Python for Data Analysis

Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects

- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance

- many other python libraries are built on NumPy

Link: http://www.numpy.org/

Python Libraries for Data Science

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more

- part of SciPy Stack

- built on NumPy

Link: https://www.scipy.org/scipylib/

Python Libraries for Data Science

Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)

- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.

- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.

- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats

- a set of functionalities similar to those of MATLAB

- line plots, scatter plots, barcharts, histograms, pie charts etc.

- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

Python Libraries for Data Science

Seaborn:

- based on matplotlib

- provides high level interface for drawing attractive statistical graphics

- Similar (in style) to the popular ggplot2 library in R

Link: https://seaborn.pydata.org/

Pandas

Loading Python Libraries

In [ ]:
```
#Import Python Libraries
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mpl
import seaborn as sns
```

Press `Shift+Enter` to execute the *jupyter* cell

Introduction to Pandas

- It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.
- pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib.
- pandas adopts significant parts of NumPy’s idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops.
- While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data.

Introduction to pandas Data Structures

- import pandas as pd
- from pandas import Series, DataFrame
- To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

- A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data:
- obj = pd.Series([4, 7, -5, 3])
- obj
- Output: 0   4
-                 1   7
-                 2   -5
-                 3   3
-                 dtype: int64

Series

- Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:
- In [13]: obj.values
- Out[13]: array([4, 7, -5, 3])
- In [14]: obj.index # like range(4)
- Out[14]: RangeIndex(start=0, stop=4, step=1)

Series

- Often it will be desirable to create a Series with an index identifying each data point with a label:
- In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
- d   4
- b   7
- a   -5
- c   3
- dtype: int64
- In [17]: obj2.index
- Index(['d', 'b', 'a', 'c'], dtype='object')

- Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:
- obj2['a']
- -5
- obj2['d'] = 6
- obj2[['c', 'a', 'd']]
- c 3 a -5 d 6 dtype: int64
- Here ['c', 'a', 'd'] is interpreted as a list of indices, even though it contains strings instead of integers.
- obj2[obj2 > 0]

  d 6

  b 7

  c 3

  dtype: int64
- obj2 * 2

  d 12

  b 14

  a -10

  c 6 dtype: int64

Series

- Another way to think about a Series is as a fixed-length, ordered dict,
- As it is a mapping of index values to data values, it can be used in many contexts where you might use a dict:

In [24]: 'b' in obj2

True

- Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

- obj3 = pd.Series(sdata)
- obj3

Nan Object

- When you are only passing a dict, the index in the resulting Series will have the dict’s keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:
- states = ['California', 'Ohio', 'Oregon', 'Texas']
- obj4 = pd.Series(sdata, index=states)
- obj4

```
California     NaN
Ohio           35000.0
Oregon         16000.0
Texas          71000.0 dtype: float64
```

Here, three values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number), which is considered in pandas to mark missing or NA values. Since 'Utah' was not included in states, it is excluded from the resulting object.

Nan Object

- The isnull and notnull functions in pandas should be used to detect missing data:
- pd.isnull(obj4)

```
California True
Ohio       False
Oregon     False
Texas      False
dtype: bool
```

- pd.notnull(obj4) Out[33]:

```
California  False
Ohio        True
Oregon      True
Texas       True
dtype: bool
```

- Series also has these as instance methods:
- obj4.isnull()

Series Alignment

- A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:
- obj3

```
Ohio       35000
Oregon     16000
Texas      71000
Utah        5000
dtype: int64
```

