Data Science with Python: Pandas

Python Pandas

What is Pandas

- Pandas is a data science library aimed at quick and simplified data munging and exploratory analysis in Python.
- Specifically, it provides high-level data structures like the 'DataFrame' (similar to the R *data.frame*) and 'Series'.
- Additionally it has specialized methods for importing, manipulating and visualizing cross sectional and time series data.
- It is built on a solid foundation of NumPy arrays and is optimized for performance (pandas is about 15x faster)

Pandas Features

- Data structures with labeled axes that enable (automatic or explicit) data alignment
- Ability to handle both time-series and traditional data
- Facilities to add and remove columns on-the-fly
- Powerful management of missing data
- SQL-like joins (Merge, Append, Set Operations and other relational maneuvers)
- Methods for data I/O from/to various file formats like csv, Excel, HDFS, SQL databases
- *Reshaping* (long-to-wide, wide-to-long) and *Pivoting* (Excel-like)
- Label sub setting, fancy slicing
- A powerful "GroupBy" method that implements the *split-apply-combine* strategy operations
- Advanced time-series functions
- Hierarchical axis indexing (to work with high-dimensional data) in a lower-dimensional data structure

Overview of sections ahead

- Installation & importing Pandas
- Pandas Data Structures (Series & Data Frames)
- Creating Data Structures (Data import – reading into pandas)
- Data munging (Inspection, data cleaning, data manipulation, data analysis)
- Data visualizations
- Data export – writing from pandas

Installation of Pandas

Getting started: installing pandas

- Method 1:
  - Simply go to your command line tool and type

    pip install pandas
- Alternative – install **Anaconda**
  - Anaconda is a zero cost Python meta-distribution that includes 700+ popular Python packages for data science.

    conda install pandas

Importing pandas package

Import pandas as pd
From pandas import *

Pandas Data Structures

Pandas Data Structures

- 1-dimensional: Series
  - NumPy array subclass with item label vector (Index)
  - Both ndarray and dictionary-like
- 2-dimensional: DataFrame
  - Represents a dictionary of Series objects
  - Confirms Series to a common Index

Pandas Data Structures: Series

- What is Series-

  A Series is a one-dimensional array-like data structure containing a vector of data (of any valid NumPy type) and an associated array of data labels, called its index.
- About Indexes-
  - By default, if not specified, it is integer values: 0 to N-1
  - You can also specify your own indexes
  - When only passing a dictionary, the index in the resulting Series will have the dictionary's keys in sorted order.
  - There can be duplicate indexes.

Pandas Series

- Series can be created using tuple, list, dictionary, set and numpy array using series() function
- Creating a series:

  Series(numpy-array, index = [Generally a list object])

- If the user does not specify an index explicitly, a default one is created that consists of the natural integers 0 through N - 1 (N being the size of the series).
- Unlike the NumPy array, though, the index of a pandas *Series* could be a character vector or something else (other than integers.)

```
#Creating Series from a Numpy Array
x= pd.Series([21, 42, -31, 85], index=['d', 'b', 'a', 'c'])
print (x)

d     21
b     42
a    -31
c     85
dtype: int64
```

```
#Creating Series from a Dictionary
d = {'Delhi': 100, 'Nagpur': 120, 'Pune': 600, 'Mumbai': 700, 'Chennai': 450,
'Lucknow': None}
cities = pd.Series(d)
print (cities)

Chennai     450.0
Delhi       100.0
Lucknow       NaN
Mumbai      700.0
Nagpur      120.0
Pune        600.0
dtype: float64
```

A series can be converted into a list or a dictionary using methods like tolist() and to_dict()

Pandas Series - Attributes

- Just like attributes for primitive Python data structures like Lists or Dictionaries provide useful metadata about the contents of the structure, we can use Series attributes like **values, index, shape**

```
In []: series_2
Out[]:
a    34.0
b    60.0
c    21.0
d    22.0
e     7.0
Name: S2, dtype: float64
# Get the underlying NumPy array
In []: series_2.values
Out[]: array([ 34.,  60.,  21.,  22.,   7.])
```

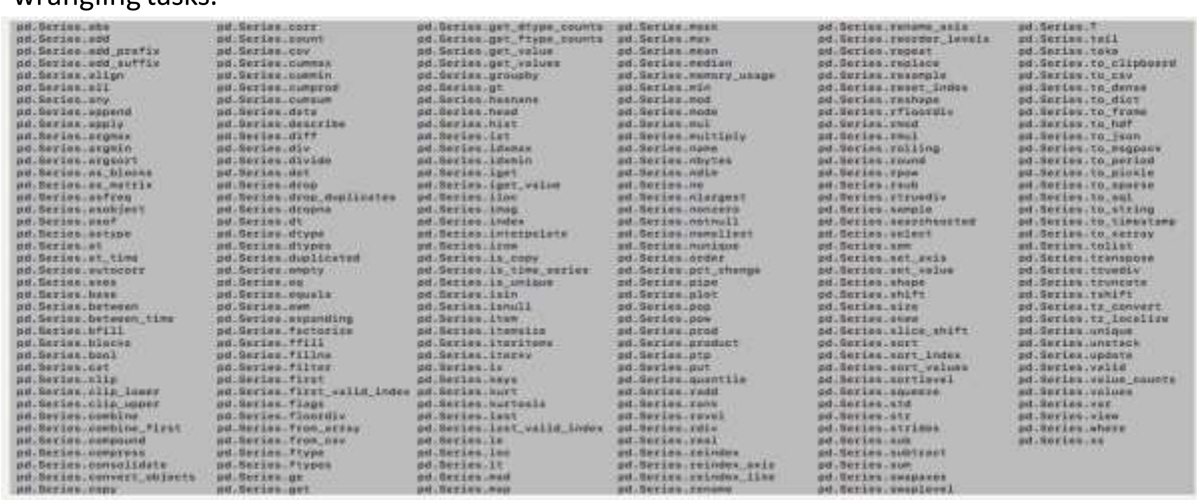
```
# Get the index
In []: series_2.index
Out[]: Index([u'a', u'b', u'c', u'd', u'e'],
dtype='object')

# Get the size on disk
In []: series_2.nbytes
Out[]: 40

# Get the number of elements
In []: series_2.shape
Out[]: (5,)
```

Pandas series: Subsetting

- There are many ways to extract subsets of a Series in Pandas. In addition to allowing NumPy-like subsetting using integer lists and slices, it is possible to subset a Series using
  - label-based indexing by passing index labels associated with the values
    - Single/list of labels
    - Slice of labels
    - Positional slicing
    - Reversing the series
  - Fancy indexing using methods like loc, iloc, ix, at, iat
    - .loc() for label based subsetting
    - .iloc() for integer based subsetting
    - .ix() and .at(), .iat() exist, but they serve the same purpose like loc and iloc
  - Boolean indexing for subsetting with logical arrays
    - boolean indexing works in the same way as it does for subsetting NumPy arrays. We create a boolean of the same length as the Series, (using the same Series), and then pass the booleanto the squre bracket subsetter

Pandas Series: Important Methods

There's a variety of other methods that are useful across the entire spectrum of data wrangling tasks.



Pandas Series – Data Wrangling Tasks

1. **Peaking the data: head and tail** are used to view a small sample of a Series or DataFrame object, use the **head() and tail()** methods. The default number of elements to display is five, but you may pass a custom number.
2. **Type Conversion: astype** explicitly convert dtypes from one to another
3. **Treating Outliers:**
    1. **clip_upper, clip_lower** can be used to clip outliers at a threshold value. All values lower than the one supplied to **clip_lower** , or higher than the one supplied to **clip_upper** will be replaced by that value.
    2. This function is especially useful in treating outliers when used in conjunction with .quantile()

       (Note: In data wrangling, we generally clip values at the 1st-99th Percentile (or the 5th-95th percentile))
4. **Replacing Values: replace** is an effective way to replace source values with target values by suppling a dictionary with the required substitutions
5. **Checking values belonging to a list: isin** produces a boolean by comparing each element of the Series against the provided list. It takes True if the element belongs to the list. This boolean may then be used for subsetting the Series.

Pandas Series – Data Wrangling Tasks

6. Finding uniques and their frequency: unique, nunique, value_counts

These methods are used to find the array of distinct values in a categorical Series, to count the number of distinct items, and to create a frequency table respectively.

7. Dealing with Duplicates: **duplicated**

Duplicated produces a boolean that marks every instance of a value after its first occurrence as True. **drop_duplicates** returns the Series with the duplicates removed. If you want to drop duplicated permanently, pass the inplace=True argument.

8. Finding the largest/smallest values: idxmax, idxmin, nlargest, nsmallest

As their names imply, these methods help in finding the largest, smallest, n-largest and n-smallest respectively. Note that the index label is returned with these values, and this can be especially helpful in many cases.

9. Sorting the data: sort_values , sort_index help in sorting a Series by values or by index. Note: that in order to make the sorting permanent, we need to pass an inplace=True argument.

Pandas Series – Data Wrangling Tasks

10. Mathematical Summaries: mean, median, std, quantile, describe are mathematical methods employed to find the measures of central tendency for a given set of data points. **quantile** finds the requested percentiles, whereas **describe** produces the summary statistics for the data.

