M.A.N.U

2/28/2023

Mathematics Association of Nairobi University
isaak@students.uonbi.ac.ke
https://github.com/Isaakkamau/MANU

# Creating Pandas DataFrames

Pandas DataFrames are two-dimensional data structures with labeled rows and columns, that can hold many data types. If you are familiar with Excel, you can think of Pandas DataFrames as being similar to a spreadsheet. We can create Pandas DataFrames manually or by loading data from a file. In this lesson, we will start by learning how to create Pandas DataFrames manually from dictionaries, and later we will see how we can load data into a DataFrame from a data file.

## Create a DataFrame manually

We will start by creating a DataFrame manually from a dictionary of Pandas Series. It is a two-step process:

1. The first step is to create the dictionary of Pandas Series.
2. After the dictionary is created we can then pass the dictionary to the `pd.DataFrame()` function.

We will create a dictionary that contains items purchased by two people, Alice and Bob, on an online store. The Pandas Series will use the price of the items purchased as data, and the purchased items will be used as the index labels to the Pandas Series. Let's see how this done in code:

In [1]:

```
# We import Pandas as pd into Python
import pandas as pd

# We create a dictionary of Pandas Series
items = {'Bob' : pd.Series(data = [245, 25, 55], index = ['bike', 'pants', 'watch']),
         'Alice' : pd.Series(data = [40, 110, 500, 45], index = ['book', 'glasses', 'bike', 'pants'])}

# We print the type of items to see that it is a dictionary
print(type(items))
```

```
<class 'dict'>
```

Now that we have a dictionary, we are ready to create a DataFrame by passing it to the `pd.DataFrame()` function. We will create a DataFrame that could represent the shopping carts of various users, in this case we have only two users, Alice and Bob.

**Example 1. Create a DataFrame using a dictionary of Series.**

In [2]:

```
# We create a Pandas DataFrame by passing it a dictionary of Pandas Series
shopping_carts = pd.DataFrame(items)

# We display the DataFrame
shopping_carts
```

Out[2]:

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

1. We see that DataFrames are displayed in tabular form, much like an Excel spreadsheet, with the labels of rows and columns in bold.
2. Also, notice that the row labels of the DataFrame are built from the union of the index labels of the two Pandas Series we used to

construct the dictionary. And the column labels of the DataFrame are taken from the keys of the dictionary.

3. Another thing to notice is that the columns are arranged alphabetically and not in the order given in the dictionary. We will see later that this won't happen when we load data into a DataFrame from a data file.
4. The last thing we want to point out is that we see some `NaN` values appear in the DataFrame. `NaN` stands for Not a Number, and is Pandas way of indicating that it doesn't have a value for that particular row and column index. For example, if we look at the column of Alice, we see that it has NaN in the watch index. You can see why this is the case by looking at the dictionary we created at the beginning. We clearly see that the dictionary has no item for Alice labeled watches. So whenever a DataFrame is created, if a particular column doesn't have values for a particular row index, Pandas will put a NaN value there.
    A. If we were to feed this data into a machine learning algorithm we will have to remove these `NaN` values first. In a later lesson, we will learn how to deal with `NaN` values and clean our data. For now, we will leave these values in our DataFrame.

In the example above, we created a Pandas DataFrame from a dictionary of Pandas Series that had clearly defined indexes. If we don't provide index labels to the Pandas Series, Pandas will use numerical row indexes when it creates the DataFrame. Let's see an example:

**Example 2. DataFrame assigns the numerical row indexes by default.**

In [3]:
```
# We create a dictionary of Pandas Series without indexes
data = {'Bob' : pd.Series([245, 25, 55]),
        'Alice' : pd.Series([40, 110, 500, 45])}

# We create a DataFrame
df = pd.DataFrame(data)

# We display the DataFrame
df
```

Out[3]:

| | Bob | Alice |
|---|---|---|
| 0 | 245.0 | 40 |
| 1 | 25.0 | 110 |
| 2 | 55.0 | 500 |
| 3 | NaN | 45 |

We can see that Pandas indexes the rows of the DataFrame starting from 0, just like NumPy indexes ndarrays.

