Chapter 14 - Pandas

14 Pandas

Pandas is an open source Python library that is used for data manipulation and analysis of large amounts of data especially specializing with tabular data. We will be learning how to prepare, clean and transform data for plotting. We will need to import the Pandas package into our workspace like so `import pandas as pd`. This chapter comes with a reference Jupyter Notebook file. Please refer to it for more examples.

14.1 Data Structures: Series and DataFrame

Pandas has 2 main data structures, namely Series and DataFrame.

- **Series** - is a 1 dimensional homogeneous array. It's size is **immutable** (meaning cannot be changed).

  An example of a **Series** structure with homogeneous data is as follows

5	8	50	9	40	69	20	14

  Points to note about the Series structure:

  - Homogeneous Data (ie data of the same type)
  - Size is Immutable (size is 8 for the example above)
  - Values are changeable (mutable)
- **DataFrame** - is a 2 dimensional heterogeneous typed columns. It's size is **mutable** and it has a tabular structure.

  An example of a **DataFrame** structure with heterogeneous data is

	Name	Broadcaster	Genre	Ratings
0	Grimm	NBC	Supernatural	7.8
1	Game of Thrones	HBO	Fantasy	9.3
2	Grey's Anatomy	ABC	Medical	7.6
3	Siren	Freeform	Fantasy	7

  From the example above we can see that the table represents the ratings of several American TV series. The data is represented in a tabular format with each column representing an attribute and each row representing a TV series. The data in the table is mixed of different types with column *Ratings* of type float and the rest are strings. The left most unnamed column is the *index*. The *index* is automatically created when Pandas creates a dataframe.

  Points to note about DataFrame structure:

  - Heterogeneous Data
  - Size is Mutable (able to add more rows of data)
  - Values are changeable (mutable)

14.2 Object Creation

We shall now look at how the 2 data structures are created using Pandas, NumPy and loading data from an external `csv` file.

14.2.1 Series

```
import pandas as pd
import numpy as np

# creating a numpy array
data = np.array(['a','b','c','d'])
# creating a series from the numpy array
s = pd.Series(data)
print(s)
```

the output is

```
0    a
1    b
2    c
3    d
dtype: object
```

From the output, the first column is the indexes where the range is `len(data)-1` and the second column are the values of the NumPy array. The indexes can be changed to a fixed number range by adding an `index` parameter and value to the `Series` function like so `pd.Series(data, index=[5,6,7,8])`. Series can also be created with `dictionary` and scalar values.

14.2.2 DataFrame

A DataFrame can be created from Python lists, dictionaries, Series, NumPy `ndarrays` and another DataFrame.

```
import pandas as pd

# Creating from Python lists
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

the output is

```
   0
0  1
1  2
2  3
3  4
4  5
```

From the output, the first column is the indexes of the rows and the second column has a `0` as the header for the values from the Python list. A better representation would be to pass in column names like so `pd.DataFrame(data,columns=['values'])`. For data from Python dictionary and Series, Pandas will use the dictionary keys and Series index (if provided) as column names.

14.2.3 From External Files

So far the data sets that we have created for the Series and DataFrames are small but what happens when the data is stored in an external file and its size is huge? That's when we have to use Pandas to load the data from either a CSV file, a JSON file or SQL database. We will be using CSV files for this unit as JSON files are similar in structure but larger in size and SQL data requires a database connection which at this point, we do not have. Do keep in mind that read data is stored in your computer's memory therefore when reading a huge dataset, make sure your computer has enough memory and read the data in chunks.

The CSV file that we will be using is called `EdStatsCountry.csv` and it should be packaged along with this handout. This file contains data on education statistics by country from the World Bank, further information about the dataset can be found here. You may like to view this file via a spreadsheet software, to see what kind of data we will be working with.

To load an external CSV file into Pandas, use the function `read_csv()` and pass it the full file name including its path if it is not located in the same directory as your workspace. Be careful about using the `print()` function to print a large dataset to screen as it can take up a lot of memory. Instead in the next section, we will learn how to view some parts of a loaded large dataset.