11. Dealing with missing data: isnull, notnull are complementary methods that work on a Series with missing data to produce boolean Series to identify missing or non-missing values respectively. Note that both the NumPy **np.nan** and the base Python **None** type are identified as missing values

12. Missing values imputation: fillna, ffill and bfill, dropna This set of Series methods allow us to deal with missing data by choosing to either impute them with a particular value, or by copying the last known value over the missing ones (typically used in time-series analysis.) We may sometimes want to drop the missing data altogether and **dropna** helps us in doing that. (Note: It is a common practice in data science to replace missing values in a numeric variable by its mean (or median if the data is skewed) and in categorical variables with its mode

Pandas Series – Data Wrangling Tasks

13. Apply function to each element:

map is perhaps the most important of all series methods. It takes a general-purpose or user-defined function and applies it to each value in the Series. Combined with base Python's lambda functions, it can be an incredibly powerful tool in transforming a given Series.

This sounds like the **map** function for List objects in Base Python. The **.map()** method can be understood as a wrapper around that function

14. Visualizing the data:

The plot method is a gateway to a treasure trove of potential visualizations like histograms, bar charts, scatterplots, boxplots and more.

Pandas Data Structures: DataFrame

- It is 2-dimensional table-like data structure/
- It is fundamentally different from NumPy 2D arrays in that here each column can be a different dtype
  - Has both a row and column index for
    - Fast lookups
    - Data alignment and joins
  - Is operationally identical to the R *data.frame*
  - Can contain columns of different data types
  - Can be thought of a dictionary of Series objects.
  - Has a number of associated methods that make commonplace tasks like merging, plotting etc. very straightforward

Pandas DataFrame

- **Creating a DataFrame**: We can create Data frame from multiple ways.
  1. From a dictionary of arrays or lists or from NumPy 2D arrays

     Syntax: DataFrame(data=, index=, columns=)

     As was the case with Series, if the index and the columns parameters are not specified, default numeric sequences running from 0 to N-1 will be used.
  2. From importing external data using different functions
  3. Connecting data bases using different functions
- **Creating a DataFrame from 2D Array**

```
my_df = pd.DataFrame(np.arange(20, 32).reshape(3, 4),\
columns = ['c1', 'c2', 'c3', 'c4'],\
index = list('abc'))
print (my_df)
```

```
   c1  c2  c3  c4
a  20  21  22  23
b  24  25  26  27
c  28  29  30  31
```

Pandas DataFrame

- **Creating DataFrame from a dictionary**

  -The simplest way of creating a pandas *DataFrame* is using a Python dictionary of arrays/lists.
  -The keys of the dictionary will be utilized as column names, and a list of strings can be provided to be utilized as the index.
  -As with Series, if you pass a column that isn't contained in data, it will appear with NaN values in the result.

```
# creating DataFrame using a dict of equal length lists
my_dict = {'ints': np.arange(5),\
'floats': np.arange(0.1, 0.6, 0.1),\
'strings': list('abcde')}
my_dict
```

```
{'floats': array([ 0.1,  0.2,  0.3,  0.4,  0.5]),
 'ints': array([0, 1, 2, 3, 4]),
 'strings': ['a', 'b', 'c', 'd', 'e']}
```

```
my_df2 = pd.DataFrame(my_dict, index = list('vwxyz'),\
                      columns=['floats','ints','strings','p','q'])
print (my_df2)
```

```
   floats  ints strings    p    q
v     0.1     0       a  NaN  NaN
w     0.2     1       b  NaN  NaN
x     0.3     2       c  NaN  NaN
y     0.4     3       d  NaN  NaN
z     0.5     4       e  NaN  NaN
```

Reading External Data into Pandas

Reading Data into Pandas

- **Importing structured data (Cross Sectional & Time Series Data)**
  - CSV and Flat file
  - Excel
  - Databases (Sql Server, Oracle, Postgre sql, Teradata etc.)
  - FTP location/Web location
  - HDFS (Hadoop Distributed File system)
- **Importing Semi structured data**
  - JSON file
  - XML file
- **Importing Unstructured data**
  - Text data
  - Images
  - Audio & Video files
- **Importing Data from API's**
  - Twitter
  - Facebook
  - Scrapping data from website url

Read a CSV or Flat file

We read a csv using read_csv() function.

Syntax: read_csv('file path', <options>)

Important options available –

- **Delimiter** -Delimiter to use.
- **Header** - Row number(s) to use as the column names, and the start of the data. If header=None, then no name passed.
- **Skiprow**-number of lines to skip (int) at the start of the file
- **Names**- LIST of column names to use
- **Nrows**- *int, default None*Number of rows of file to read. Useful for reading pieces of large files

```
#reading from a csv
my_sales=pd.read_csv("sales_data.csv")

type(my_sales)
```

```
pandas.core.frame.DataFrame
```

```
my_sales.head()
```

For general delimited file (**Flat File**) we can also use **read_table**() function. that **has same syntax** as **read_csv()**

read_table is read_csv withsep=',' replaced by sep='\t', they are two thin wrappers around the same function so the performance will be identical

	Customer_id	Customer_name	Subsegment	City	Division	Category	Version	Sales_amount	No_of_Licences	Sales_Date
0	129	C1	Lower Mid-Market	Chennai	RSD9	RSD9_RSC3	2003	58,719	37	3/8/2008
1	419	C2	Upper Mid-Market	Delhi	RSD9	RSD9_RSC5	2002_V2	16,944	12	11/25/2008

Read an Excel file: part 1

Method 1:

```
#reading an excel file
#step 1: load excel FILE using ExcelFile() function
my_bank_excel=pd.ExcelFile("Bank_Accounts.xlsx")
my_bank_excel
```

```
<pandas.io.excel.ExcelFile at 0x1541de2ef0>
```

```
#step 2: load excel SHEET using read_excel() function
my_bank_sheet=pd.read_excel(my_bank_excel,sheetname='Bank Account Details')
my_bank_sheet.head()
```

	Zone_Name	Branch_Name	Branch_Code	Sales_Rep_Code	Sales_Rep_Name	In
0	CENTRAL 1	MAIN BRANCH	1	01005XE	Christopher	84
1	CENTRAL 1	MAIN BRANCH	1	01005CB	George	25

Method 2:

```
#we can directly use read_excel()
new_bank_sheet=pd.read_excel("Bank_Accounts.xlsx",sheetname='Bank Account Details')
new_bank_sheet.head()
```

	Zone_Name	Branch_Name	Branch_Code	Sales_Rep_Code	Sales_Rep_Name	Inactive_A
0	CENTRAL 1	MAIN	1	01005XE	Christopher	84

Read an Excel File: part 2

Read_excel() options-

- **Sheetname** – possible values can be
  - Defaults to 0 -> 1st sheet as a DataFrame
  - 1 -> 2nd sheet as a DataFrame
  - "Sheet1" -> 1st sheet as a DataFrame
  - [0,1,"Sheet5"] -> 1st, 2nd & 5th sheet as a dictionary of DataFrames
  - None -> All sheets as a dictionary of DataFrames

- **header** *: int, list of ints, default 0*
  Row (0-indexed) to use for the column labels

- **names** *: array-like, default None*
  List of column names to use. If file contains no header row, then you should explicitly pass header=None

For more - http://pandas.pydata.org/pandas-docs/version/0.18.1/generated/pandas.read_excel.html

Read a JSON files

- To read a json file we have function read_json() that converts a json string into pandas object

  Syntax: read_json(*path_or_buf=None, <other options>)*

  - **path_or_buf** *: a valid JSON string or file-like, default: None.* The string could be a URL
  - Other options -[http://pandas.pydata.org/pandas-](http://pandas.pydata.org/pandas-docs/version/0.19.2/generated/pandas.read_json.html) docs/version/0.19.2/generated/pandas.read_json.html

sales_data.json - Notepad

File Edit Format View Help

{"Customer_id":{"0":129,"1":419,"2":270
9,"112":487,"113":220,"114":314,"115":2
14":361,"215":348,"216":140,"217":195,"

Our sample file: **sales_data.json**

```
#read a json file
sales_data_json = pd.read_json("sales_data.json")
sales_data_json.head(5)
```

	Category	City	Customer_id	Customer_name	Division	No_of_Licences	Sales_Date	Sales_amount	Subsegment
0	RSD9_RSC3	Chennai	129	C1	RSD9	37	3/8/2008	58,719	Lower Mid-Market
1	RSD9_RSC5	Delhi	419	C2	RSD9	12	11/25/2008	16,944	Upper Mid-

Data Munging

Data Munging

Once you read data into a pandas object(mostly a DataFrame), you will perform various operations that include-

Inspect data –

- Checking attributes –index, values, row labels, column labels, data types, shape, info etc
- Check –
  - If a value exists
  - Containing missing values
- Descriptive statistics on your data – mean, median, mode, skew, kurtosis, max, min, sum, std, var, mad, percentiles, count etc

Clean data / Manipulate

- Mutation of table (Adding/deleting columns)
- Renaming columns or rows
- Binning data
- Creating dummies from categorical data
- Type conversions
- Handling missing values – detect, filter, replace
- Handling duplicates
- Slicing of data – sub setting
- Handling outliers
- Sorting – by data, index
- Table manipulation-
  - Aggregation – Group by processing
  - Merge, Join, Concatenate
  - Reshaping & Pivoting data – stack/unstack, pivot table, summarizations
  - Standardize the variables

Data Analysis

- Univariate Analysis (Distribution of data, Data Audit)
- Bi-Variate Analysis (Statistical methods, Identifying relationships)
- Simple & Multivariate Analysis

