Python: Pandas Course Notes

Klein
carlj.klein@gmail.com

1 Learning Objectives

Concepts

- Creating Pandas Series
- Accessing Elements in Pandas Series
- Arithmetic Operations on a Pandas Series
- Creating Pandas DataFrames
- Exercise: Manipulate a Pandas Series
- Accessing Elements in Pandas DataFrames
- Dealing with NaN
- Loading Data Directly Into a Pandas DataFrame
- Exercise: Manipulate a Pandas DataFrame
- Exercise: Statistics from Stock Data

Commands

- pd.Series
- .shape
- .ndim
- .size
- .index
- .values
- in
- pd.DataFrame
- .loc

- .iloc
- .append
- .insert
- .pop
- .drop
- .isnull()
- .sum()
- .any()
- .isnull().sum()
- .isnull().sum().sum()
- .isnull().any()
- .dropna
- .fillna
- .interpolate
- pd.set_option
- .describe()
- General Statistics
  - .mean()
  - .median()
  - .std()
  - .corr()
- pd.date_range
- .rename
- .join
- .rolling

2 Creating Pandas Series

With Pandas Series we can pair data with index labels. We'll use a shopping list for an example, which pairs different data types with labels of the groceries required.

```
import pandas as pd
groceries = pd.Series(data = [30, 6, 'Yes', 'No'],
index = ['eggs', 'apples', 'milk', 'bread'])

"""
eggs        30
apples       6
milk       Yes
bread       No
dtype: object
"""
```

After we've created the Pandas Series object, there are some essential functions we can call on it. Namely,

- shape: will return size of each dimensions
- ndim: will return the number of dimensions
- size: will return the number of elements in the series
- index: will return a tuple of the index values
- values: will return a tuple of the data values
- in: use to check if a certain value is an index label

Continuing from our example, we'll see these commands in action.

```
groceries.shape
# (4,)

groceries.ndim
# 1

groceries.size
# 4

groceries.index
# Index(['eggs', 'apples', 'milk', 'bread'], dtype='object')

groceries.values
# array([30, 6, 'Yes', 'No'], dtype=object)

'eggs' in groceries
# True
'fruit' in groceries
# False
```

3 Accessing Elements in Pandas Series

We have the information in the Pandas Series, now what can we do with it? If we specify label(s) or indices, we can find the associated value(s).

```
groceries['eggs']
# 30

groceries[['eggs', 'bread']]
"""
eggs      30
bread     No
dtype: object
"""

groceries[0]
# 30
```

By using a list of indices, we have their associated values returned. Lists of indices can be generated following numeric patterns, so it could be an efficient method of extracting values.

We can also change elements by specifying labels, or drop them completely. Dropping the element uses a new function:

- drop: will delete an element in Pandas Series (specify to alter original)

```
# changing elements
groceries['eggs'] = 2

# deleting elements
groceries.drop('apples') # this will not modify the original
                                series
groceries.drop('apples', inplace = True) # this will modify the
                                original series
```

4 Arithmetic Operations on a Pandas Series

Arithmetic operations can be performed on the entire series. To an extent, the operation will even extend to series of mixed values. Covered in a different set of notes, mathematical functions from NumPy can also be applied to series.

```
groceries * 2
"""
eggs           60
apples         12
milk       YesYes
bread        NoNo
"""
```

Certain scenarios absolutely call for the repetition of string values, but there is a limit to what mathematical operations can performed on mixed data (string values for instance). Multiplication obviously works, but operations like division reach the logical limit and cannot be performed.