```
# for CSV files with headers on the first row
df = pd.read_csv('EdStatsCountry.csv', header=0)

# for JSON files with headers on the first row
df = pd.read_json('file_name.json')
```

14.3 Viewing Data

After creating the data structures, we occasional would like to view some of the attributes of the data set to check for correctness. The properties of (general) interests are listed in the table below

Attribute/Method	Description
axes	The overall stats of the dataset like Header name, start to end row numbers, step size and dtype
dtype	dtype of the object
ndim	Number of dimensions of the data, by default it is 1 for Series
size	Number of elements `([rows * columns] - header_columns)` of this data set
`head()`	Returns the first `n` rows
`tail()`	Returns the last `n` rows

```
# for CSV files with headers on the first row
df = pd.read_csv('EdStatsCountry.csv', header=0)
df.axes
df.head(2)
df.tail(1)
```

Please refer to the lab notebook for the output of these commands.

14.4 Adding more data to DataFrames

After loading a dataset to memory, we may need to add more columns, rows or even append another dataset to the current dataframe.

- adding rows to an existing dataframe is equivalent to appending another dataset. It uses the `append()` function from the DataFrame object or the `concat()` function from the main Pandas package, both functions accepts either a Python dictionary or another Pandas dataframe object. If there is a need for the record indexes to be consecutive, set the `ignore_index` argument to `True`. Note that if the dataframes to add/combine have different columns, Pandas will in used the `Nan` object to fill in the missing data.

```
df1 = pd.DataFrame([[1, 2], [3, 4]], columns=['A','B'])
df2 = pd.DataFrame([[54, 28], [87, 96]], columns=['A','B'])
df1.append(df2, ignore_index=True)
```

- adding a column to a dataframe. This is achieved using the same technique as adding another `key` to a Python dictionary. The value can either be a Python List or a Pandas Series or DataFrame. Note that any extra elements beyond the maximum row index is omitted. To add it in, use the above `append()` or `concat()` methods.

```
df1 = pd.DataFrame([[1, 2], [3, 4]], columns=['A','B'])
df1['C'] = [5,6 ]
```

14.5 Selection

From NumPy and Python, we have been exposed to the indexing operators using `[]` and the dot `.` operator to gain quick access to the data. Those operators are also usable with Pandas but Pandas does provide optimized data access methods for us to use.

Indexing Method	Description
`.loc[]`	Label based
`.iloc[]`	Integer based

The inputs for both `.loc` and `iloc` are the same in that it is of the form `[row,col]` which is used to access a group of rows and columns by label or integer or boolean array. However, their output are different. The function's input properties are as follows

- Single label/integer (index label not value)
- A list or array of labels/integers
- A slice of object with labels/integers

- A boolean array of the same length as the axis being sliced
- A `callable` function with 1 argument calling a Series or DataFrame and returns a valid output for the indexing

Example

```
# .loc[]
# getting the rows from 0 to 5 (including 5)
# with the columns 'Region'
df.loc[0:5, 'Region']

# .iloc[]
# getting first 3 rows of data, returns in a DataFrame format
df.iloc[:3]

# .at[]
# getting the cell value at row 5, col 'Currency Unit'
df.at[4, 'Currency Unit']
```

After a selection has been made, the values of the DataFrame can be updated as required

```
# getting the cell value at row 5, col 'Currency Unit'
df.at[4, 'Currency Unit'] = 'USD'
```

14.6 Sorting

Sorting of data is done either using column labels or indexes.

Example

```
import pandas as pd
import numpy as np

# unsorted data via index
unsorted_df = pd.DataFrame(np.random.randn(10,2),
                           index=[1,4,6,2,3,5,9,8,0,7],
                           columns=['col2','col1'])

# sorting via index, default is ascendng order
unsorted_df.sort_index()
# sorting via index, descendng order
unsorted_df.sort_index(ascending=False)

# sorting via values of col1
unsorted_df.sort_values(by='col1')
```

14.7 Missing Data

Missing data can happens when there is a need to add more rows to the DataFrame without values (values could be added later) or when there's no data in a cell from external sources. Missing Data is represented as `NaN` in a DataFrame cell.

Depending on business logic, missing data can be handled in a variety of ways

- Replacing them with a scalar/generic/specific value

- Pad or fill the missing data with values from the rows above or below respectively
- Drop (remove) the whole row