Detail steps of data manipulation

- Understand the data
- Sub Setting Data or Filtering Data or Slicing Data
  - Using [] brackets
  - Using indexing or referring with column names/rows
  - Using functions
- Dropping rows & columns
- Mutation of table (Adding/deleting columns)
- Binning data (Binning numerical variables in to categorical variables using cut() and qcut() functions)
- Renaming columns or rows
- Sorting
  - by data/values, index
  - By one column or multiple columns
  - Ascending or Descending
- Type conversions
- Setting index
- Handling duplicates
- Handling missing values – detect, filter, replace
- Handling outliers
- Creating dummies from categorical data (using get_dummies())
- Applying functions to all the variables in a data frame (broadcasting)

Detail steps of data manipulation

- Table manipulation-
  - Aggregation – Group by processing, Pivot Tables
  - Reshaping & Pivoting data – stack/unstack, pivot table, summarizations
  - Merge/ Join (left, right, outer, inner)
  - Concatenate (appending)– Binding or stacking or union all
  - Standardize the variables
- Random Sampling (with replacement/with out replacement)
- Handling Time Series Data
- Handling text data
  - -with functions
  - -with Regular expressions

Inspect Data

Checking attributes: Series

- **Attributes of a Series**-These include **.values** and **.index**, using which we can get the array representation and index object of the Series respectively.

```
In [18]: my_series = pd.Series(np.random.randn(5))
         my_series.values
Out[18]: array([-1.29650542, -0.79977866, -0.90396059, -0.10252797,  0.53042503])

In [19]: my_series.index
Out[19]: RangeIndex(start=0, stop=5, step=1)
```

- We can assign a name to the Series using the **.index.name**

```
In [20]: my_series.index.name = 'row.names'
         my_series.index
Out[20]: RangeIndex(start=0, stop=5, step=1, name='row.names')
```

Checking attributes: DataFrame

- Attributes of a DataFrame: important ones are **index, columns, dtypes, info**

Our Data -

```
# Create a DataFrame using a dict of lists
data = {'country': ['BE', 'FR',\
                    'GR', 'NL', 'UK'],
'population': [11.3, 64.3, 81.3, 16.9, 64.9],
'area': [30510, 671308, 357050, 41526, 244820],
'capital': ['Brussels', 'Paris',\
            'Berlin', 'Amsterdam', 'London']}
countries = pd.DataFrame(data,\
                         index=list('abcde'))
print (countries)

     area    capital country  population
a   30510   Brussels      BE        11.3
b  671308      Paris      FR        64.3
c  357050     Berlin      GR        81.3
d   41526  Amsterdam      NL        16.9
e  244820     London      UK        64.9
```

```
# Check row names
countries.index

Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

# Check column names
countries.columns

Index(['area', 'capital', 'country', 'population'], dtype='object')

# Check data types
countries.dtypes

area            int64
capital        object
country        object
population    float64
dtype: object

# Gives overview of the dataset
countries.info

<bound method DataFrame.info of       area    capital country  population
0   30510   Brussels      BE        11.3
1  671308      Paris      FR        64.3
2  357050     Berlin      GR        81.3
3   41526  Amsterdam      NL        16.9
4  244820     London      UK        64.9>
```

Check for – value, missing values: Series

- To check if an item exists: use **'in'** keyword

```
print ( my_series)

a     50
b     55
c     60
d     65
e     70
dtype: int32
```

```
#to check value
50 in my_series.values

True
```

```
#to check value
100 in my_series.values

False
```

```
#to check index
'a' in my_series

True
```

```
#to check index
'q' in my series

False
```

- To Detect if there is any missing data: missing data in Pandas appears as NaN(Not a number), and to detect them we use- **isnull()** and **notnull()** functions.

```
index2 = ['a', 'd', 'e', 'f', 'g']
my_series2 = my_series[index2];
print (my_series2)

a      50.0
d      65.0
e      70.0
f       NaN
g       NaN
dtype: float64
```

```
# isnull() function:
my_series2.isnull()
# or pd.isnull(my_series2)

a       True
d       True
e       True
f      False
g      False
dtype: bool
```

```
# notnull() function
my_series2.notnull()
#or pd.notnull(my_series2)

a       True
d       True
e       True
f      False
g      False
dtype: bool
```

Pandas Descriptive Statistics: Numercal Data

- Pandas objects have a set of common math/stat methods that extract
  - a single summary value from a Series
  - a Series of summary values by row/column from a DataFrame (along a specified axis)

count	sum	mean	median
min/max	skew	kurt	cumsum

- When these methods are called on a DataFrame, they are applied over each row/column as specified and results collated into a Series.
- Missing values are ignored by default by these methods. Pass skipna=False to disable this.

Pandas Descriptive Statistics: Numercal Data

Our Data ->

```
print (myFrame)

    v   w   x   y   z
a   0   1   2   3   4
b   5   6   7   8   9
c  10  11  12  13  14
d  15  16  17  18  19
e  20  21  22  23  24
```

```
# Getting colsums
myFrame.sum()

v    50
w    55
x    60
y    65
z    70
dtype: int64
```

```
# Find the min/max for each row
myFrame.min(axis=1)

a     0
b     5
c    10
d    15
e    20
dtype: int32
```

```
# For rowsums, pass axis=1
myFrame.sum(axis=1)

a     10
b     35
c     60
d     85
e    110
```

```
# Find the min/max for each column
myFrame.min(axis=0)

v    0
w    1
x    2
y    3
z    4
dtype: int32
```

```
# Find the location of the min value across rows
myFrame.idxmin()

v    a
w    a
x    a
y    a
z    a
dtype: object
```

Pandas Descriptive Statistics: Numercal Data

- **Describe()** function -It works on numeric Series and produces the summary statistics including – min, max, count, mean, standard deviation, median and percentiles (25th, 75th)
- You can call describe on a Series (a column in a DataFrame) or an entire DataFrame (in which case it will produce results for each **numeric** column.)

```
#works on Numeric columns to provide numeric statistics
myFrame.describe()
```

	v	w	x	y	z
count	5.000000	5.000000	5.000000	5.000000	5.000000
mean	10.000000	11.000000	12.000000	13.000000	14.000000
std	7.905694	7.905694	7.905694	7.905694	7.905694
min	0.000000	1.000000	2.000000	3.000000	4.000000
25%	5.000000	6.000000	7.000000	8.000000	9.000000
50%	10.000000	11.000000	12.000000	13.000000	14.000000
75%	15.000000	16.000000	17.000000	18.000000	19.000000
max	20.000000	21.000000	22.000000	23.000000	24.000000

Pandas Descriptive Statistics: Categorical Data

Pandas has some interesting methods for working on Categorical data. These include functions for getting unique values (**unique**), frequency ta bles (**value_counts**), membership (**isin**).

```
#create a series
mySeries = pd.Series(list('the quick brown\
fox jumped over the lazy dog'))

#get distinct values in the Series
print (mySeries.unique())
```

```
['t' 'h' 'e' ' ' 'q' 'u' 'i' 'c' 'k' 'b' 'r' 'o' 'w' 'n' 'f' 'x' 'j' 'm'
 'p' 'd' 'v' 'l' 'a' 'z' 'y' 'g']
```

```
# Getting a Frequency Table
print (mySeries.value_counts())
```

```
     8
e    4
o    4
h    2
d    2
t    2
u    2
r    2
m    1
v    1
```

```
# isin returns a boolean,
#indicating the position where a match occurred
colours = pd.Series(['red', 'blue', \
                     'white', 'green', \
                     'black', 'white', None])
colours.isin(['white'])
```

```
0    False
1    False
2     True
3    False
4    False
5     True
6    False
dtype: bool
```

Pandas Descriptive Statistics: Categorical Data

- Calling the **describe()** function on categorical data returns summary information about the Series that includes the
  - count of non-null values,
  - the number of unique values,
  - the mode of the data
  - the frequency of the mode

```
colours.describe()
```

```
count         6
unique        5
top       white
freq          2
dtype: object
```

Clean & Data Manipulation

Adding and Deleting Columns: DataFrame

- Adding Columns:
  New columns can be added or derived from existing columns

```
my_df2['const'] = np.pi
my_df2
```

	floats	ints	strings	p	q	const
v	0.1	0	a	NaN	NaN	3.141593
w	0.2	1	b	NaN	NaN	3.141593
x	0.3	2	c	NaN	NaN	3.141593
y	0.4	3	d	NaN	NaN	3.141593
z	0.5	4	e	NaN	NaN	3.141593

To delete column we have 2 options –

- del
- drop

```
del my_df2['p']
my_df2
```

	floats	ints	strings	q	const
v	0.1	0	a	NaN	3.141593

```
# delete rows
my_df2.drop(['x', 'y'])
```

	floats	ints	strings	q	const
v	0.1	0	a	NaN	3.141593
w	0.2	1	b	NaN	3.141593

```
# delete columns
my_df2.drop(['const', 'q'], axis=1)
```

	floats	ints	strings
v	0.1	0	a

Pandas - Handling missing values

- Important functions concerning missing values are-
  - Detect missing values : isnull(), notnull()
  - Filter out missing values: dropna()
  - Replace missing values: fillna()