Now, just like with Pandas Series we can also extract information from DataFrames using attributes. Let's print some information from our `shopping_carts` DataFrame

**Example 3. Demonstrate a few attributes of DataFrame**

In [4]:
```
# We print some information about shopping_carts
print('shopping_carts has shape:', shopping_carts.shape)
print('shopping_carts has dimension:', shopping_carts.ndim)
print('shopping_carts has a total of:', shopping_carts.size, 'elements')
print()
print('The data in shopping_carts is:\n', shopping_carts.values)
print()
print('The row index in shopping_carts is:', shopping_carts.index)
print()
print('The column index in shopping_carts is:', shopping_carts.columns)
```

```
shopping_carts has shape: (5, 2)
shopping_carts has dimension: 2
shopping_carts has a total of: 10 elements

The data in shopping_carts is:
 [[245. 500.]
 [ nan  40.]
 [ nan 110.]
 [ 25.  45.]
 [ 55.  nan]]

The row index in shopping_carts is: Index(['bike', 'book', 'glasses', 'pants', 'watch'], dtype='object')

The column index in shopping_carts is: Index(['Bob', 'Alice'], dtype='object')
```

When creating the `shopping_carts` DataFrame we passed the entire dictionary to the `pd.DataFrame()` function. However, there might be cases when you are only interested in a subset of the data. Pandas allows us to select which data we want to put into our DataFrame by means of the keywords `columns` and `index`. Let's see some examples:

In [5]:
```
# We Create a DataFrame that only has Bob's data
bob_shopping_cart = pd.DataFrame(items, columns=['Bob'])

# We display bob_shopping_cart
bob_shopping_cart
```

Out[5]:

| | Bob |
|---|---|
| bike | 245 |
| pants | 25 |
| watch | 55 |

In [6]:
```
# Example 4. Selecting specific rows of a DataFrame

# We Create a DataFrame that only has selected items for both Alice and Bob
sel_shopping_cart = pd.DataFrame(items, index = ['pants', 'book'])

# We display sel_shopping_cart
sel_shopping_cart
```

Out[6]:

| | Bob | Alice |
|---|---|---|
| pants | 25.0 | 45 |
| book | NaN | 40 |

In [7]:
```
#Example 5. Selecting specific columns of a DataFrame

# We Create a DataFrame that only has selected items for Alice
alice_sel_shopping_cart = pd.DataFrame(items, index = ['glasses', 'bike'], columns = ['Alice'])

# We display alice_sel_shopping_cart
alice_sel_shopping_cart
```

Out[7]:

| | Alice |
|---|---|
| glasses | 110 |
| bike | 500 |

You can also manually create DataFrames from a dictionary of lists (arrays). The procedure is the same as before, we start by creating the dictionary and then passing the dictionary to the `pd.DataFrame()` function. In this case, however, all the lists (arrays) in the dictionary must be of the same length. Let' see an example:

**Example 6. Create a DataFrame using a dictionary of lists**

In [8]:
```
# We create a dictionary of lists (arrays)
data = {'Integers' : [1,2,3],
        'Floats' : [4.5, 8.2, 9.6]}

# We create a DataFrame
df = pd.DataFrame(data)

# We display the DataFrame
df
```

Out[8]:

| | Integers | Floats |
|---|---|---|
| 0 | 1 | 4.5 |
| 1 | 2 | 8.2 |
| 2 | 3 | 9.6 |

In [9]:
```
#Example 7. Create a DataFrame using a dictionary of lists, and custom row-indexes (labels)

# We create a dictionary of lists (arrays)
data = {'Integers' : [1,2,3],
        'Floats' : [4.5, 8.2, 9.6]}

# We create a DataFrame and provide the row index
df = pd.DataFrame(data, index = ['label 1', 'label 2', 'label 3'])

# We display the DataFrame
df
```

Out[9]:

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

In [10]:
```
#Example 8. Create a DataFrame using a of list of dictionaries

# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35},
          {'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]
```

```
# We create a DataFrame
store_items = pd.DataFrame(items2)

# We display the DataFrame
store_items
```