5 Exercise: Manipulate a Pandas Series

In this exercise, we're given a list of planets and a list of their distances from the sun in units of 10^6 km. We're tasked with creating 3 separate series from this data:

1. Create a Pandas Series using the data called "dist_planets".
2. Create a Pandas Series called "time_light" which calculates the time (in minutes) it takes the light from the Sun to reach each planet. You can do this by dividing each planet's distance from the Sun by the speed of light. Use c = 18 for the speed of light, since light travels at 18 x 10^6 km/minute.
3. Use Boolean indexing to create a Pandas Series called "close_planets" that contains only those planets which sunlight takes less than 40 minutes to reach.

```
planets = ['Earth', 'Saturn', 'Venus', 'Mars', 'Jupiter']
distance_from_sun = [149.6, 1433.5, 108.2, 227.9, 778.6]

# 1) Turn the data into a Pandas Series
dist_planets = pd.Series(distance_from_sun, planets)

# 2) Calculate the time it takes for light to reach them
time_light = dist_planets / 18

#3) Calculate which planets are reached by light in under 40
                                 minutes
close_planets = time_light[time_light < 40]

"""
Earth      8.311111
Venus      6.011111
Mars      12.661111
dtype: float64
"""
```

6 Creating Pandas DataFrames

DataFrames are truly where Pandas excels. Think of a DataFrame as a really powerful spreadsheet. A DataFrame is an object which contains multiple Series.

Let's begin with an example of individuals shopping around for numerous items.

Using a Pandas Series, let’s put their item names as the labels and respective prices as the values. Next, we’ll create a dictionary which connects the individual’s name with their shopping cart. Finally, we’ll use the dictionary to create a DataFrame.

```
bob_cart = pd.Series(data = [245, 25, 55], index = [’bike’, ’
                                pants’, ’watch’])
alice_cart = pd.Series(data = [40, 110, 500, 45], index = [’book’
                                , ’glasses’, ’bike’, ’pants’])
items = {’Bob’ : bob_cart, ’Alice’ : alice_cart}

shopping_carts = pd.DataFrame(items)
```

Depending on the Python IDE used to run the code, the formatting of the output will differ slightly, however the general structure is as follows:

	Bob	Alice
bike	245.0	500.0
book	NaN	40.0
glasses	NaN	110.0
pants	25.0	45.0
watch	55.0	NaN

Note how the individuals had some different items than each other. The DataFrame was able to recognize this, placing NaN values where there was no data. Additionally, both the column and row labeling can be omitted, in which case they will be represented by indices.

A number of commands and functions we’ve seen with Series are applicable with DataFrames as well.

```
shoppping_carts.shape
# (5, 2)

shoppping_carts.ndim
# 2

shoppping_carts.size
# 10
```

There are many similarities in accessing, extracting, and editing the information inside a DataFrame to the methods we examined for Series, but that’ll be the main topic in the next section. Before we move on to slicing already built DataFrames, it’s important to address that data queries can be built in during the creation of DataFrames.

Recall the creation of the ”items” dictionary in which Bob’s and Alice’s shopping carts were stored, prior to creating the ”shopping_carts” DataFrame. We’ll build off the ”items” dictionary to show alternative ways to create DataFrames with subsets of the data.

```
# column subset
pd.DataFrame(items, columns = ['Bob'])

# index subset
pd.DataFrame(items, index = ['pants', 'book'])

# column and index subset
pd.DataFrame(items, index = ['glasses', 'bike'], columns = ['
                    Alice'])
```

Column Subset

	Bob
bike	245
pants	25
watch	55

Index Subset

	Bob	Alice
pants	25.0	45
book	NaN	40

Column and Index Subset

	Alice
glasses	110
bike	500

We know that if we have the omit either the column or row labeling, we'll just have indices as the labeling. What if while we were building the DataFrame, we decided that the data needed more labeling? Let's take a look at two examples where we pass an entirely new index to datasets that didn't previously have this information. We'll use the more generic terms of data and df for these examples as well.

```
# Example 1
data = {'Integers': [1, 2, 3], 'Floats': [4.5, 8.2, 9.6]}
df = pd.DataFrame(data, index = ['label 1', 'label 2', 'label 3']
                    )

# Example 2
data = [{'bikes': 20, 'pants': 30, 'watches': 35},
{'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]
df = pd.DataFrame(data, index = ['store 1', 'store 2'])
```