Detect missing values-

Our data -

```
print (mySeries)

0      abc
1      pqr
2      NaN
3      xyz
4      NaN
5      ijk
6     None
dtype: object
```

```
#Return like-type object containing boolean values,
#indicating which values are missing / NA
mySeries.isnull()

0    False
1    False
2     True
3    False
4     True
5    False
6     True
dtype: bool
```

```
#Return like type object containing boolean values,
# indicating which values are NOT missing
mySeries.notnull()

0     True
1     True
2    False
3     True
4    False
5     True
6    False
dtype: bool
```

Pandas - Handling missing values

Filter missing values: dropna()

Our Data -

```
print (myFrame)

     0    1    2    3
0  1.0  6.5  3.0  NaN
1  1.0  NaN  NaN  NaN
2  NaN  NaN  NaN  NaN
3  NaN  6.5  3.0  NaN
```

```
# By default it will drop all ROWS,
# with any NaN value
myFrame.dropna()
```

	0	1	2	3

```
#to drop ROWS with ALL missing values
myFrame.dropna(how='all')
```

	0	1	2	3
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
#to drop COLUMNS with ALL missing values
myFrame.dropna(how='all',axis=1)
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
#to drop COLUMNS with ANY missing values
myFrame.dropna(how='any',axis=1)
```

0
1
2
3

```
#retain ROWS with,
# at least 2 NON missing values
myFrame.dropna(thresh=2)
```

	0	1	2	3
0	1.0	6.5	3.0	NaN
3	NaN	6.5	3.0	NaN

Pandas - Handling missing values

Replacing missing value: fillna()

Our data -

```
print (myFrame)
```

```
     0    1    2    3
0  1.0  6.5  3.0  NaN
1  1.0  NaN  NaN  NaN
2  NaN  NaN  NaN  NaN
3  NaN  6.5  3.0  NaN
```

```
#pass your own value
myFrame.fillna(0)
```

	0	1	2	3
0	1.0	6.5	3.0	0.0
1	1.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	6.5	3.0	0.0

```
#you can use dictionary to fill diff value for each column
myDict=dict( { 0:10, 1:20, 2:30, 3:40} )
myFrame.fillna(myDict)
```

	0	1	2	3
0	1.0	6.5	3.0	40.0
1	1.0	20.0	30.0	40.0
2	10.0	20.0	30.0	40.0
3	10.0	6.5	3.0	40.0

Pandas removing duplicates

duplicated() method

Returns boolean Series denoting duplicate rows, optionally only considering certain columns

	C1	C2
0	A	1
1	B	2
2	C	4
3	A	3
4	B	2
5	C	4

```
# Creates a boolean series to indicate which rows have duplicates
df.duplicated()
```

```
0    False
1    False
2    False
3    False
4     True
5     True
dtype: bool
```

```
# Retain the rows that have duplicates
df[df.duplicated()]
```

	C1	C2
4	B	2
5	C	4

Pandas removing duplicates

Drop_duplicates()

- Returns DataFrame with duplicate rows removed, optionally only considering certain columns
- By default, this methods consider all of the columns. To specify a subset for detecting duplicates, use-
  df.drop_duplicates(['list-of-columns'])

	C1	C2
0	A	1
1	B	2
2	C	4
3	A	3
4	B	2
5	C	4

```
# Retain the first occurrence of each row (drop dups)
df.drop_duplicates()
```

	C1	C2
0	A	1
1	B	2
2	C	4
3	A	3

```
# Retain the last occurrence of each row (drop dups)
df.drop_duplicates(take_last=True)
```

	C1	C2
0	A	1
3	A	3
4	B	2
5	C	4

```
df.drop_duplicates(['C2'])
```

	C1	C2
0	A	1
1	B	2
2	C	4
3	A	3

Slicing: Series

- Subsetting a Series: Slicing operations

```
# note: arange function is similar to in-built range() function
# and is used to create an Numpy Array
my_series = pd.Series(np.arange(50, 71, 5), index = list('abcde'))
print (my_series)
```

```
a    50
b    55
c    60
d    65
e    70
dtype: int32
```

```
In [23]: # slice using index label
         print (my_series['a'])
Out[23]: a    50
         b    55
         c    60
         d    65
         e    70
         dtype: int32
```

```
In [25]: # slicing using a list of labels
         print (my_series[['a', 'c', 'e']])
         a    50
         c    60
         e    70
         dtype: int32
```

```
In [26]: # positional slicing
         print (my_series[0:3])
         a    50
         b    55
         c    60
         dtype: int32
```

```
In [27]: # slicing using a boolean
         print (my_series[my_series > 60])
         a    50
         b    55
         c    60
         d    65
         e    70
         dtype: int32
```

Slicing: Series

- Subsetting a Series(contd): indexes in Series need not be unique.

```
mySeries=pd.Series(np.arange(10,20,2),index=list('aaabb'))
print (mySeries)
```

```
a     10
a     12
a     14
b     16
b     18
dtype: int32
```

```
mySeries['a']
```

```
a     10
a     12
a     14
dtype: int32
```

To read any specify observation, use 'ix' function.
(**More on this later)**

```
mySeries['a'].ix[2]
```

```
14
```

Slicing: DataFrame

- Subsetting: slicing operations

Selecting 1 column

```
countries['area'] #or countries.area
```

```
a      30510
b     671308
c     357050
d      41526
e     244820
Name: area, dtype: int64
```

Selecting 2 or more column

```
countries[ ['area','population'] ]
```

	area	population
a	30510	11.3
b	671308	64.3
c	357050	81.3
d	41526	16.9
e	244820	64.9

Slicing: DataFrame

- **Advanced slicing: using ix()**

  Syntax: df.ix[<specify-rows>, <specify-cols>]

- Specify-cols could be –
  - a singular/list/splice of column name(s)
  - integer ranges (splices).

Specify-rows can be done using –
indices (if you want to subset rows by name)
Integer splices (if you want to subset by position)

- **The columns returned when indexing a DataFrame is a view on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the original DataFrame. The column can be explicitly copied using the Series copy method**

```
countries.ix['FR':'GR', :]
```

	area	capital	population
country			
FR	671308	Paris	64.3
GR	357050	Berlin	81.3

```
countries.ix[:, 0:2]
```

	area	capital
country		
BE	30510	Brussels
FR	671308	Paris
GR	357050	Berlin
NL	41526	Amsterdam
UK	244820	London

```
countries.ix[countries['population']>5, ['capital', 'area']]
```

	capital	area
country		
BE	Brussels	30510
FR	Paris	671308
GR	Berlin	357050
NL	Amsterdam	41526
UK	London	244820

```
countries.ix[[0,3], 0:2]
```

	area	capital
country		
BE	30510	Brussels
NL	41526	Amsterdam

Array operations

- **Array Operations :** Array or Vectorized operations will preserve the index-value links.

```
print (my_series)
a    50
b    55
c    60
d    65
e    70
dtype: int32

In [28]: print (my_series * 2)
a    100
b    110
c    120
d    130
e    140
dtype: int32

In [29]: print (np.sqrt(my_series))
a    7.071068
b    7.416198
c    7.745967
d    8.062258
e    8.366600
dtype: float64
```

```
print (myFrame)
    v   w   x   y   z
a   0   1   2   3   4
b   5   6   7   8   9
c  10  11  12  13  14
d  15  16  17  18  19
e  20  21  22  23  24

myFrame['v']=myFrame['v']*2
print (myFrame)
    v   w   x   y   z
a   0   1   2   3   4
b  10   6   7   8   9
c  20  11  12  13  14
d  30  16  17  18  19
e  40  21  22  23  24
```

Pandas Sorting: Series

Our data -

```
# Create a Series with explicit index
mySeries = pd.Series(np.random.randn(5), index=list('dcbae'));
print (mySeries)
```

```
d     0.163111
c    -1.199557
b    -0.642723
a     0.891565
e     0.182959
dtype: float64
```

To sort on index: sort_index()

```
# Sorting on the index
mySeries.sort_index()
```

```
a     0.891565
b    -0.642723
c    -1.199557
d     0.163111
e     0.182959
dtype: float64
```

To sort on values: sort_values()

```
#sorting by values
mySeries.sort_values(ascending=False)
```

```
a     0.891565
e     0.182959
d     0.163111
b    -0.642723
c    -1.199557
dtype: float64
```

Pandas Sorting: DataFrame

Our data -

```
myFrame=pd.DataFrame(np.random.randn(9).reshape(3,3),\
                     index=list('cba'),\
                     columns=list('prq'))
print (myFrame)
```

```
          p         r         q
c -0.500858 -1.785182  0.948032
b -0.784855 -0.874356 -0.913320
a -1.347760  0.284159 -0.390920
```

Sort by row or column index: sort_index()

```
#without arguments, sort_index() will
#sort the index (rows) of the DataFrame
myFrame.sort_index()
```

	p	r	q
a	0.848088	-2.011534	-0.315217
b	0.105489	-1.167543	-0.167860
c	-0.610992	-1.381862	-0.454777

```
# To sort column names: axis=1
myFrame.sort_index(axis=1)
```

	p	q	r
c	-0.610992	-0.454777	-1.381862
b	0.105489	-0.167860	-1.167543
a	0.848088	-0.315217	-2.011534

Sort by values: sort_values()

```
# Sort the data by the values of a column
myFrame.sort_values(by='p')
```

	p	r	q
a	-1.347760	0.284159	-0.390920
b	-0.784855	-0.874356	-0.913320
c	-0.500858	-1.785182	0.948032