Out[10]:

| | bikes | pants | watches | glasses |
|---|---|---|---|---|
| **0** | 20 | 30 | 35 | NaN |
| **1** | 15 | 5 | 10 | 50.0 |

In [29]:

```
#Example 9. Create a DataFrame using a of list of dictionaries, and custom row-indexes (labels)

# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35},
          {'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]

# We create a DataFrame  and provide the row index
store_items = pd.DataFrame(data = items2, index = ['store 1', 'store 2'])

# We display the DataFrame
store_items
```

Out[29]:

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

# Accessing Elements in Pandas DataFrames

In [30]:

```
#Example 1. Access elements using labels

# We print the store_items DataFrame
print(store_items)

# We access rows, columns and elements using labels
print()
print('How many bikes are in each store:\n', store_items[['bikes']])
print()
print('How many bikes and pants are in each store:\n', store_items[['bikes', 'pants']])
print()
print('What items are in Store 1:\n', store_items.loc[['store 1']])
print()
print('How many bikes are in Store 2:', store_items['bikes']['store 2'])
```

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

How many bikes are in each store:
          bikes
store 1     20
store 2     15

How many bikes and pants are in each store:
          bikes  pants
store 1     20     30
store 2     15      5

What items are in Store 1:
          bikes  pants  watches  glasses
store 1     20     30       35      NaN

How many bikes are in Store 2: 15
```

In [31]:

```
#Example 2. Add a column to an existing DataFrame

# We add a new column named shirts to our store_items DataFrame indicating the number of
# shirts in stock at each store. We will put 15 shirts in store 1 and 2 shirts in store 2
store_items['shirts'] = [15,2]

# We display the modified DataFrame
store_items
```

Out[31]:

| | bikes | pants | watches | glasses | shirts |
|---|---|---|---|---|---|
| **store 1** | 20 | 30 | 35 | NaN | 15 |
| **store 2** | 15 | 5 | 10 | 50.0 | 2 |

In [32]:

```
#Example 3. Add a new column based on the arithmetic operation between existing columns of a DataFrame

# We make a new column called suits by adding the number of shirts and pants
store_items['suits'] = store_items['pants'] + store_items['shirts']
```

```python
# We display the modified DataFrame
store_items
```

Out[32]:

| | bikes | pants | watches | glasses | shirts | suits |
|---|---|---|---|---|---|---|
| **store 1** | 20 | 30 | 35 | NaN | 15 | 45 |
| **store 2** | 15 | 5 | 10 | 50.0 | 2 | 7 |

In [33]:

```python
#Example 4 a. Create a row to be added to the DataFrame

# We create a dictionary from a list of Python dictionaries that will contain the number of different items at
new_items = [{'bikes': 20, 'pants': 30, 'watches': 35, 'glasses': 4}]

# We create new DataFrame with the new_items and provide and index labeled store 3
new_store = pd.DataFrame(new_items, index = ['store 3'])

# We display the items at the new store
new_store
```

Out[33]:

| | bikes | pants | watches | glasses |
|---|---|---|---|---|
| **store 3** | 20 | 30 | 35 | 4 |

In [34]:

```python
#Example 4 b. Append the row to the DataFrame

# We append store 3 to our store_items DataFrame
store_items = store_items.append(new_store)

# We display the modified DataFrame
store_items
```

```
C:\Users\Isaac\AppData\Local\Temp\ipykernel_9144\2608833744.py:4: FutureWarning: The frame.append method is dep
recated and will be removed from pandas in a future version. Use pandas.concat instead.
  store_items = store_items.append(new_store)
```

Out[34]:

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

In [35]:

```python
#Example 6. Add new column at a specific location

# We insert a new column with label shoes right before the column with numerical index 4
store_items.insert(4, 'shoes', [8,5,0])