Example 1

	Integers	Floats
label 1	1	4.5
label 2	2	8.2
label 3	3	9.6

Example 2

	bikes	pants	watches	glasses
store 1	20	30	35	NaN
store 2	15	5	10	50.0

7 Accessing Elements in Pandas DataFrames

After a DataFrame object has been created, it's possible to access rows, columns and elements of the DataFrame. Apart from how to zoom into the data, this section introduces a few more essential functions. We'll touch on:

- loc: slices the DataFrame by passing the name of columns or rows through loc, can accept boolean data (i.e. can select data according to some conditions)
- iloc: indexed based selecting method, cannot accept boolean data
- append: will add an object to the end of a dataset (DataFrame to DataFrame in these examples)
- insert: will add an object at a specified index to a dataset
- pop: will remove specified columns, as well as isolate the specified columns into a new DataFrame
- drop: will remove specified columns or rows, must declare axis (columns: axis = 1, rows: axis = 0)
- rename:

First, let's recreate a previous example and name it something more applicable such as "store_items".

```
data = [{'bikes': 20, 'pants': 30, 'watches': 35},
{'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]
store_items = pd.DataFrame(data, index = ['store 1', 'store 2'])
```

The following example shows a few ways to select data from a DataFrame:

```
# Example 1: Columns
store_items[['bikes', 'pants']]

# Example 2: Using loc to extract either columns or rows by name
store_items.loc[['store 1']]

# Example 3: Element extraction ([column][row])
store_items['bikes']['store 2']
```

Example 1

	bikes	pants
store 1	20	30
store 2	15	5

Example 2

	bikes	pants	watches	glasses
store 1	20	30	35	NaN

Example 3

Returns a single element, which is 15.

We've seen a few different methods for slicing and dicing the data with a Pandas DataFrame. Now, we'll look at adding, editing, and deleting the data.

First, we'll look at a few ways to add data:

```
# Example 1: Simple Method to Add a Column
# Add the column 'shirts' to the DataFrame with prices for both
                                stores
store_items['shirts'] = [15, 2]

# Example 2: Create Column from Current Columns
# Add the column 'suits' as the sum of the prices for 'shirts'
                                and 'pants'
store_items['suits'] = store_items['shirts'] + store_items['pants
                                ']

# Example 3: Using append to Add New Row
# Create a new store and append it to the existing DataFrame
new_items = [{'bikes': 20, 'pants': 30, 'watches': 35, 'glasses':
                                 4}]
new_store = pd.DataFrame(new_items, index = ['store 3'])
store_items = store_items.append(new_store)

# Example 4: Using insert to Add a Column
# Insert a new column 'shoes' before the 5th index in the
                                existing DataFrame
store_items.insert(5, 'shoes', [8, 5, 0])
```

Now, a few ways to delete data:

```
# Example 1: Use pop to Remove a Column
# Remove 'watches' from the DataFrame
store_items.pop('watches')

# Example 2: Use drop to Remove Columns and Rows
# Must declare axis to specify columns (axis=1) vs rows (axis=0)
# Remove the items 'suts' and 'shoes'
# Remove the stores 'store 1' and 'store 2'
store_items = store_items.drop(['suits', 'shoes'], axis=1)
store_items = store_items.drop(['store 1', 'store 2'], axis=0)
```

Throwing one more editing technique into this section, how to rename columns or rows. A solid method to accomplish a name change is to use a dictionary setup.

```
store_items = store_items.rename(index = {'store 3': 'last store'
                                })
```

8 Dealing with NaN

Most data in the real world doesn't come ready to analyze, and requires cleaning. One of the most common errors that will need to be addressed is missing values. A few essential functions covered will be:

- isnull: will assign a boolean value True or False for every element in a DataFrame (can be useful to combine with the sum function)
- count: useful in many cases, but can be used in locating NaN values by discrepancies in column counts
- any: will return boolean values if certain conditions are met
- dropna: with very large data sets, it can be acceptable to completely disregard data with NaN values by deleting either rows or columns (must specify axis)
- fillna: use the data surrounding NaN values to replace the NaN according to the rules specified
- interpolate: use the data surrounding NaN values to follow patterns such as regressions replace the NaN values