```
# Sort the data by the values of 2 columns
myFrame.sort_values(by=['p', 'r'], ascending=False)
```

	p	r	q
c	-0.500858	-1.785182	0.948032
b	-0.784855	-0.874356	-0.913320
a	-1.347760	0.284159	-0.390920

Pandas Groupby processing

It involves following steps-

- **Split**
  - A DataFrame can be split up by rows(axis=0) or columns(axis=1) into groups.
  - We use pd.groupby() to create a groupby object
- **Apply**
  - A function is applied to each group using .agg() or .apply()
- **Combine**
  - The results of applying functions to groups are put together into an object
  - Note: Data types of returned objects are handled gracefully by pandas

Our dataset is a DataFrame with 100 rows and 4 columns – k1, k2, v1,v2.
k1,k2 = categorical data
v1,v2 = numerical data

```
df = pd.DataFrame({'k1': list('abcd' * 25),
'k2': list('xy' * 25 + 'yx' * 25),
'v1': np.random.rand(100),
'v2': np.random.rand(100)})
print (df.head(10))
```

```
  k1 k2        v1        v2
0  a  x  0.172171  0.793763
1  b  y  0.501872  0.553222
2  c  x  0.010050  0.185891
3  d  y  0.849452  0.582324
4  a  x  0.381330  0.052616
5  b  y  0.164315  0.503026
6  c  x  0.536370  0.080457
7  d  y  0.349445  0.718603
8  a  x  0.269342  0.559472
9  b  y  0.200403  0.941988
```

Snapshot of first 10 rows

Pandas Groupby processing

Grouping by 1 key

This results in a summarized data frame indexed by levels of the key

```
#grouping by 1 key
# Since k1 has 4 categories, this will return 4 rows
df.groupby(df['k1']).mean()
```

	v1	v2
k1		
a	0.465451	0.475523
b	0.376174	0.538208
c	0.506546	0.471130
d	0.568742	0.503672

```
# Since k2 has 2 categories, this will return 2 rows
df.groupby(df['k2']).sum()
```

	v1	v2
k2		
x	21.438036	24.937493
y	26.484775	24.775846

Grouping by 2 keys

This will result in a summarized data frame with a hierarchical index.

```
#grouping by 2 keys
# A dataframe with a hierarchical index,
# formed by a combination of the levels
df.groupby([df['k1'], df['k2']]).sum()
```

		v1	v2
k1	k2		
a	x	4.893579	6.656481
	y	6.742693	5.231588
b	x	4.485687	5.750269
	y	4.918652	7.704940
c	x	4.800692	5.894614
	y	7.862947	5.883645
d	x	7.250077	6.636128
	y	6.960484	5.955674

Pandas Groupby processing

Column-wise aggregation – optimal statistical method

```
# Summing a Series
df['v1'].groupby(df['k1']).sum()
```

```
k1
a     11.636272
b      9.404339
c     12.663639
d     14.218561
Name: v1, dtype: float64
```

```
# Summing all Series of a DataFrame
df.groupby(df['k2']).sum()
```

	v1	v2
k2		
x	21.438036	24.937493
y	26.484775	24.775846

Can also be done using agg()

```
# Summing a Series using agg()
df['v1'].groupby(df['k1']).agg('sum')
```

```
k1
a     11.636272
b      9.404339
c     12.663639
d     14.218561
Name: v1, dtype: float64
```

Pandas Groupby processing

- **agg() function** -When we have a groupBy object, we may choose to apply 1 or more functions to one or more columns, even different functions to individual columns. The **.agg() method** allows us to easily and flexibly specify these details.
- It takes as arguments the following –
  - list of function names to be applied to all selected columns
  - tuples of (colname, function) to be applied to all selected columns
  - dict of (df.col, function) to be applied to each df.col

Apply 1 function to All selected columns by passing names of functions to agg() as a list

```
#Apply min, mean, max and max to v1 grouped by k1
df['v1'].groupby(df['k1']).agg(['min', 'mean', 'max'])
```

	min	mean	max
k1			
a	0.027424	0.499384	0.989610
b	0.077218	0.506872	0.959979
c	0.003850	0.499564	0.967212
d	0.008391	0.530888	0.928070

```
# Apply min and max to all numeric columns of df grouped by k2
df[['v1', 'v2']].groupby(df['k2']).agg(['min', 'max'])
```

	v1		v2	
	min	max	min	max
k2				
x	0.003850	0.989610	0.037857	0.989309
y	0.008391	0.967212	0.007497	0.996997

Pandas Groupby processing

agg() function contd

We can supply names for the columns in the (new) aggregated DataFrame to the agg() method, in a **list of tuples**

```
# Provide names for the aggregated columns
df[['v1', 'v2']].groupby(df['k1']).agg([('smallest','min'),\
                                         ('largest', 'max')])
```

	v1		v2	
	smallest	largest	smallest	largest
k1				
a	0.027424	0.989610	0.113730	0.984053
b	0.077218	0.959979	0.045005	0.996997
c	0.003850	0.967212	0.007497	0.955553
d	0.008391	0.928070	0.037857	0.992447

```
# Apply max and min to v1; and mean and sum to v2; all grouped by k1
df[['v1', 'v2']].groupby(df['k1']).agg({'v1': [('large','max'),
                                       'min'], 'v2': ['mean','sum']})
```

	v1		v2	
	large	min	mean	sum
k1				
a	0.924970	0.052500	0.475523	11.888068
b	0.902905	0.016180	0.538208	13.455210
c	0.978955	0.010950	0.471130	11.778259
d	0.993237	0.083124	0.503672	12.591802

We can supply DataFrame column names and which functions to apply to each**, in a dictionary**

Pandas Groupby processing

- Apply() method- This method takes as argument the following:
  - a general or **user defined function**
  - any other **parameters** that the function would take

```
# Retrieve the top N cases from each group
def topN(data, col, N):
    return data.sort(columns=col, ascending=False).ix[:, col].head(5)
df.groupby(df['k2']).apply(topN, col='v1', N=5)
k2
x    28    0.989610
     48    0.973469
     57    0.959979
     34    0.925725
     91    0.910017
y    70    0.967212
     82    0.934823
     27    0.928070
     25    0.923844
     39    0.891839
Name: v1, dtype: float64
```

Pandas Merge

- The **merge()** function in pandas is similar to the SQL join operations;
- It links rows of tables using one or more keys
- **Syntax:**

```
merge(df1, df2,
how='left', on='key', left_on=None, right_on=None,
left_index=False, right_index=False,
sort=True, copy=True,
suffixes=('_x', '_y'))
```

Pandas Merge

The syntax includes specifications of the following arguments:

- **Which column to merge on;**
  - the on='key' if the same key is in the two DFs,
  - or left_on='lkey', right_on='rkey' if the keys have different names in the DFs

  Note: To merge on multiple keys, pass a list of column names
- **The nature of the join;**
  the how= option, with left, right, outer
  By **default**, the merge is an inner join
- Tuple of string values to append to **overlapping column names** to identify them in the merged dataset
  - the suffixes= option
  - **defaults** to ('_x', '_y')
- If you wish **to merge on the DataFrame index**,
  - pass left_index=True or right_index=True or both.
- Whether to **Sort the result DataFrame by the join keys** in lexicographical order or not;
  - The sort= option;
  - Defaults to True, setting to False will improve performance substantially in many cases

Pandas Merge

- Datasets used-

df1

	data1	key
0	-0.783222	b
1	0.235611	b
2	1.699517	a
3	-0.428308	c
4	-1.256941	a
5	0.037266	a
6	-1.456009	b

df2

	data2	key
0	0.291307	a
1	1.793256	b
2	-1.967771	d

df3

	data3	lkey
0	0.412956	b
1	-0.452773	b
2	0.588230	a
3	-0.002321	c
4	0.355012	a
5	0.855518	a
6	1.112227	b

df4

	data4	rkey
0	-0.157451	a
1	-0.477377	b
2	-0.808517	d

Pandas Merge

Default Merge with No Parameters

```
pd.merge(df1, df2)
```

	data1	key	data2
0	-0.783222	b	1.793256
1	0.235611	b	1.793256
2	-1.456009	b	1.793256
3	1.699517	a	0.291307
4	-1.256941	a	0.291307
5	0.037266	a	0.291307

Note that
- It is an inner join by default (output key is the intersection of input keys)
- Merge happens on the column 'key' which is common to both datasets;
- We could've written **pd.merge(df1, df2, on='key')** to the same effect

Specifying Which Columns To Merge On (If Keys Have Different Names In Datasets)

```
# still an inner join!
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

	data3	lkey	data4	rkey
0	0.412956	b	0.477377	b
1	-0.452773	b	-0.477377	b
2	1.112227	b	-0.477377	b
3	0.588230	a	0.157451	a
4	0.355012	a	-0.157451	a
5	0.855518	a	-0.157451	a

Pandas Merge

Specifying which type of join

```
# the merged dataset will have a union of the keys,
# imputing NaNs where values aren't found
pd.merge(df1, df2, how='outer')
```

	data1	key	data2
0	0.783222	b	1.793256
1	0.235611	b	1.793256
2	1.456009	b	1.793256
3	1.699517	a	0.291307
4	1.256941	a	0.291307
5	0.037266	a	0.291307
6	0.428308	c	NaN
7	NaN	d	-1.967771