```
missing_data = pd.DataFrame(np.random.randn(5, 3),
                            index=['a', 'c', 'e', 'f','h'],
                            columns=['one', 'two', 'three'])

# rematch the data to the new indexes, any new rows created will have NaN in them
missing_data = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

# replaceing with scalar 0 on all NaN values
missing_data.fillna(0)

# filling all the NaN values with the values from the row above
missing_data.fillna(method='pad')

# drop the rows with NaN values
missing_data.dropna()
```

14.8 Other useful functions

- `isnull()` - coupled with `sum()` counts the number of `NaN` objects in each column of a dataframe.
- `mean()`, `std()`, `describe()` - statistics functions for either the whole dataframe or a single column.
- `astype()` - changes all columns or specific columns to another datatype.
- `merge()` - merge DataFrames using the same techniques as the standard database join operations.
- `groupby()` - splitting the DataFrame into groups.
- `series.unique()` - getting all unique values from a series. A dataframe column is a Series.
- `reset_index()` - reset the indexes of a DataFrame. Useful when a dataframe is sorted by another column and needs the indexes renumbered.

14.9 Matplotlib with Pandas

When working with larger datasets or external data, this data is generally loaded using a library that can handle processing large datasets. During the processing phase, the data will be transformed from a raw messy data to smaller tidy datasets and more often that not, these datasets are plotted to help infer some hypothesis.

These tidy datasets may or may not be stored in CSV files. If they are in CSV files (external files), the data has to be loaded into a Pandas dataframe before processing. For instance, a tidy dataset can be data of the weather of the State of Philadelphia over 1 year period (July 2014 to June 2015) which is used in the example below. More information about this data can be found here. This example below shows the initial plot of a column of data from the dataset without processing.

```
# loading the data into Pandas dataframe
weather_data = pd.read_csv('Weather_data_Philadelphia.csv', header=0)

# change the date from string to datetime format (optional)
weather_data['date'] = pd.to_datetime(weather_data['date'], format='%Y-%m-%d')