Let's return to the example of examining prices across a few stores. We'll make up some data for three stores, where not all stores have the same items.

```
items = [{'bikes': 20, 'pants': 30, 'watches': 35, 'shirts': 15,
                                'shoes':8, 'suits':45},
{'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5, 'shirts':
                                2, 'shoes':5, 'suits':7},
{'bikes': 20, 'pants': 30, 'watches': 35, 'glasses': 4, 'shoes':
                                10}]

store_items = pd.DataFrame(items2, index = ['store 1', 'store 2',
                                'store 3'])
```

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	NaN	10	NaN	4.0

With a small dataset, it's easy to locate the NaN values. However, some datasets will have hundreds, thousands, or millions upon millions of elements. In those cases, it's not as easy to find the missing values by visual queue alone. Before we do anything with the missing values, we need a method to find them.

```
# isnull() can be used to find the missing values
# if there are missing values, there are a few ways to illustrate
                                them
```

```
# We can have the DataFrame returned with True and False values
store_items.isnull()

# We can have a DataFrame with a single column returned with the
                                number of missing values
store_items.isnull().sum()

# The count function works as a sort of the complement to the
                                above
# If the values aren't equivalent across the column returns, then
                                 there is missing data
store_items.count()

# similar to the above two functions, isnull combined with any
                                will return a single column
                                showing if any of the items have
                                missing values throughout the
                                differnt stores
store_items.insull().any()

# Taking another sum of the null column returned will give the
                                total number of missing values
store_items.isnull().sum().sum()
```

isnull()

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	False	False	False	False	False	False	True
store 2	False	False	False	False	False	False	False
store 3	False	False	False	True	False	True	False

isnull().sum()

	0
bikes	0
pants	0
watches	0
shirts	1
shoes	0
suits	1
glasses	1

count()

	0
bikes	3
pants	3
watches	3
shirts	2
shoes	3
suits	2
glasses	2

Once it has been confirmed there are missing values in the dataset, they can't be left in there! We'll show a few methods for dealing with these data sets. Note that with massive datasets, missing values are common and could be negligible enough to the results to simply delete them. However the data is dealt with, it's important to state how they were dealt with in the final analysis.

```
# we can drop both entire rows and columns that contain NaN
                                values
# drop rows (axis = 0), drop columns (axis = 1)

# we can completely drop any rows (i.e. stores) that have missing
                                 values
store_items.dropna(axis =0)
# we can completely drop any columns (i.e. items) that have
                                missing values
store_items.dropna(axis=1)

# note that the above examples do not modify the original
                                DataFrame unless inplace is
                                specified
store_items.dropna(axis=1, inplace = True)


# instead of deleting data, we can choose to fill the NaN values
                                a number of ways
# fillna is a useful function to do this
# note that there are more methods than illustrated here

# replace the NaN values with 0
store_items.fillna(0)

# keep the following in mind for the next few examples:
# forwarding filling: replaces each NaN with the value prior to
                                NaN
# forward filling won't affect any elements in first row / column
# switch row and column using axis specification
# backward filling won't affect any elements in last row / column

# forward fill using rows
store_items.fillna(method = 'ffill', axis = 0)

# forward fill using columns
store_items.fillna(method = 'ffill', axis = 1)

# backward fill using rows
store_items.fillna(method = 'backfill', axis = 0)

# backward fill using columns
store_items.fillna(method = 'backfill', axis = 1)


# instead of using the exact surrounding numbers in place of the
                                NaN values, we can apply a
                                mathematical filter using the
                                surrounding numbers
# we can use interpolate, which also has plenty more options than
                                 what is shown
```

```
store_items.interpolate(method = 'linear', axis = 1)
```