Specifying suffixes

```
# Add a column with the same name to df1 and df2
df1['colx'] = np.random.randn(7)
df2['colx'] = np.random.randn(3)
# Specifying suffixes to identify columns with the same name
pd.merge(df1, df2, on='key', suffixes=['_l', '_r'])
```

	data1	key	colx_l	data2	colx_r
0	-0.783222	b	-0.775727	1.793256	-2.295756
1	0.235611	b	-0.332252	1.793256	-2.295756
2	-1.456009	b	-0.653220	1.793256	-2.295756
3	1.699517	a	-0.313720	0.291307	-1.420432
4	-1.256941	a	-0.240593	0.291307	-1.420432
5	0.037266	a	0.656888	0.291307	-1.420432

Pandas Merge

Merge on columns and index

```
# Set lkey to be the index of df3
df3.set_index('lkey', inplace=True)
# Note: Do this only once. Re-running set_index will produce errors.
# You'll have to reset index before you can set it again.

# We specify that
# - for the df2 we will use the column 'key' and
# - for the df4, we will use its index to merge
pd.merge(df2, df3, how='left', left_on='key', right_index=True)
```

	data2	key	colx	data3
0	0.291307	a	-1.420432	1.187114
0	0.291307	a	-1.420432	-0.217327
0	0.291307	a	-1.420432	0.030147
1	1.793256	b	-2.295756	-1.377747
1	1.793256	b	-2.295756	0.572455
1	1.793256	b	-2.295756	0.516989
2	-1.967771	d	-0.863023	NaN

Pandas Join

- The join() function in pandas is a convenient DataFrame method for combining many DataFrame objects with
  - same or similar indexes but
  - non-overlapping columns

  into a single result DataFrame.
  By default, the join method performs a left join on the join keys.
  For simple index-on-index merges we can pass a list of DataFrames to join.

Pandas Join

Datasets used -

df1

	W	X
c	1.818955	-1.222425
d	0.526972	-0.030206
e	1.429800	0.667704
f	1.044459	-0.926347
g	0.658220	-0.036131
h	0.077873	-1.784317

df2

	Y	Z
a	-0.862884	1.119847
b	-1.882780	1.587770
c	-0.271788	-0.955701
d	-0.791876	2.018576
e	-0.268978	0.862089

Pandas Join

Default join = Left Join

```
df1.join(df2)
```

	W	X	Y	Z
c	1.818955	-1.222425	-0.271788	-0.955701
d	0.526972	-0.030206	-0.791876	2.018576
e	1.429800	0.667704	-0.268978	0.862089
f	1.044459	-0.926347	NaN	NaN
g	0.658220	-0.036131	NaN	NaN
h	0.077873	-1.784317	NaN	NaN

Use 'How=' option to specify other kind of join

```
df1.join(df2, how='outer')
```

	W	X	Y	Z
a	NaN	NaN	-0.862884	1.119847
b	NaN	NaN	-1.882780	1.587770
c	1.818955	-1.222425	-0.271788	-0.955701
d	0.526972	-0.030206	-0.791876	2.018576
e	1.429800	0.667704	-0.268978	0.862089
f	1.044459	-0.926347	NaN	NaN
g	0.658220	-0.036131	NaN	NaN
h	0.077873	-1.784317	NaN	NaN

Pandas concatenate

- The concat() function in pandas is used to Concatenate pandas objects along a particular axis with optional set logic along the other axes.

Pandas concatenate: Series object

- Series Datasets used-

S1

```
a    -0.975852
b    -1.052620
c     0.595705
dtype: float64
```

S3

```
h     0.529832
i     1.755314
dtype: float64
```

S2

```
d    -1.069066
e     0.534835
f    -0.222088
g     0.737064
dtype: float64
```

S4

```
a    -0.145600
b    -0.214596
c    -1.377337
d     0.477379
e    -0.448025
dtype: float64
```

Pandas concatenate: Series object

For Series object with no overlapping indexes

Axis=0 (default) will **append** the Series

```
# Default action is to append the data
pd.concat([s1, s2, s3],axis=0)
```

```
a     0.975852
b    -1.052620
c     0.595705
d    -1.069066
e     0.534835
f    -0.222088
g     0.737064
h     0.529832
i     1.755314
dtype: float64
```

Axis=1 will **merge** the Series to produce a DataFrame

```
# concat with axis=1 (non-overlapping index)
pd.concat([s1, s2, s3],axis=1)
```

	0	1	2
a	-0.975852	NaN	NaN
b	-1.052620	NaN	NaN
c	0.595705	NaN	NaN
d	NaN	-1.069066	NaN
e	NaN	0.534835	NaN
f	NaN	-0.222088	NaN
g	NaN	0.737064	NaN
h	NaN	NaN	0.529832
i	NaN	NaN	1.755314

Keys option with **axis=0** creates hierarchical index

```
pd.concat([s1, s2, s3], axis=0, \
          keys=['one', 'two', 'thr'])
```

```
one  a   -0.975852
     b   -1.052620
     c    0.595705
two  d    1.069066
     e    0.534835
     f   -0.222088
     g    0.737064
thr  h    0.529832
     i    1.755314
dtype: float64
```

Keys option with **axis=1** gives names to columns

```
pd.concat([s1, s2, s3],axis=1,keys=['one', 'two', 'thr'])
```

	one	two	thr
a	-0.975852	NaN	NaN
b	-1.052620	NaN	NaN
c	0.595705	NaN	NaN
d	NaN	-1.069066	NaN
e	NaN	0.534835	NaN
f	NaN	-0.222088	NaN
g	NaN	0.737064	NaN
h	NaN	NaN	0.529832
i	NaN	NaN	1.755314

Pandas concatenate: Series object

For Series object with overlapping indexes

- If there is an overlap on indexes, we can specify the join= parameter to intersect the data
- **Note**: that the join= option takes only 'inner' and 'outer'

```
# concat with overlapping index
# (default join type is outer)
pd.concat([s1, s4], axis=1)
```

	0	1
a	-0.975852	-0.145600
b	-1.052620	-0.214596
c	0.595705	-1.377337
d	NaN	0.477379
e	NaN	-0.448025

```
# if we specify a join type,
# this will be equivalent to a merge
pd.concat([s1, s4], axis=1, join='inner')
```

	0	1
a	-0.975852	-0.145600
b	-1.052620	-0.214596
c	0.595705	-1.377337

Pandas concatenate: DataFrame object

For DataFrame object with no overlapping indexes

Datasets used ->

```
df1
```

	X	Y	Z
a	-0.991374	-0.569228	0.931171
b	1.738033	-0.058462	0.572353
c	-1.270316	-0.666415	-0.796420

```
df2
```

	X	Z
p	0.517311	2.159012
q	-1.077229	0.182628

axis = 0 will produce a **concatenation**

```
#non overlapping indexes and axis=0
pd.concat([df1, df2], axis=0)
```

	X	Y	Z
a	0.991374	0.569228	0.931171
b	1.738033	-0.058462	0.572353
c	1.270316	0.666415	0.796420
p	0.517311	NaN	2.159012
q	1.077229	NaN	0.182628

axis = 1 will produce as **merge**

```
#non-overlapping indexes and axis=0
pd.concat([df1, df2], axis=1)
```

	X	Y	Z	X	Z
a	-0.991374	-0.569228	0.931171	NaN	NaN
b	1.738033	0.058462	0.572353	NaN	NaN
c	-1.270316	-0.666415	-0.796420	NaN	NaN
p	NaN	NaN	NaN	0.517311	2.159012
q	NaN	NaN	NaN	-1.077229	0.182628

Pandas concatenate: DataFrame object

For DataFrame object with overlapping indexes

Datasets used->

df3

	X	Y	Z
a	0.989159	0.479682	0.188061
b	-0.002296	0.409742	0.559754
c	0.817331	1.995039	-0.528110

df4

	X	Z
a	1.197605	2.162543
c	-0.698781	-0.050585

Axis=0

```
# when axis=0 will concatenate
pd.concat([df3, df4])
```

	X	Y	Z
a	0.989159	0.479682	0.188061
b	-0.002296	0.409742	0.559754
c	0.817331	1.995039	-0.528110
a	1.197605	NaN	-2.162543
c	-0.698781	NaN	-0.050585

Axis=1

```
# Overlapping indexes will be merged
pd.concat([df3, df4], axis=1)
```

	X	Y	Z	X	Z
a	0.989159	0.479682	0.188061	1.197605	-2.162543
b	-0.002296	0.409742	0.559754	NaN	NaN
c	0.817331	1.995039	-0.528110	-0.698781	-0.050585

```
# This will create a hierarchical index
pd.concat([df3, df4], axis=1, keys=['lev_1', 'lev_2'])
```

	lev_1			lev_2	
	X	Y	Z	X	Z
a	0.989159	0.479682	0.188061	1.197605	2.162543
b	-0.002296	0.409742	0.559754	NaN	NaN
c	0.817331	1.995039	-0.528110	-0.698781	-0.050585

Pandas Reshape data: Stack &Unstack

For pandas dataframes with hierarchical indices, stack and unstack provide a convenient way to reshape the data from wide-to-long or long-to-wide formats.