# increase the plot's width and height in inches
fig = plt.figure(figsize=(15,8))

# Plot some data on the axes.
plt.plot(weather_data['date'], weather_data['actual_min_temp'])
plt.xlabel('Date')
plt.ylabel('Temperature (F)')
plt.title('Actual Minimum Temperature per day from July 2014-2015')
plt.show()
```

the output is shown in figure 1 below.

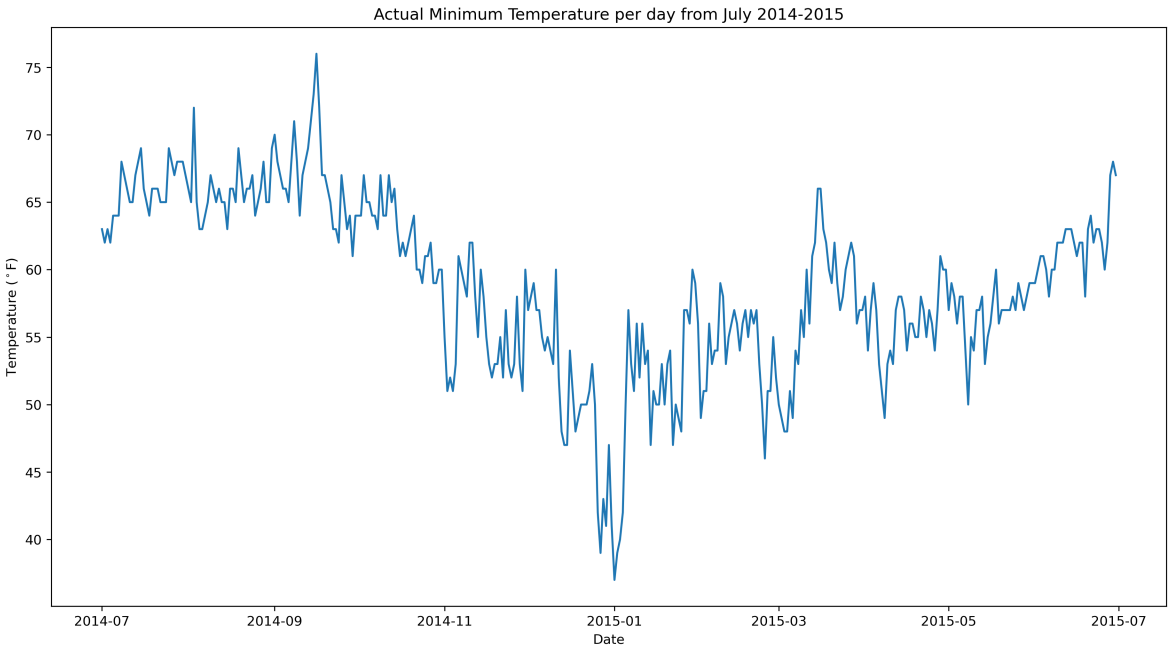

Figure 1: Line chart of the actual minimum temperature of the State of Philadelphia from July 2014 to 2015.

Just plotting the a single column data from a dataset will not allow scientists to infer much of anything but when a combination of data is plotted after some processing, hypotheses can be made. The graph below (figure 2) is a combination of the 5 columns of data from the same weather dataset for the State of Philadelphia over a year. The codes to plot this graph is in the lab session notebook.

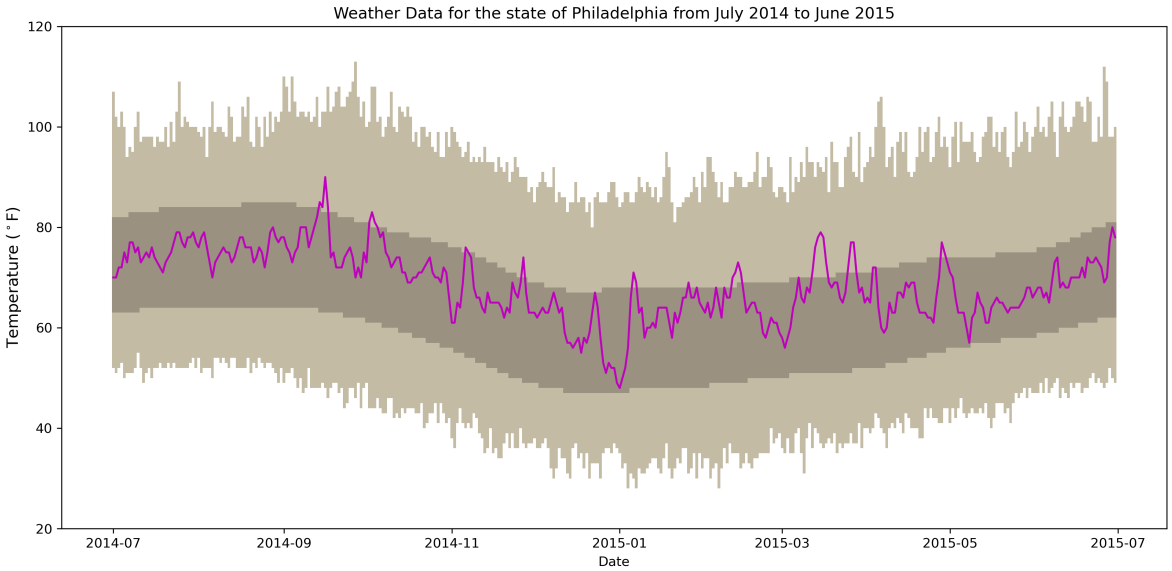

Figure 2: Weather data for the State of Philadelphia from July 2014 to 2015.

The parts of the plots is as follows

- **light brown** - the difference between recorded temperature on the day since 1880
- **dark brown** - the difference between average temperature on the day since 1880
- **purple line** - the actual mean temperature on the day

From this graph, information like sudden spikes in temperatures can be easily seen when compared with the average temperature since 1880. Sudden temperature pikes could be indication of climate change. With more weather data from the surrounding states, it could even help explain the sudden drop in temperatures in Jan 2015.

14.10 References

1. API reference, https://pandas.pydata.org/pandas-docs/stable/reference/index.html
2. Hilpisch, 2018, 'Chapter 5. Data Analysis with pandas' in Python for Finance, 2nd Edition, O`Reilly Media, Inc.

14.10.1 Data Sets

1. https://datacatalog.worldbank.org/dataset/education-statistics
2. https://github.com/fivethirtyeight/data