Forward Fill - Rows

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	2.0	10	7.0	4.0

Forward Fill - Columns

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20.0	30.0	35.0	15.0	8.0	45.0	45.0
store 2	15.0	5.0	10.0	2.0	5.0	7.0	50.0
store 3	20.0	30.0	35.0	35.0	10.0	10.0	4.0

Backward Fill - Rows

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	50.0
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	NaN	10	NaN	4.0

Backward Fill - Columns

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20.0	30.0	35.0	15.0	8.0	45.0	NaN
store 2	15.0	5.0	10.0	2.0	5.0	7.0	50.0
store 3	20.0	30.0	35.0	10.0	10.0	4.0	4.0

Interpolate - Linear

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20.0	30.0	35.0	15.0	8.0	45.0	45.0
store 2	15.0	5.0	10.0	2.0	5.0	7.0	50.0
store 3	20.0	30.0	35.0	22.5	10.0	7.0	4.0

9 Loading Data Directly Into a Pandas DataFrame

As stated previously, a Pandas DataFrame is essentially a very powerful spreadsheet, so it should come as no surprise that data from other forms of spreadsheets can be loaded directly in. Take the example of a CSV file. Additionally, we'll use this example to demonstrate a few essential statistical functions:

- describe: returns a number of useful statistics:
  - count
  - mean
  - std
  - min

  - 25%
  - 50%
  - 75%
  - max
  - name

- max
- min
- mean
- Many statistics can be called directly, not just those that are combined with passing the describe() function

```
df = pd.read_csv('file.csv')

df.describe()
df.max()
df.mean()
df.std()
# The statistical examples could continue...
# Apart from the entire dataset, it's useful to know statistcs on
                                  individual columns
df['Column A'].describe()
```

10 Exercise: Manipulate a Pandas DataFrame

In this exercise we're given Series of books, Series of authors, and then Series of different users' ratings on the books.

The task is to create a dictionary using the data, place the dictionary into a DataFrame, and then replace any missing values with the average rating from each column.

```
import numpy as np
# Set the precision of our dataframes to one decimal place.
pd.set_option('precision', 1)

# Books
books = pd.Series(data = ['Great Expectations', 'Of Mice and Men'
                                , 'Romeo and Juliet', 'The Time
                                Machine', 'Alice in Wonderland' ]
                                )

# Authors
authors = pd.Series(data = ['Charles Dickens', 'John Steinbeck',
                                'William Shakespeare', ' H. G.
                                Wells', 'Lewis Carroll' ])
```

```
# Ratings
user_1 = pd.Series(data = [3.2, np.nan ,2.5])
user_2 = pd.Series(data = [5., 1.3, 4.0, 3.8])
user_3 = pd.Series(data = [2.0, 2.3, np.nan, 4])
user_4 = pd.Series(data = [4, 3.5, 4, 5, 4.2])

# Dictionary
dat = {'Author': authors, 'Book Title': books, 'User 1': user_1,
  'User 2': user_2, 'User 3': user_3, 'User 4': user_4}

# DataFrame
book_ratings = pd.DataFrame(dat)

# Replace missing values with the average values
book_ratings.fillna(book_ratings.mean(), inplace = True)
```

Book Ratings

	Author	Book Title	User 1	User 2	User 3	User 4
0	Charles Dickens	Great Expectations	3.20	5.000	2.000000	4.0
1	John Steinbeck	Of Mice and Men	2.85	1.300	2.300000	3.5
2	William Shakespeare	Romeo and Juliet	2.50	4.000	2.766667	4.0
3	H. G. Wells	The Time Machine	2.85	3.800	4.000000	5.0
4	Lewis Carroll	Alice in Wonderland	2.85	3.525	2.766667	4.2

11 Exercise: Statistics from Stock Data

This next example uses massive datasets featuring stocks from major companies Google, Apple and Amazon. We're interested in the "Date" and "Adj Close" columns. We will have the "Date" column as our row index, where we'll have the DataFrame recognize the dates as actual dates (year/month/day) and not as strings. All of these requirements can be accomplished while loading the data. Here are some hints:

- Use the index_col keyword to indicate which column you want to use as an index. For example index_col = ['Open']
- Set the parse_dates keyword equal to True to convert the Dates into real dates of the form year/month/day
- Use the usecols keyword to select which columns you want to load into the DataFrame. For example usecols = ['Open', 'High']

We're going to load in the datasets and perform some exploratory analysis on them.

```
# The filepaths for the companies we're previously saved values:
# goog_path: file path for Google
# aapl_path: file path for Apple
# amzn_path: file path for Amazon
# Load the data using the read_csv features hinted at above
google_stock = pd.read_csv(goog_path, index_col = ['Date'],
                                 parse_dates = True, usecols = ['
                                 Date', 'Adj Close'])
apple_stock = pd.read_csv(aapl_path, index_col = ['Date'],
                                 parse_dates = True, usecols = ['
                                 Date', 'Adj Close'])
amazon_stock = pd.read_csv(amzn_path, index_col = ['Date'],
                                 parse_dates = True, usecols = ['
                                 Date', 'Adj Close'])

# We create calendar dates between '2000-01-01' and  '2016-12-31'
dates = pd.date_range('2000-01-01', '2016-12-31')

# We create and empty DataFrame that uses the above dates as
                                 indices
all_stocks = pd.DataFrame(index = dates)

# preparing to join the datasets together

# Change the Adj Close column label to Google
google_stock = google_stock.rename(columns = {'Adj Close': '
                                 Google'})

# Change the Adj Close column label to Apple
apple_stock = apple_stock.rename(columns = {'Adj Close': 'Apple'}
                                 )

# Change the Adj Close column label to Amazon
amazon_stock = amazon_stock.rename(columns = {'Adj Close': '
                                 Amazon'})

# now time to join the dataframe
# We join the Google stock to all_stocks
all_stocks = all_stocks.join(google_stock)

# We join the Apple stock to all_stocks
all_stocks = all_stocks.join(apple_stock)

# We join the Amazon stock to all_stocks
all_stocks =all_stocks.join(amazon_stock)

# let's describe the stocks
all_stocks.describe()

# Dealing with NaN values
# Print the column-wise count of NaN values, if any, in the
                                 all_stocks dataframe
all_stocks.isnull().sum()
'''
Google    3095
Apple     1933
Amazon    1933
dtype: int64
'''
```

```
# Remove any rows that contain NaN values. Do this operation
                                inplace.
all_stocks.dropna(axis=0, inplace = True)

# let's get some basic statistics
# Print the average stock price for each stock
all_stocks.mean()
'''
Google    347.420229
Apple      47.736018
Amazon    216.598177
'''

# Print the median stock price for each stock
all_stocks.median()
'''
Google    286.397247
Apple      39.461483
Amazon    161.820007
'''

# Print the standard deviation of the stock price for each stock
all_stocks.std()
'''
Google    187.671596
Apple      37.421555
Amazon    199.129792
'''

# Print the correlation between stocks (result left out for
                                formatting issues)
all_stocks.corr()


'''
We will now look at how we can compute some rolling statistics,
                                also known as moving statistics.
We can calculate for example the rolling mean (moving average) of
                                 the Google stock price by using
                                the
Pandas dataframe.rolling().mean() method. The dataframe.rolling(N
                                ).mean() calculates the rolling
                                mean over
an N-day window. In other words, we can take a look at the
                                average stock price every N days
                                using
the above method. Fill in the code below to calculate the average
                                 stock price every 150 days for
Google stock
'''

rollingMean = all_stocks['Google'].rolling(150).mean()

# time to visualize the data
import matplotlib.pyplot as plt

# We plot the Google stock data
```

```
plt.plot(all_stocks['Google'])

# We plot the rolling mean ontop of our Google stock data
plt.plot(rollingMean)
plt.legend(['Google Stock Price', 'Rolling Mean'])
```

