-01 1990-02-28 5.22 2.89 18.06 0.54

1990-03-02 1990-02-28 5.22 2.89 18.06 0.54

1990-03-05 1990-02-28 5.22 2.89 18.06 0.54

1990-03-06 1990-02-28 5.22 2.89 18.06 0.54

>>>

### 5.2.3. Plotting the data

In this section, we will plot various data from the DataFrame ‘stocks’ for various time ranges,

* First, plot the data of ‘AA’ for complete range,

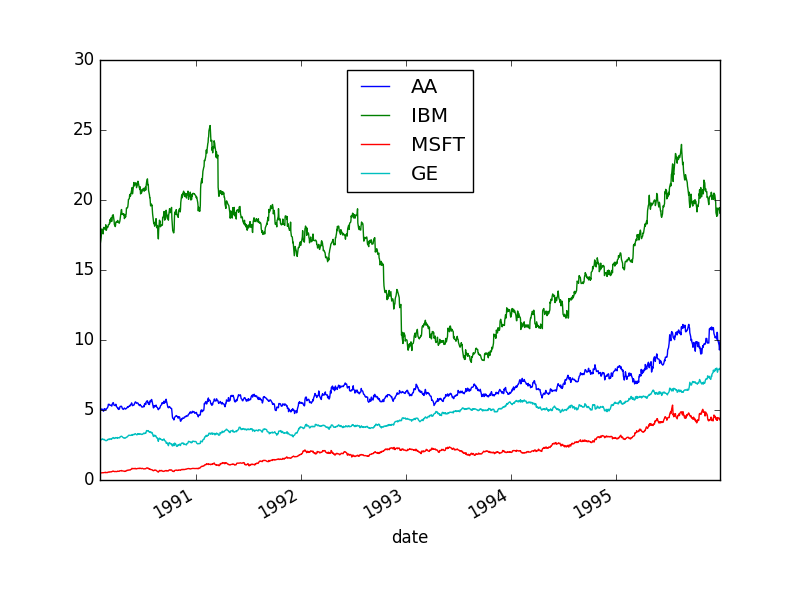
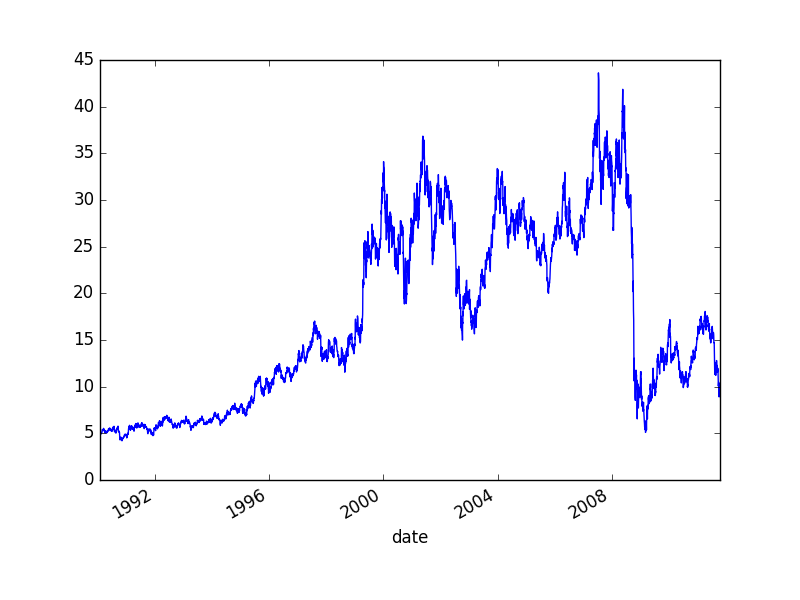
>>>

**>>> import** **matplotlib.pyplot** **as** **plt**

**>>>** stocks.AA.plot()

<matplotlib.axes.\_subplots.AxesSubplot object at 0xa9c3060c>

**>>>** plt.show()

[](https://pandasguide.readthedocs.io/en/latest/_images/plot3.png)[](https://pandasguide.readthedocs.io/en/latest/_images/plot12.png)

* We can plot various data in the same window, by selecting the column using ‘ix’,