# we display the modified DataFrame
store_items
```

Out[35]:

| | bikes | pants | watches | glasses | shoes | shirts | suits |
|---|---|---|---|---|---|---|---|
| **store 1** | 20 | 30 | 35 | NaN | 8 | 15.0 | 45.0 |
| **store 2** | 15 | 5 | 10 | 50.0 | 5 | 2.0 | 7.0 |
| **store 3** | 20 | 30 | 35 | 4.0 | 0 | NaN | NaN |

In [36]:

```python
#Example 8. Delete multiple columns from a DataFrame

# We remove the watches and shoes columns
store_items = store_items.drop(['watches', 'pants'], axis = 1)

# we display the modified DataFrame
store_items
```

Out[36]:

| | bikes | glasses | shoes | shirts | suits |
|---|---|---|---|---|---|
| **store 1** | 20 | NaN | 8 | 15.0 | 45.0 |
| **store 2** | 15 | 50.0 | 5 | 2.0 | 7.0 |
| **store 3** | 20 | 4.0 | 0 | NaN | NaN |

In [38]:

```python
#Example 9. Delete rows from a DataFrame

# We remove the store 2 and store 1 rows
store_items = store_items.drop(['store 2', 'store 1'], axis = 0)

# we display the modified DataFrame
store_items
```

Out[38]:

| | bikes | glasses | shoes | shirts | suits |
|---|---|---|---|---|---|
| **store 3** | 20 | 4.0 | 0 | NaN | NaN |

In [39]:
```
#Example 10. Modify the column label

# We change the column label bikes to hats
store_items = store_items.rename(columns = {'bikes': 'hats'})

# we display the modified DataFrame
store_items
```

Out[39]:

| | hats | glasses | shoes | shirts | suits |
|---|---|---|---|---|---|
| **store 3** | 20 | 4.0 | 0 | NaN | NaN |

In [40]:
```
#Example 11. Modify the row label

# We change the row label from store 3 to last store
store_items = store_items.rename(index = {'store 3': 'last store'})

# we display the modified DataFrame
store_items
```

Out[40]:

| | hats | glasses | shoes | shirts | suits |
|---|---|---|---|---|---|
| **last store** | 20 | 4.0 | 0 | NaN | NaN |

In [41]:
```
#Example 12. Use existing column values as row-index

# We change the row index to be the data in the pants column
store_items = store_items.set_index('hats')

# we display the modified DataFrame
store_items
```

Out[41]:

| | glasses | shoes | shirts | suits |
|---|---|---|---|---|
| **hats** | | | | |
| **20** | 4.0 | 0 | NaN | NaN |

# Dealing with NaN

As mentioned earlier, before we can begin training our learning algorithms with large datasets, we usually need to clean the data first. This means we need to have a method for detecting and correcting errors in our data. While any given dataset can have many types of bad data, such as outliers or incorrect values, the type of bad data we encounter almost always is missing values. As we saw earlier, Pandas assigns `NaN` values to missing data. In this lesson we will learn how to detect and deal with `NaN` values.

We will begin by creating a DataFrame with some `NaN` values in it.

**Example 1. Create a DataFrame**

In [42]:
```
# We create a list of Python dictionaries
items2 = [{'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}]

# We create a DataFrame  and provide the row index
store_items = pd.DataFrame(items2, index = ['store 1', 'store 2', 'store 3'])

# We display the DataFrame
store_items
```

Out[42]:

| | 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 |

We can clearly see that the DataFrame we created has 3 NaN values: one in store 1 and two in store 3. However, in cases where we load very large datasets into a DataFrame, possibly with millions of items, the number of `NaN` values is not easily visualized. For these cases, we can use a combination of methods to count the number of `NaN` values in our data. The following example combines the `.isnull()` and the `sum()` methods to count the number of NaN values in our DataFrame.

**Example 2 a. Count the total NaN values**

In [43]:
```
# We count the number of NaN values in store items
```

```
x =  store_items.isnull().sum().sum()