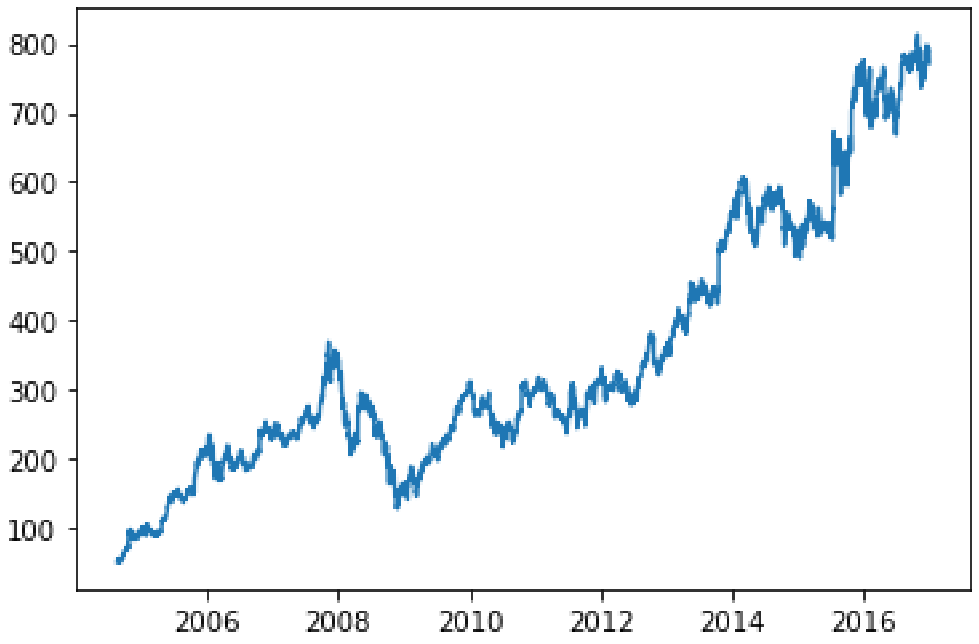

Figure 1: Google Stocks - Total

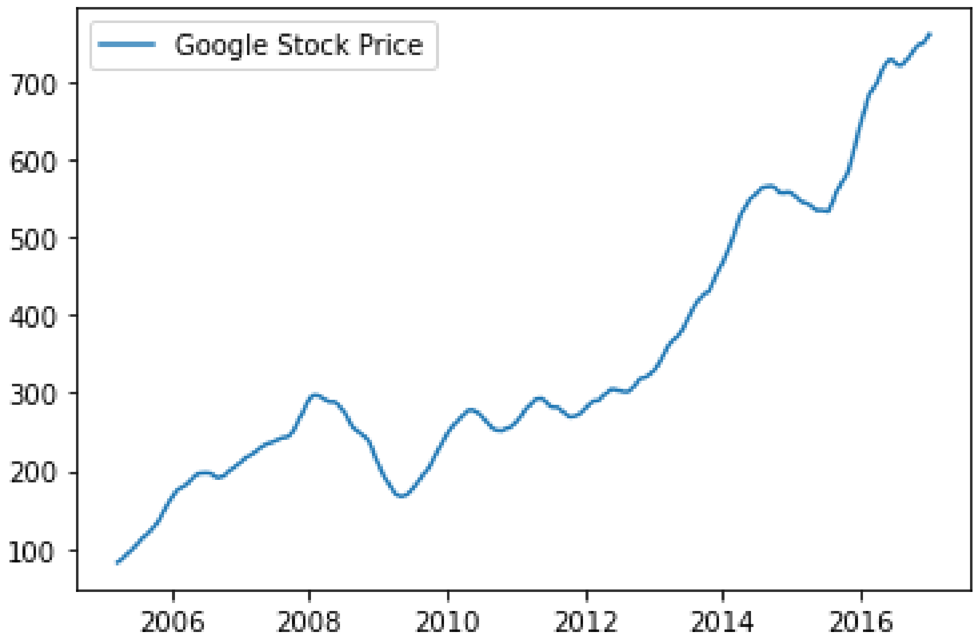

Figure 2: Google Stocks - Rolling