>>>

**>>>** stocks.ix['1990':'1995', ['AA', 'IBM', 'MSFT', 'GE']].plot()

<matplotlib.axes.\_subplots.AxesSubplot object at 0xa9c2d2ac>

**>>>** plt.show()

>>>

### 5.2.4. Moving windows functions

Pandas provide the ways to analyze the data over a sliding window e.g. in below code the data of ‘AA’ is plotted aalong with the mean value over a window of length 250,

>>>

**>>>** stocks.AA.plot()

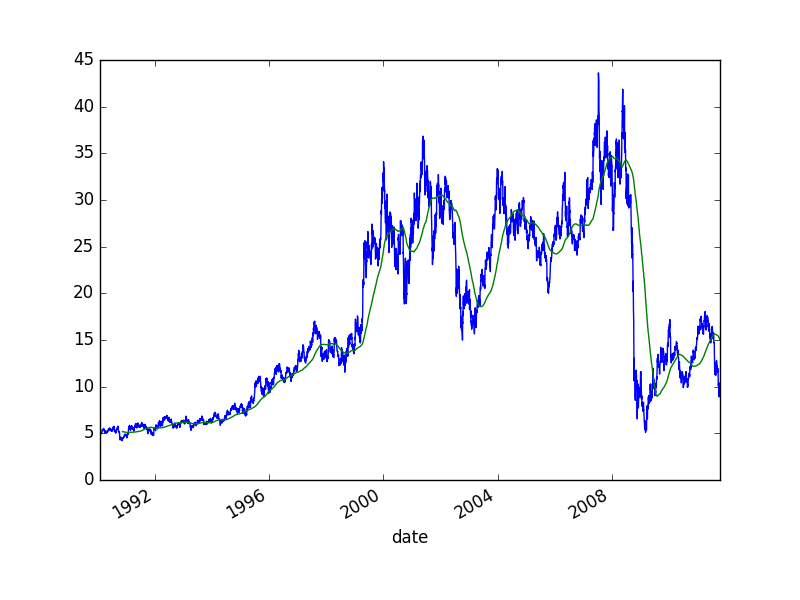
<matplotlib.axes.\_subplots.AxesSubplot object at 0xa9c5f4ec>

**>>>** stocks.AA.rolling(window=200,center=False).mean().plot()

<matplotlib.axes.\_subplots.AxesSubplot object at 0xa9c5f4ec>

**>>>** plt.show()

>>>

[](https://pandasguide.readthedocs.io/en/latest/_images/plot41.png)

# 6. Reading multiple files

In previous chapters, we used only one or two files to read the data. In this chapter, multiple files are concatenated to analyze the data.

## 6.1. Example: Baby names trend

In this section, various operations are performed on the various text-files to gather the useful information from it. These text file contains the list to names of babies since 1880. Each record in the individual annual files has the format “name,sex,number,” where name is 2 to 15 characters, sex is M (male) or F (female) and “number” is the number of occurrences of the name. Each file is sorted first on sex and then on number of occurrences in descending order. When there is a tie on the number of occurrences, names are listed in alphabetical order. Following is the list of files on which various operations will be performed.

$ ls

yob1880.txt yob1882.txt yob1884.txt yob1886.txt

yob1881.txt yob1883.txt yob1885.txt yob1887.txt

Following are the first ten lines in the yob1880.txt file,

$ head -n 10 yob1980.txt

Jennifer,F,58375

Amanda,F,35817

Jessica,F,33914

Melissa,F,31625

Sarah,F,25737

Heather,F,19965

Nicole,F,19910

Amy,F,19832

Elizabeth,F,19523

Michelle,F,19113

## 6.2. Total boys and girls in year 1880

* First, we will read one file and then check to total number of rows in that file,

>>>

**>>> import** **pandas** **as** **pd**

**>>>** names1880 = pd.read\_csv('yob1880.txt', names=['name', 'gender', 'birthcount'])

**>>>** *# total number of rows in yob1880.txt*

**>>>** len(names1880)

2000

>>>

**>>>** names1880.head()

name gender birthcount

0 Mary F 7065

1 Anna F 2604

2 Emma F 2003

3 Elizabeth F 1939

4 Minnie F 1746

>>>

* Note that, the file contains 2000 rows; and each row contains a name and total number of babies with that particular name along with the gender information. We can calculate the total number of boys and girls by adding the ‘birthcount’ based on gender; i.e. we need to group the data based on gender and then add the individual group’s birthcount,