# We print x
print('Number of NaN values in our DataFrame:', x)
```

```
Number of NaN values in our DataFrame: 3
```

In [44]:
```
# Example 2 c. Count NaN down the column.
x =  store_items.isnull().sum()

x
```

Out[44]:
```
bikes      0
pants      0
watches    0
shirts     1
shoes      0
suits      1
glasses    1
dtype: int64
```

In [45]:
```
# We count the number of NaN values in store_items
x =  store_items.isnull()

x
```

Out[45]:

| | 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 |

In the above example, the `.isnull()` method returns a Boolean DataFrame of the same size as store_items and indicates with `True` the elements that have `NaN` values and with `False` the elements that are not. Let's see an example:

**Example 2 b. Return boolean True/False for each element if it is a NaN**

In [46]:
```
store_items.isnull()
```

Out[46]:

| | 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 |

Instead of counting the number of `NaN` values we can also do the opposite, we can count the number of `non-NaN` values. We can do this by using the `.count()` method as shown below:

**Example 3. Count the total non-NaN values**

In [48]:
```
# We print the number of non-NaN values in our DataFrame
print()
print('Number of non-NaN values in the columns of our DataFrame:\n', store_items.count())
```

```
Number of non-NaN values in the columns of our DataFrame:
 bikes      3
pants      3
watches    3
shirts     2
shoes      3
suits      2
glasses    2
dtype: int64
```

## Eliminating NaN Values

Now that we learned how to know if our dataset has any `NaN` values in it, the next step is to decide what to do with them. In general, we have two options, we can either delete or replace the NaN values. In the following examples, we will show you how to do both.

We will start by learning how to eliminate rows or columns from our DataFrame that contain any NaN values. The `.dropna(axis)` method eliminates any rows with `NaN` values when `axis = 0` is used and will eliminate any columns with NaN values when `axis = 1` is used.

> **Tip:** Remember, you learned that you can read **axis = 0** as **down** and **axis = 1** as **across** the given Numpy ndarray or Pandas dataframe

Let's see some examples.

**Example 4. Drop rows having NaN values**

In [50]:
```
# We drop any rows with NaN values
store_items.dropna(axis=0)
```

Out[50]:

| | bikes | pants | watches | shirts | shoes | suits | glasses |
|---|---|---|---|---|---|---|---|
| **store 2** | 15 | 5 | 10 | 2.0 | 5 | 7.0 | 50.0 |

**Example 5. Drop columns having NaN values**

In [52]:
```
# We drop any columns with NaN values
store_items.dropna(axis=1)
```

Out[52]:

| | bikes | pants | watches | shoes |
|---|---|---|---|---|
| **store 1** | 20 | 30 | 35 | 8 |
| **store 2** | 15 | 5 | 10 | 5 |
| **store 3** | 20 | 30 | 35 | 10 |

# Substituting NaN Values

Now, instead of eliminating `NaN` values, we can replace them with suitable values. We could choose for example to replace all `NaN` values with the value 0. We can do this by using the `.fillna()` method as shown below.

**Example 6. Replace NaN with 0**

In [54]:
```
# We replace all NaN values with 0
store_items.fillna(0)
```

Out[54]:

| | bikes | pants | watches | shirts | shoes | suits | glasses |
|---|---|---|---|---|---|---|---|
| **store 1** | 20 | 30 | 35 | 15.0 | 8 | 45.0 | 0.0 |
| **store 2** | 15 | 5 | 10 | 2.0 | 5 | 7.0 | 50.0 |
| **store 3** | 20 | 30 | 35 | 0.0 | 10 | 0.0 | 4.0 |

**Example 7. Forward fill NaN values down (axis = 0) the dataframe**

In [55]:
```
# We replace NaN values with the previous value in the column
store_items.fillna(method = 'ffill', axis = 0)
```

Out[55]:

| | 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 |

Notice that the two `NaN` values in store 3 have been replaced with previous values in their columns. However, notice that the `NaN` value in store 1 didn't get replaced. That's because there are no previous values in this column, since the `NaN` value is the first value in that column. However, if we do forward fill using the previous row values, this won't happen. Let's take a look:

**Example 8. Forward fill NaN values across (axis = 1) the dataframe**

**Notice!** Notice that the **.fillna()** method replaces (fills) the **NaN** values out of place. This means that the original DataFrame is not modified. You can always replace the **NaN** values in place by setting the keyword **inplace = True** inside the **fillna()** function.

In [56]:
```
store_items
```

Out[56]:

| | 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 |

In [57]:
```
# We replace NaN values with the next value in the column
store_items.fillna(0, inplace = True)
```

In [58]:
```
store_items
```

Out[58]:

| | bikes | pants | watches | shirts | shoes | suits | glasses |
|---|---|---|---|---|---|---|---|
| **store 1** | 20 | 30 | 35 | 15.0 | 8 | 45.0 | 0.0 |
| **store 2** | 15 | 5 | 10 | 2.0 | 5 | 7.0 | 50.0 |
| **store 3** | 20 | 30 | 35 | 0.0 | 10 | 0.0 | 4.0 |

***checking our current working directory***

In [75]:
```
pwd
```

Out[75]:
```
'C:\\Users\\Isaac'
```

**Or you can use this method to check your current working directory**

In [87]:
```
cd
```