- obj4

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
```

- obj3 + obj4

```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
```

Series Alignment

obj4.name = 'population'

obj4.index.name = 'state'

- obj4

  State

  California NaN

  Ohio 35000.0

  Oregon 16000.0

  Texas 71000.0

  Name: population, dtype: float64

Altering Series Index

- A Series’s index can be altered in-place by assignment:
- In [41]: obj
- 0   4
- 1   7
- 2   -5
- 3   3 dtype: int64
- obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
- Obj
- Bob   4
- Steve   7
- Jeff   -5
- Ryan   3
- dtype: int64

DataFrame

- A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
- The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index.
- Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays

DataFrame

- There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:
- data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002, 2003], 'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
- frame = pd.DataFrame(data)
- The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:
- In [45]: frame
- pop state year
- 0 1.5 Ohio 2000
- 1 1.7 Ohio 2001
- 2 3.6 Ohio 2002
- 3 2.4 Nevada 2001
- 4 2.9 Nevada 2002
- 5 3.2 Nevada 2003

DataFrame

- For large DataFrames, the head method selects only the first five rows:
- frame.head()
- If you specify a sequence of columns, the DataFrame’s columns will be arranged in that order:
- pd.DataFrame(data, columns=['year', 'state', 'pop'])
-     year   state pop
- 0 2000   Ohio 1.5
- 1 2001   Ohio 1.7
- 2 2002   Ohio 3.6
- 3 2001 Nevada 2.4
- 4 2002 Nevada 2.9
- 5 2003 Nevada 3.2
- If you specify a sequence of columns, the DataFrame’s columns will be arranged in that order:
- In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])

DataFrame

- If you pass a column that isn't contained in the dict, it will appear with missing values in the result:
- frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],

  index=['one', 'two', 'three', 'four', ...: 'five', 'six'])
- frame2
- year state pop debt
- one 2000 Ohio 1.5 NaN
- two 2001 Ohio 1.7 NaN
- …….
- six 2003 Nevada 3.2 NaN

Retrieving Columns in a Dataframe

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:
- frame2['state']
- frame2.year
- frame2.loc['three']
- Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:
- frame2['debt'] = 16.5
- frame2['debt'] = np.arange(6.)

Assignment in a dataframe

- When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame.
- If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes. Frame 2 is
- year state pop debt
- one 2000 Ohio 1.5 16.5
- two 2001 Ohio 1.7 16.5
- three 2002 Ohio 3.6 16.5
- four 2001 Nevada 2.4 16.5
- five 2002 Nevada 2.9 16.5
- six 2003 Nevada 3.2 16.5

Assigninment in a DataFrame

- Val=pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
- frame2['debt'] = val
- Shows
- year state pop debt
- one 2000 Ohio 1.5 NaN
- two 2001 Ohio 1.7 -1.2
- three 2002 Ohio 3.6 NaN
- four 2001 Nevada 2.4 -1.5
- five 2002 Nevada 2.9 -1.7
- six 2003 Nevada 3.2 NaN

Adding a new column in df

- As an example of del, first add a new column of boolean values where the state column equals 'Ohio':
- frame2['eastern'] = frame2.state == 'Ohio'
- frame2

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
Five	2002	Nevada	2.9	-1.7	False
six	2003	Nevada	3.2	NaN	False

- New columns cannot be created with the frame2.eastern syntax.

Deleting a column

- The del method can then be used to remove this column:
- del frame2['eastern']
- frame2.columns
- Index(['year', 'state', 'pop', 'debt'], dtype='object')
- The column returned from 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 DataFrame.
- The column can be explicitly copied with the Series's copy method.

Nesting of dictionaries

- Another common form of data is a nested dict of dicts:
- pop = {'Nevada': {2001: 2.4, 2002: 2.9},

  ....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

Instead of what we have already done

- data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002, 2003], 'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
- If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices:
- frame3 = pd.DataFrame(pop)
- frame3

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Transposing Dataframes

- You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:
- In [68]: frame3.T
- 2000 2001 2002
- Nevada NaN 2.4 2.9
- Ohio 1.5 1.7 3.6
- The keys in the inner dicts are combined and sorted to form the index in the result. This isn't true if an explicit index is specified:
- In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
- Nevada Ohio
- 2001 2.4 1.7
- 2002 2.9 3.6
- 2003 NaN NaN

Pandas

- **Important things you should know about Numpy and Pandas**
- The data manipulation capabilities of pandas are built on top of the numpy library. In a way, numpy is a dependency of the pandas library.
- Pandas is best at handling tabular data sets comprising different variable types (integer, float, double, etc.). In addition, the pandas library can also be used to perform even the most naive of tasks such as loading data or doing feature engineering on time series data.
- Numpy is most suitable for performing basic numerical computations such as mean, median, range, etc. Alongside, it also supports the creation of multi-dimensional arrays.
- Numpy library can also be used to integrate C/C++ and Fortran code.
- Remember, python is a zero indexing language unlike R where indexing starts at one.
- The best part of learning pandas and numpy is the strong active community support you'll get from around the world.

Reading data using pandas

```
In [ ]: #Read csv file
        df = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1', index_col=None, na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
```

Data Frame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Data Frame data types

In [4]:
```
#Check a particular column type
df['salary'].dtype
```

Out[4]: dtype('int64')

In [5]:
```
#Check types for all the columns
df.dtypes
```

Out[4]:
```
rank          object
discipline    object
phd           int64
service       int64
sex           object
salary        int64
dtype: object
```

Data Frames methods

Unlike attributes, python methods have *parenthesis.*
All attributes and methods can be listed with a *dir()* function: `dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Data Frames *groupby* method

Once groupby object is create we can calculate various statistics for each group:

In [ ]:
```
#Calculate mean salary for each professor rank:
df.groupby('rank')[['salary']].mean()
```

	salary
rank	
AssocProf	91786.230769
AsstProf	81362.789474
Prof	123624.804348

Note: If single brackets are used to specify the column (e.g. salary), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

Data Frames *groupby* method

groupby performance notes:

- no grouping/splitting occurs until it's needed. Creating the *groupby* object only verifies that you have passed a valid mapping
- by default the group keys are sorted during the *groupby* operation. You may want to pass sort=False for potential speedup:

In [ ]:
```
#Calculate mean salary for each professor rank:
df.groupby(['rank'], sort=False)[['salary']].mean()
```

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than $120K:

```
In [ ]: #Calculate mean salary for each professor rank:
        df_sub = df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;
< less; <= less or equal;
== equal; != not equal;

```
In [ ]: #Select only those rows that contain female professors:
        df_f = df[ df['sex'] == 'Female' ]
```

Data Frames: Slicing

There are a number of ways to subset the Data Frame:

- one or more columns
- one or more rows
- a subset of rows and columns

Rows and columns can be selected by their position or label

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

```
In [ ]: #Select column salary:
        df['salary']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

```
In [ ]: #Select column salary:
        df[['rank','salary']]
```

Data Frames: Selecting rows

If we need to select a range of rows, we can specify the range using ":"

```
In [ ]: #Select rows by their position:
        df[10:20]
```

Notice that the first row has a position 0, and the last value in the range is omitted: So for 0:10 range the first 10 rows are returned with the positions starting with 0 and ending with 9

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

```
In [ ]: #Select rows by their labels:
        df_sub.loc[10:20,['rank','sex','salary']]
```

Out[ ]:

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

```
In [ ]: #Select rows by their index:
        df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[ ]:

	rank	service	sex	salary
26	Prof	19	Male	148750
27	Prof	43	Male	155865
29	Prof	20	Male	123683
31	Prof	21	Male	155750
35	Prof	23	Male	126933
36	Prof	45	Male	146856
39	Prof	18	Female	129000
40	Prof	36	Female	137000
44	Prof	19	Female	151768
45	Prof	25	Female	140096

Data Frames: method iloc (summary)

```
df.iloc[0]  # First row of a data frame
df.iloc[i]  #(i+1)th row
df.iloc[-1] # Last row
```

```
df.iloc[:, 0]  # First column
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]        #First 7 rows
df.iloc[:, 0:2]     #First 2 columns
df.iloc[1:3, 0:2]   #Second through third rows and first 2 columns
df.iloc[[0,5], [1,3]]  #1st and 6th rows and 2nd and 4th columns
```

Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

In [ ]:
```
# Create a new data frame from the original sorted by the column Salary
df_sorted = df.sort_values( by ='service')
df_sorted.head()
```

Out[ ]:

	rank	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

Data Frames: Sorting

We can sort the data using 2 or more columns:

In [ ]:
```
df_sorted = df.sort_values( by =['service', 'salary'], ascending = [True, False])
df_sorted.head(10)
```

Out[ ]:

	rank	discipline	phd	service	sex	salary
52	Prof	A	12	0	Female	105000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
55	AsstProf	A	2	0	Female	72500
57	AsstProf	A	3	1	Female	72500
28	AsstProf	B	7	2	Male	91300
42	AsstProf	B	4	2	Female	80225
68	AsstProf	A	4	2	Female	77500

Missing Values

Missing values are marked as NaN

In [ ]:
```
# Read a dataset with missing values
flights = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/flights.csv")
```

In [ ]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[ ]:

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWR	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWR	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Missing Values

There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- cumsum() and cumprod() methods ignore missing values but preserve them in the resulting arrays
- Missing values in GroupBy method are excluded (just like in R)
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default (unlike R)

Aggregation Functions in Pandas

Aggregation - computing a summary statistic about each group, i.e.

- compute group sums or means
- compute group sizes/counts

Common aggregation functions:

min, max
count, sum, prod
mean, median, mode, mad
std, var

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

In [ ]:
```
flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

Out[ ]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

Graphics to explore the data

Seaborn package is built on matplotlib but provides high level interface for drawing attractive statistical graphics, similar to ggplot2 library in R. It specifically targets statistical data visualization

To show graphs within Python notebook include inline directive:

```
In [ ]: %matplotlib inline
```

Graphics

	description
distplot	histogram
barplot	estimate of central tendency for a numeric variable
violinplot	similar to boxplot, also shows the probability density of the data
jointplot	Scatterplot
regplot	Regression plot
pairplot	Pairplot
boxplot	boxplot
swarmplot	categorical scatterplot
factorplot	General categorical plot

Basic statistical Analysis

statsmodel and scikit-learn - both have a number of function for statistical analysis

The first one is mostly used for regular analysis using R style formulas, while scikit-learn is more tailored for Machine Learning.

statsmodels:

- linear regressions
- ANOVA tests
- hypothesis testings
- many more ...

scikit-learn:

- kmeans
- support vector machines
- random forests
- many more ...