>>>

**>>>** *# total number of boys and girls in year 1880*

**>>>** names1880.groupby('gender').birthcount.sum()

gender

F 90993

M 110493

Name: birthcount, dtype: int64

>>>

## 6.3. pivot\_table

In previous chapters, we saw various examples of groupby and unstack operations. These two operations can be performed by a single operation as well i.e. pivot\_table. In this section, we will calculate the total number of births in years 1880 to 1887 using pivot\_table. For this first we need to merge the data from the files for these year.

>>>

**>>>** years = range(1880, 1887)

**>>>** pieces = []

**>>>** columns = ['name', 'gender', 'birthcount']

**>>> for** year **in** years:

**...**  path = 'yob{}.txt'.format(year)

**...**  columns = ['name', 'gender', 'birthcount']

**...**  **for** year **in** years:

**...**  path = 'yob{}.txt'.format(year)

**...**  df = pd.read\_csv(path, names=columns)

**...**  df['year']=year

**...**  pieces.append(df)

**...**  allData = pd.concat(pieces, ignore\_index=True)

**...**

>>>

**>>>** len(allData)

105903

>>>

**>>>** allData.head(2)

name gender birthcount year

0 Mary F 7065 1880

1 Anna F 2604 1880

* Total number of birth can be calculated using pivot\_table, as shown below,

>>>

**>>> import** **matplotlib.pyplot** **as** **plt**

**>>>** total\_births = allData.pivot\_table('birthcount', index=['year'], columns=['gender'], aggfunc=sum)

**>>>** total\_births.head(3)

gender F M

year

1880 636951 773451

1881 643685 705236

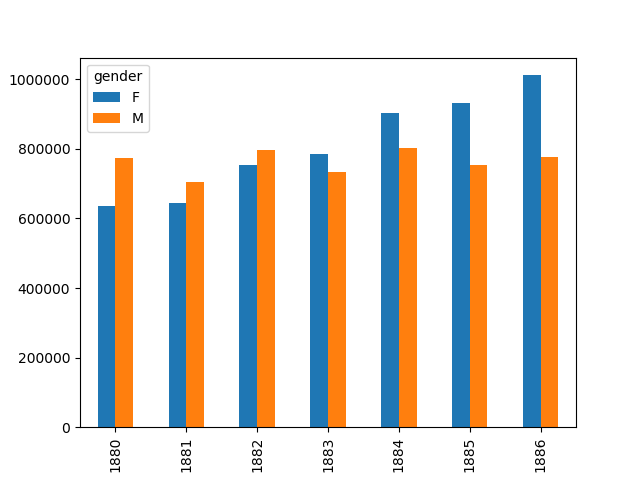
1882 754957 795809

**>>>** total\_births.plot(kind='bar')

<matplotlib.axes.\_subplots.AxesSubplot object at 0xa580b44c>

**>>>** plt.show()

>>>

[](https://pandasguide.readthedocs.io/en/latest/_images/plot1.png)

* Same can be achieved by using ‘groupby’ option as below,

>>>

**>>>** allData.groupby(['year', 'gender']).sum().unstack('gender').head(3)

birthcount

gender F M

year

1880 636951 773451

1881 643685 705236

1882 754957 795809

* Next, we want to check the ratio of the names with total number of names. For this, we can write a function, which calculates the ration and apply it to groupby option,

>>>

**>>>** *# calculate ratio*

**... def** add\_prop(group):

**...**  births = group.birthcount.astype(float)

**...**  group['prop'] = births/births.sum() *# add column prop*

**...**  **return** group

**...**

>>>

**>>>** names = allData.groupby(['year', 'gender']).apply(add\_prop)

**>>>** names.head()

name gender birthcount year prop

0 Mary F 7065 1880 0.011092

1 Anna F 2604 1880 0.004088

2 Emma F 2003 1880 0.003145

3 Elizabeth F 1939 1880 0.003044

4 Minnie F 1746 1880 0.00274