- `**stack**` pivots the columns into rows
- `**unstack**` pivots rows into columns

Dataset used

df

		X	Y
one	two		
A	C	-0.863573	-1.165235
B	D	-1.416981	0.992815
A	E	0.161577	-1.255732
B	F	0.570989	-1.005938

Stack()

```
my_stack = df.stack()
my_stack
one  two
A    C     X   -0.863573
           Y    1.165235
B    D     X   -1.416981
           Y    0.992815
A    E     X    0.161577
           Y    1.255732
B    F     X    0.570989
           Y    1.005938
dtype: float64
```

Unstack()

```
my_unstack = my_stack.unstack()
my_unstack
```

		X	Y
one	two		
A	C	0.863573	1.165235
	E	0.161577	-1.255732
B	D	1.416981	0.992815
	F	0.570989	-1.005938

Pandas Reshape data: Pivot table

Pivot() method takes the names of columns to be used as row (index=) and column indexes (columns=) and a column to fill in the data as (values=).

	date	Item	status
0	2000-01-03	A	1.562997
1	2000-01-04	B	-0.036311
2	2000-01-05	C	0.031682
3	2000-01-03	D	2.305477
4	2000-01-04	A	0.222675
5	2000-01-05	B	0.118446
6	2000-01-03	C	0.207927
7	2000-01-04	D	0.720234
8	2000-01-05	A	2.167930
9	2000-01-03	B	-2.343839
10	2000-01-04	C	1.805317
11	2000-01-05	D	1.172874

```
df.pivot(index='date', columns='item', values='status')
```

item	A	B	C	D
date				
2000-01-03	1.562997	-2.343839	-0.207927	2.305477
2000-01-04	0.222675	-0.036311	-1.805317	0.720234
2000-01-05	-2.167930	0.118446	-0.031682	1.172874

Pandas Reshape data: Pivot table

pivot_table() method is similar to pivot, but-

- can work with duplicate indices and
- lets you **specify an aggregation function**

The pivot_table function in pandas is a very natural way of specifying the same thing you would using Excel.

	C1	C2	N1
0	x	a	0.874334
1	x	b	-0.388188
2	x	b	-0.889307
3	x	b	-0.229557
4	y	a	0.897041
5	y	a	0.755192
6	y	b	0.522836
7	y	a	0.745250
8	x	a	3.383863
9	x	b	-0.078494
10	x	b	-0.040139
11	x	b	1.052541
12	y	a	-0.661041
13	y	a	0.883821
14	y	b	-0.403303
15	y	a	-1.161802

```
df2.pivot_table(index='C1',columns='C2',values='N1',aggfunc='sum')
```

C2	a	b
C1		
x	4.258197	-0.573144
y	1.458461	0.119533

Data Visualization

Plotting in Pandas

- There are high level plotting methods that take advantage of the fact that data are organized in
- DataFrames (have index, colnames)
- Both Series and DataFrame objects have a pandas.plot method for making different plot types by
- specifying a kind= parameter
- Other parameters that can be passed to pandas.plot are:
  - xticks, xlim, yticks, ylim
  - label
  - style (as an abbreviation,) and alpha
  - grid=True
  - rot (rotate tick labels by and angle 0-360)
  - use_index (use index for tick labels)
- Note: If you’re using the IPython Notebook, run the following code %matplotlib inline

Python Data Visualization libraries

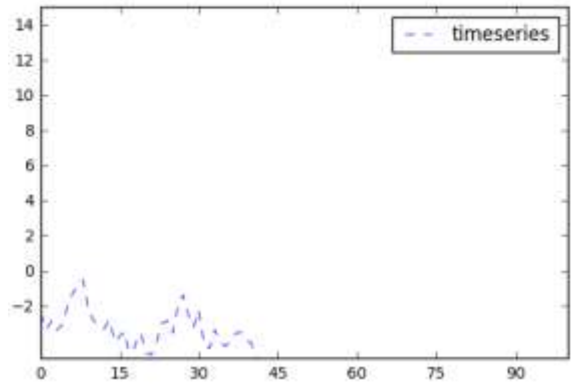
- Matplotlib
- Seaborn
- GGPLOT
- Altair
- Plotly
- Plotting in Pandas

Plotting in Pandas

- Univariate data– plotting a numeric Series
  - Line chart
  - Histogram
  - Desity plots
- Multivariate data- plotting a numeric DataFrame
  - Line plot
  - Bar plot and Stacked bar plot
  - Scatter plot
  - Scatter plot matrix

Plotting in Pandas: Univariate data – Line plot

```
%matplotlib inline

s = pd.Series(np.random.randn(100).cumsum())

s.plot(kind='line',
grid=False, legend=True,
label='timeseries',
xlim=(0, 100), ylim=(-5, 15),
xticks=np.arange(0, 100, 15), yticks=np.arange(-2, 15, 2),
style='b--', alpha=0.7 )
```

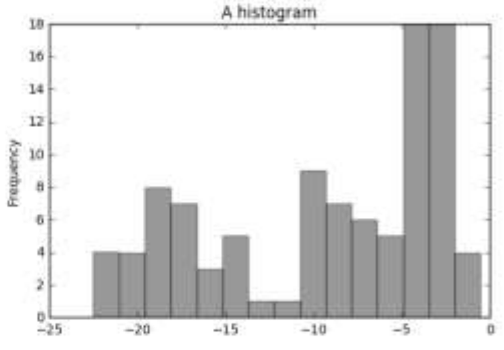


Plotting in Pandas: Univariate – histogram

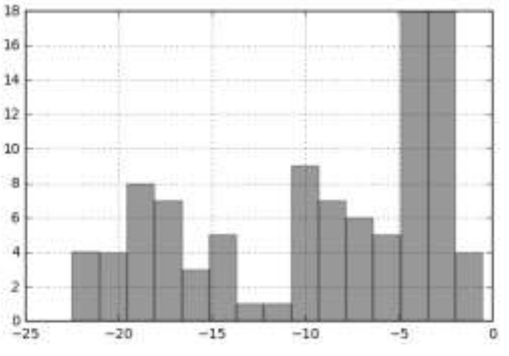
Histograms: Pass kind='hist' to pd.plot() or use the method pd.hist()

```
s = pd.Series(np.random.randn(100).cumsum())
```

```
s.plot(kind='hist', bins=15, color='k', alpha=0.4, title='A histogram')
<matplotlib.axes._subplots.AxesSubplot at 0xdd71d535c0>
```




```
s.hist(bins=15, color='k', alpha=0.4);
<matplotlib.axes._subplots.AxesSubplot at 0xdd7365eef0>
```



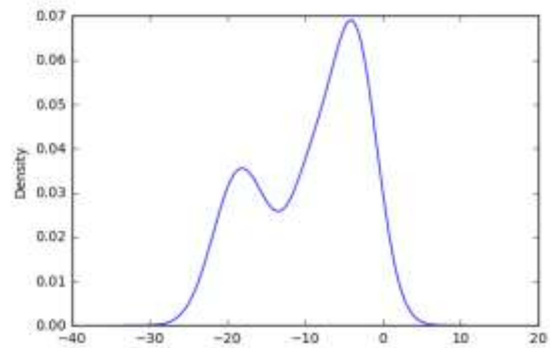
Plotting in Pandas: Univariate – Density Plots

- Plots : Use kind='kde'

```
s = pd.Series(np.random.randn(100).cumsum())
```

```
#plotting a density plot
s.plot(kind='kde')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xdd68c2ec18>
```



Plotting in Pandas: Multivariate data

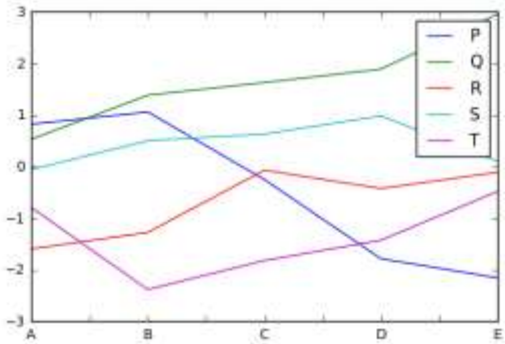
- We can choose between plotting
  - All Variables on one plot
  - Each variable on a separate plot
- In addition to the parameters above, DataFrame.plot also takes
  - subplots=False (default is to plot all on the same figure)
  - sharex=False, sharey=False
  - figsize
  - title, legend
  - sort_columns

Plotting in Pandas: Multivariate data -Lineplot

Variables on the same plot – Lineplot

```
df = pd.DataFrame(np.random.randn(5,5), index=list('ABCDE'), columns=list('PQRST'))
print (df)
# Default plot
df.cumsum().plot()
```

	P	Q	R	S	T
A	0.824870	0.532092	-1.594693	-0.055563	-0.788238
B	0.235545	0.853170	0.318968	0.558812	-1.592538
C	-1.308210	0.245264	1.208790	0.129900	0.561593
D	-1.535374	0.259693	-0.355441	0.351119	0.392783
E	-0.370577	1.067251	0.313526	-0.878314	0.946814



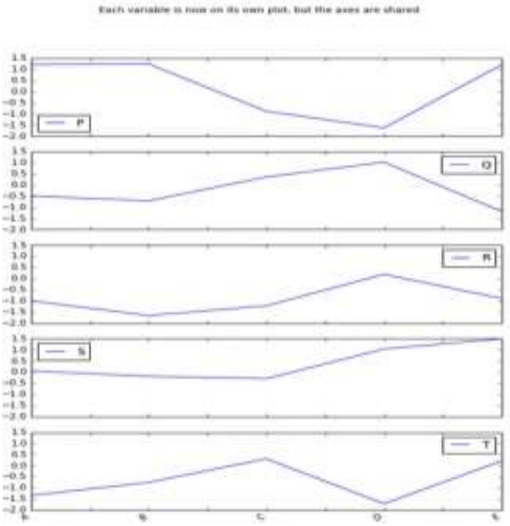
Plotting in Pandas: Multivariate data -Lineplot