```
C:\Users\Isaac
```

In [88]:
```
#Also you can use this method to check your current working directory
import os

os.getcwd()
```

Out[88]:
```
'C:\\Users\\Isaac'
```

**Changing the current working directory**

In [89]:
```
#Changing the current working directory

import os
os.chdir('C:\\Users\\Isaac\\Downloads')

#Now let check the new working directory
os.getcwd()
```

Out[89]:
```
'C:\\Users\\Isaac\\Downloads'
```

**Or you can use this method to change the directory**

In [91]:
```
cd C:\Users\Isaac
```

```
C:\Users\Isaac
```

In Data analysis you will most likely use databases from many sources. Pandas allows us to load databases of different formats into DataFrames. One of the most popular data formats used to store databases is csv. CSV stands for Comma Separated Values and offers a simple format to store data. We can load CSV files into Pandas DataFrames using the `pd.read_csv()` function. Let's load Google stock data into a Pandas DataFrame. The `GOOG.csv` file contains Google stock data from `8/19/2004 till 10/13/2017` taken from Yahoo Finance.

# Example 1. Load the data from a `.csv` file

In [92]:
```
# We load Google stock data in a DataFrame
Google_stock = pd.read_csv('./GOOG.csv')

# We print some information about Google_stock
print('Google_stock is of type:', type(Google_stock))
print('Google_stock has shape:', Google_stock.shape)
```

```
Google_stock is of type: <class 'pandas.core.frame.DataFrame'>
Google_stock has shape: (3313, 7)
```

In [93]:
```
Google_stock
```

Out[93]:

| | Date | Open | High | Low | Close | Adj Close | Volume |
|---|---|---|---|---|---|---|---|
| **0** | 2004-08-19 | 49.676899 | 51.693783 | 47.669952 | 49.845802 | 49.845802 | 44994500 |
| **1** | 2004-08-20 | 50.178635 | 54.187561 | 49.925285 | 53.805050 | 53.805050 | 23005800 |
| **2** | 2004-08-23 | 55.017166 | 56.373344 | 54.172661 | 54.346527 | 54.346527 | 18393200 |
| **3** | 2004-08-24 | 55.260582 | 55.439419 | 51.450363 | 52.096165 | 52.096165 | 15361800 |
| **4** | 2004-08-25 | 52.140873 | 53.651051 | 51.604362 | 52.657513 | 52.657513 | 9257400 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **3308** | 2017-10-09 | 980.000000 | 985.424988 | 976.109985 | 977.000000 | 977.000000 | 891400 |
| **3309** | 2017-10-10 | 980.000000 | 981.570007 | 966.080017 | 972.599976 | 972.599976 | 968400 |
| **3310** | 2017-10-11 | 973.719971 | 990.710022 | 972.250000 | 989.250000 | 989.250000 | 1693300 |
| **3311** | 2017-10-12 | 987.450012 | 994.119995 | 985.000000 | 987.830017 | 987.830017 | 1262400 |
| **3312** | 2017-10-13 | 992.000000 | 997.210022 | 989.000000 | 989.679993 | 989.679993 | 1157700 |

3313 rows × 7 columns

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js