Data Analytics
Week 8

Recap – Last week

- Introduction to numerical python – Numpy library
- Introduction to Scikit-Learn and Machine Learning

This week's lecture

- Pandas framework for data analysis
  - Introduction to Pandas (first half)
  - Data manipulation using Pandas (second half)
- Tutorial document on Pandas (Lab)

Lecture materials available as

- Pdf document on QMPlus
- As a python notebook file (same contents from pdf) both on QMPlus and GitHub https://github.com/BhusanChettri/da2020/tree/master/lecture3
- Notebook files on QMPlus are uploaded in zipped format. Having downloaded them do not forget to unzip those files!

This week’s lab class

- Machine learning with Scikit-Learn tutorials - continue
- Pandas tutorial reading and walkthrough – make use of notebook files provided!

PANDAS

AN OPEN SOURCE PYTHON LIBRARY TO SUPPORT DATA ANALYSIS

Pandas

- Introduces two new data structures to Python –

  1) Series

  2) DataFrames

- Both of these are built on top of NumPy and provides high-performance
- easy-to-use data structures and data analysis tools for the python programming language

Setting up

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
pd.set_option('max_columns', 50)
%matplotlib inline
```

Series

A Series is a one-dimensional object similar to an array, list, or column in a table. The system will assign a labelled index to each item in the Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

Example1: creating Series from a list.

```
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!’])
print(s)                    # prints the following
```

```
0       7
1       Heisenberg
2       3.14
3       -1789710578
4     Happy Eating!
dtype: object
```

NOTE: We refer pandas as pd

Indexing a series

Alternatively, you can specify an index to use when creating the Series rather than relying on the default ones.

```
s = pd.Series([7, 'Heisenberg', 3.14, -1789710578, 'Happy Eating!'],
              index=['A', 'Z', 'C', 'Y', 'E’])

print(s)                    # would print details with assigned indexes
```

```
A       7
Z       Heisenberg
C       3.14
Y       -1789710578
E       Happy Eating!
dtype: object
```

Creating a series from a dictionary

The Series constructor can convert a dictionary as well, using the keys of the dictionary as its index.

```
d = {'Chicago': 1000, 'New York': 1300, 'Portland': 900, 'San Francisco': 1100, 'Austin': 450, 'Boston': None}

cities = pd.Series(d)

print(cities)                    # would print the following
```

```
Austin          450
Boston          NaN
Chicago         1000
New York        1300
Portland        900
San Francisco   1100
dtype: float64
```

Selecting members of a series

You can use the index to select specific items from the Series ...

```
cities[‘Chicago’]                # prints:     1000.0
cities[[‘Chicago’, ‘Portland’, ‘San Francisco’]]     # prints following

Chicago          1000
Portland         900
San Francisco    1100
dtype: float64
```

Using boolean indexing for selection

```
print(cities[cities < 1000])      # shall print all cities with values less than 1000
```

```
Austin      450
Portland    900
dtype: float64
```

Changing values using boolean logic

```
cities[cities < 1000] = 750       # change values of cities less than 1000 to 750
print cities[cities < 1000]       # prints following
```

```
Austin      750
Portland    750
dtype: float64
```

Checking if an item is in a series

```
print('Seattle' in cities)              # prints   False
print('San Francisco' in cities)        # prints   True
```

Maths operations on series

1. Using scalar operation

```
cities / 3       # would print

Austin           250.000000
Boston           NaN
Chicago          466.666667
New York         433.333333
Portland         250.000000
San Francisco    366.666667
dtype: float64
```

2. Using math function

```
np.square(cities)      # squares all values of cities

Austin           562500
Boston           NaN
Chicago          1960000
New York         1690000
Portland         562500
San Francisco    1210000
dtype: float64
```

Checking for nulls

Use isnull and notnull (similar to SQL):

1) Using notnull()

```
cities.notnull( )

Austin          True
Boston          False
Chicago         True
New York        True
Portland        True
San Francisco   True
dtype: bool
```

2) Using isnull()

```
cities.isnull( )

Austin          False
Boston          True
Chicago         False
New York        False
Portland        False
San Francisco   False
dtype: bool
```

```
print(cities[cities.isnull()])   #prints Series elements with null values
Boston   NaN
dtype: float64
```

DataFrames

A DataFrame is a tabular data structure comprised of rows and columns, akin to a spreadsheet, relational database table, or R's data.frame object. You can also think of a DataFrame as a group of Series objects that share an index (the column names).

Reading data into a DataFrame

```
data = {'year': [2010, 2011, 2012, 2011, 2012, 2010, 2011, 2012],
        'team': ['Bears', 'Bears', 'Bears', 'Packers', 'Packers', 'Lions', 'Lions', 'Lions'],
'wins': [11, 8, 10, 15, 11, 6, 10, 4], 'losses': [5, 8, 6, 1, 5, 10, 6, 12]}

football = pd.DataFrame(data, columns=['year', 'team', 'wins', 'losses’])
```

- data is a list of dictionary
- we create football DataFrame using the list
- football produces a table showing wins and losses for a team and a year

Contents of the football dataframe

```
>>> football
    year    team    wins    losses
0   2010    Bears   11      5
1   2011    Bears   8       8
2   2012    Bears   10      6
3   2011    Packers 15      1
4   2012    Packers 11      5
5   2010    Lions   6       10
6   2011    Lions   10      6
7   2012    Lions   4       12
```

Deleting a dataframe

del "dataframe name"

Other ways of populating dataframes

Pandas is capable of IO with csv, excel data, hdf, sql, json, msgpack, html, gbq, stata, clipboard and pickle data, and the list continues to grow.

Check out the following link: pandas_IO for detailed information.

We will look at two examples

- I/O with CSV
- I/O with excel files

I/O with CSV files

To read a CSV file we use the pandas read_csv method. Suppose we have a CSV file storing average house prices for some geographical area. To read the file into a dataframe and print it:

```
df = pd.read_csv('house_prices.csv’)
print(df.head())
```

	Date	Value
0	2015-06-30	502300
1	2015-05-31	501500
2	2015-04-30	500100
3	2015-03-31	495800
4	2015-02-28	492700

I/O with CSV files contd..

By default, read_csv expects a comma separator between fields in the file, but this can be overridden. Notice the output has default index numbers starting from 0.

To write the data out to another file:

```
df.to_csv('house_prices_copy.csv')
```

To write out just the values column:

```
df['Value'].to_csv('house_prices_values.csv')
```

I/O with CSV contd..

You can set an index to the data in the dataframe on import of the file:

```
df = pd.read_csv('house_prices_copy.csv ', index_col=0)
print(df.head())
```

	Date	Value
0	2015-06-30	502300
1	2015-05-31	501500
2	2015-04-30	500100
3	2015-03-31	495800
4	2015-02-28	492700

I/O with CSV contd..

One way to change the column header "value" is as follows:

```
df.columns = [‘Date’, ‘House_Prices’]
# Throws error if we use:  df.columns = [‘House_Prices’]
```

```
print(df.head())
```

```
        Date  House_Prices
0 2015-06-30        502300
1 2015-05-31        501500
2 2015-04-30        500100
3 2015-03-31        495800
4 2015-02-28        492700
```

If we wish to write it to a csv file without header:

```
df.to_csv('house_prices_no_header.csv', header=False)
```

I/O with CSV contd..

If the file has no headers, but we want them in the dataframe:

```
df = pd.read_csv('house_prices_no_header.csv', names = ['Date','
House_Price'])

print(df.head())
```

	Date	House_Price
0	2015-06-30	502300
1	2015-05-31	501500
2	2015-04-30	500100
3	2015-03-31	495800
4	2015-02-28	492700

I/O with Excel

- The read_excel() method can read Excel 2003 (.xls) and Excel 2007+ (.xlsx) files using the xlrd Python module.
- The to_excel() instance method is used for saving a DataFrame to Excel.
- Generally the semantics are similar to working with csv data.
- You can install the xlrd library via pip : pip install xlrd

I/O with Excel

Let's write a dataframe to Excel. To write the football dataframe we created above:

- football.to_excel('football.xlsx', index=False)

Note that, we did not write the index from the dataframe as it is meaningless. To verify the spreadsheet has been created, we could issue a directory listing command using the ! character (in Linux for example)

- !ls -l *.xlsx

Reading from Excel

In the most basic use-case, read_excel takes a path to an Excel file, and the sheet name indicating which sheet to parse.

- football = pd.read_excel('football.xlsx', 'Sheet1’)

Above command uses the Pandas read method to repopulate the football dataframe from sheet 1 of the spreadsheet.

I/O with Excel

- To facilitate working with multiple sheets from the same file, the ExcelFile class can be used to wrap the file and can be passed into read_excel method.
- There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

```
two_sheet_xlFile = pd.ExcelFile('multi_sheets.xlsx')
df = pd.read_excel(two_sheet_xlFile , 'Sheet1')
```

Assumption: multi_sheets.xls is an excel file created in the current working directory with two sheets

I/O with Excel

The ExcelFile class can also be used as a context manager.

```
with pd.ExcelFile(‘mutli_sheets.xlsx') as xls:
        df1 = pd.read_excel(xls, 'Sheet1')
        df2 = pd.read_excel(xls, 'Sheet2’)
```

```
>>> print(df1)                    # print the data frames
>>> print(df2)
```

I/O with Excel

- The primary use-case for an ExcelFile is parsing multiple sheets with different parameters

Example: when Sheet1's format differs from Sheet2

```
data = {}      # an empty dictionary

with pd.ExcelFile('multi_sheets.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=1)
```

- Note that if the same parsing parameters are used for all sheets, a list of sheet names can simply be passed to read_excel with no loss in performance.

I/O with Excel

- using the ExcelFile class when formats of sheets are same

```
data = {}

with pd.ExcelFile(‘multi_sheets.xls') as xls:
        data['Sheet1'] = read_excel(xls, 'Sheet1', index_col=None, na_values=['NA'])
        data['Sheet2'] = read_excel(xls, 'Sheet2', index_col=None, na_values=['NA'])
```

- Equivalent using the read_excel function can be

```
data = read_excel('path_to_file.xls', ['Sheet1', 'Sheet2'], index_col=None, na_values=['NA'])
```

Working with multiple sheets

- The second argument in read_excel() is sheetname, not to be confused with ExcelFile.sheet_names.
- An ExcelFile's attribute sheet_names provides access to a list of sheets.
- The arguments sheetname allows specifying the sheet or sheets to read.
- The default value for sheetname is 0, indicating to read the first sheet
- Pass a string to refer to the name of a particular sheet in the workbook.
- Pass an integer to refer to the index of a sheet. Indices follow Python convention, beginning at 0.
- Pass a list of either strings or integers, to return a dictionary of specified sheets.
- Detailed documentation online excel_files_pandas http://pandas.pydata.org/pandas-docs/stable/io.html#excel-files