Each variable on its own plot

```
df = pd.DataFrame(np.random.randn(5,5), index=list('ABCDE'), columns=list('PQRST'))
print (df)
```

	P	Q	R	S	T
A	0.824870	0.532092	-1.594693	-0.055563	-0.788238
B	0.235545	0.853170	0.318968	0.558812	-1.592538
C	-1.308210	0.245264	1.208790	0.129900	0.561593
D	-1.535374	0.259693	-0.355441	0.351119	0.392783
E	-0.370577	1.067251	0.313526	-0.878314	0.946814

```
df.plot(kind='line',
figsize=(8, 12),
title='Each variable is now on its own plot, but the axes are shared',
color='b',
subplots=True, sharex=True, sharey=True)
```



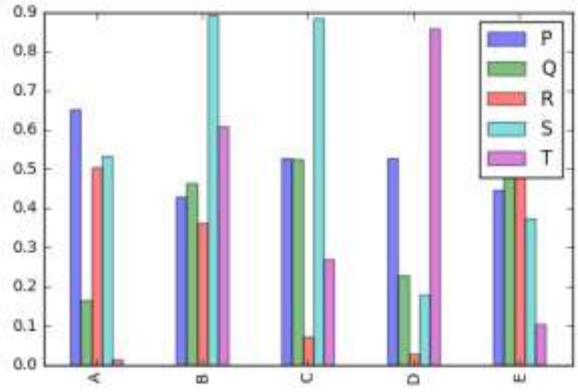
Plotting in Pandas: Multivariate data - Barplot

```
df = pd.DataFrame(np.random.rand(5,5), index=list('ABCDE'), columns=list('PQRST'))
print (df)
```

	P	Q	R	S	T
A	0.652178	0.163587	0.503694	0.533488	0.012362
B	0.428721	0.464710	0.361471	0.894103	0.608950
C	0.526888	0.525934	0.072456	0.884661	0.269902
D	0.526317	0.227760	0.028076	0.179084	0.858565
E	0.446169	0.873193	0.681589	0.373404	0.102756

```
df.plot(kind='bar', alpha=0.5)
```

<matplotlib.axes._subplots.AxesSubplot at 0xdd720c7828>

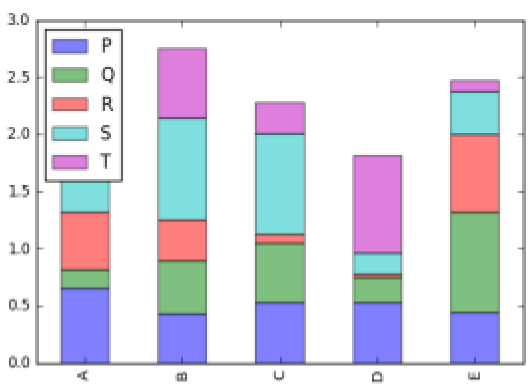


Plotting in Pandas: Multivariate data - Stacked Barplot

```
df = pd.DataFrame(np.random.rand(5,5), index=list('ABCDE'), columns=list('PQRST'))
print (df)
```

	P	Q	R	S	T
A	0.652178	0.163587	0.503694	0.533488	0.012362
B	0.428721	0.464710	0.361471	0.894103	0.608950
C	0.526888	0.525934	0.072456	0.884661	0.269902
D	0.526317	0.227760	0.028076	0.179084	0.858565
E	0.446169	0.873193	0.681589	0.373404	0.102756

```
df.plot(kind='bar', stacked=True, alpha=0.5)
```



Plotting in Pandas: Multivariate data - Scatterplot

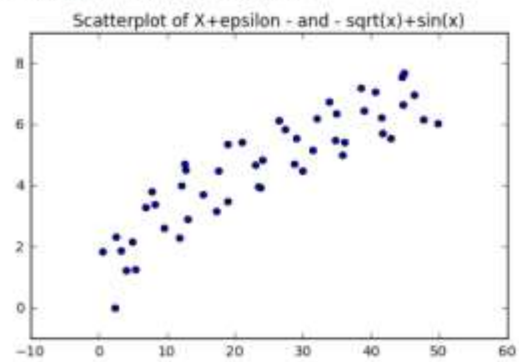
- This requires scatter function from matplotlib

```
# Create a dataset
df = pd.DataFrame({'A': np.arange(50),
'B': np.arange(50) + np.random.randn(50),
'C': np.sqrt(np.arange(50)) + np.sin(np.arange(50)) })
print (df.head())
```

```
   A         B         C
0  0  2.447854  0.000000
1  1  0.595245  1.841471
2  2  2.558939  2.323511
3  3  3.346576  1.873171
4  4  5.994226  1.243198
```

```
# Two variable Scatterplot
plt.scatter(df['B'], df['C'])
plt.title('Scatterplot of X+epsilon - and - sqrt(x)+sin(x)')
```

```
<matplotlib.text.Text at 0xdd78268f98>
```

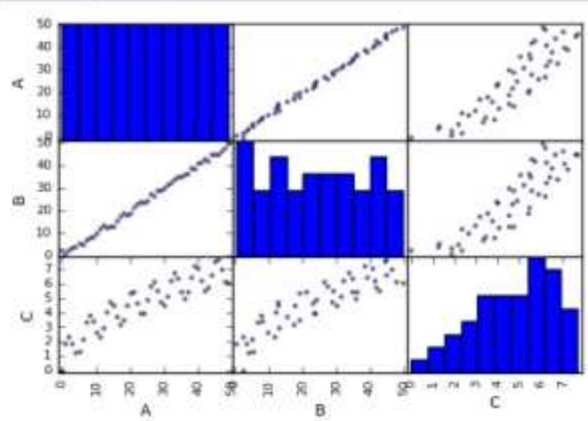


Plotting in Pandas: Multivariate data - Scattermatrix

```
# Create a dataset
df = pd.DataFrame({'A': np.arange(50),
'B': np.arange(50) + np.random.randn(50),
'C': np.sqrt(np.arange(50)) + np.sin(np.arange(50)) })
print (df.head())
```

```
   A         B         C
0  0  2.447854  0.000000
1  1  0.595245  1.841471
2  2  2.558939  2.323511
3  3  3.346576  1.873171
4  4  5.994226  1.243198
```

```
pd.scatter_matrix(df);
```



Writing Data from Pandas

Writing Data into Pandas

- Writing to a CSV or a Flat file
- Writing to an excel sheet
- Writing to a JSON file

Writing to a CSV file or a flat file

- We use to_csv() function.
- Important points –
  - Syntax= to_csv(“file name with extension”, sep=, <other options>)
  - Default separator = csv; for tab use ‘\t’
  - For tab delimited file use ‘txt’ as extension.

Our sample data

```
countries.to_csv('my_country_data.csv')
```

	area	capital	country	population
0	30510	Brussels	BE	11.3
1	671308	Paris	FR	64.3
2	357050	Berlin	GR	81.3
3	41526	Amsterdam	NL	16.9
4	244820	London	UK	64.9

```
countries.to_csv("my_country_data.txt",sep='\t')
```

```
my_country_data - Notepad
File  Edit  Format  View  Help
        area    capital   country population
0       30510   Brussels  BE      11.3
1       671308  Paris     FR      64.3
2       357050  Berlin    GR      81.3
3       41526   Amsterdam NL      16.9
4       244820  London    UK      64.9
```

Writing to an excel file

- We use to_excel() function.
- Important points –
  - Syntax= to_excel(“file name with extension xlsx”, sheetname=, <other options>)

Our sample data

	area	capital	country	population
0	30510	Brussels	BE	11.3
1	671308	Paris	FR	64.3
2	357050	Berlin	GR	81.3
3	41526	Amsterdam	NL	16.9
4	244820	London	UK	64.9

```
countries.to_excel('country_excel_data.xlsx', sheet_name='Sheet1')
```

	area	capital	country	population
0	30510	Brussels	BE	11.3
1	671308	Paris	FR	64.3
2	357050	Berlin	GR	81.3
3	41526	Amsterdam	NL	16.9
4	244820	London	UK	64.9

Writing to a JSON file

- To write to a json file we have function **to_json()** that converts a pandas object into json string
  Syntax: to_json(*File path, sheet_name='Sheet1', <other options>)*
- **sheet_name** *: string, default 'Sheet1'*
- Name of sheet which will contain DataFrame
  - Other options - http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_json.html

```
#Let us write content of our DataFrame into JSON file
my_sales.head()
```

	Customer_id	Customer_name	Subsegment	City	Division	Category	Version	Sales_amount	No_of_Licences	Sales_Date
0	129	C1	Lower Mid-Market	Chennai	RSD9	RSD9_RSC3	2003	58,719	37	3/8/2008
1	419	C2	Upper Mid-Market	Delhi	RSD9	RSD9_RSC5	2002_V2	16,944	12	11/25/2008

```
#write to json file
df=my_sales.to_json("sales_data.json")
```

sales_data.json - Notepad

File Edit Format View Help

{"Customer_id":{"0":129,"1":419,"2":270
9,"112":487,"113":220,"114":314,"115":2
14":361,"215":348,"216":140,"217":195,"

Thank you