The pandas library (written by Malte M. Max)

Introduction

pandas (**pan**el and **da**ta) will likely be the Python package that you will be using most for your data analytics projects. It has become the global standard package for data cleaning, manipulation, analysis, and is often used together with other packages (e.g., matplotlib for visualizations, statsmodels for statistical analyses) to facilitate this. You could think of pandas as Python's "replacement" for Excel, but then with *a lot* more functionalities.

At the heart of pandas lie its Series and DataFrame objects. A Series is a one-dimensional array (much comparable to a NumPy array) with an associated index. A DataFrame, in essence, contains tabulated data (rows and columns), and has both a row and a column index.

As outlined in the NumPy (numpy_documentation.ipynb) documentation, much of pandas is written around the functionalities of NumPy. So, much of what we have learned in NumPy is achieved in similar ways in pandas. That's good news!

Let's import pandas and abbreviate it as `pd` (this is customary). It is useful to also import NumPy alongside pandas, as you are likely to need both packages at a certain point in almost any project:

In [1]:
```
import numpy as np
import pandas as pd
```

Series

As stated above, Series are one-dimensional arrays with an associated index. Let's create a Series to understand this a little better:

In [2]:
```
a = pd.Series([3, 4, 5, 6, 7])
a
```

Out[2]:
```
0    3
1    4
2    5
3    6
4    7
dtype: int64
```

Printing the Series above shows us the values it contains (running from 3 to 7), and the index that each value is associated with (running from 0 to 4). With this index, we can extract parts of the Series:

```
In [3]: a[2]
Out[3]: 5
```

`a[2]` , as above, extracts the value that is associated with the index position of 2, which is 5 in this case. We can also extract several values at the same time:

```
In [4]: a[[2, 4]]
Out[4]: 2    5
        4    7
        dtype: int64
```

By passing a list to `a[]` (i.e., `[2, 4]`), we obtained the values associated with the index positions 2 and 4. We could, as we have seen in NumPy, also use slicing:

```
In [5]: a[1:3]
Out[5]: 1    4
        2    5
        dtype: int64
```

Note that, as in NumPy, the value associated with the index number after the colon is not included. If you wanted the values associated with the index positions 1, 2 *and* 3, you would use:

```
In [6]: a[1:4]
Out[6]: 1    4
        2    5
        3    6
        dtype: int64
```

It may be useful to label the index more intuitively:

```
In [7]: a = pd.Series([3, 4, 5, 6, 7], index = ['a', 'b', 'c', 'c', 'd'])
        a
Out[7]: a    3
        b    4
        c    5
        c    6
        d    7
        dtype: int64
```

Note that the index now contains the values that we assigned to it in the `index = ['a', 'b', 'c', 'c', 'd']` part of the code above.

Obtaining a desired part of the Series is now achieved by:

```
In [8]: a['c']
```

```
Out[8]: c    5
        c    6
        dtype: int64
```

or (by passing a list of indices):

```
In [9]: a[['c', 'd']]
```

```
Out[9]: c    5
        c    6
        d    7
        dtype: int64
```

```
In [10]: a['b':'d']
```

```
Out[10]: b    4
         c    5
         c    6
         d    7
         dtype: int64
```

As you will have noted above, when we use index labels as opposed to integer values for slicing, we obtain all values *including* the endpoint we specified after the `:`. That is, we wrote `'b':'d'` and got all values from `'b'` up to *and including* `'d'`. This is a subtle difference between using integers or labels to slice.

As we could in NumPy, we can also use Boolean-type indexing:

```
In [11]: a[a > 4]
```

```
Out[11]: c    5
         c    6
         d    7
         dtype: int64
```

We can also apply much of the NumPy functions to Series:

Note: We will focus on calculations with DataFrames (see below), so these are short examples only.

```
In [12]: a = pd.Series(range(5, 10), index = ['a', 'b', 'c', 'c', 'd'])
         a
```

```
Out[12]: a    5
         b    6
         c    7
         c    8
         d    9
         dtype: int64
```

```
In [13]: np.sum(a)
```

```
Out[13]: 35
```

```
In [14]: np.log(a)
```

```
Out[14]: a    1.609438
         b    1.791759
         c    1.945910
         c    2.079442
         d    2.197225
         dtype: float64
```

Or use simple arithmetic operators:

```
In [15]: a + 2
```

```
Out[15]: a     7
         b     8
         c     9
         c    10
         d    11
         dtype: int64
```

```
In [16]: a / 2
```

```
Out[16]: a    2.5
         b    3.0
         c    3.5
         c    4.0
         d    4.5
         dtype: float64
```

DataFrames

DataFrames contain data organized in rows and columns. Let's get right to it:

Creating DataFrames

DataFrames can be created in various ways. A common approach is using a dictionary of lists:

In [17]:
```
a = {'name' : ['Apple', 'Apple', 'IBM', 'IBM', 'Amazon', 'Amazon', '
Unilever', 'Unilever'],
     'year' : [2017, 2018, 2017, 2018, 2017, 2018, 2017, 2018],
     'net_income' : [48.351, 59.531, 5.753, 8.728, 3.033, 10.073, 6.
842, 11.088],
     'total_assets' : [375.319, 365.725, 125.356, 123.382, 131.310,
162.648, 68.140, 70.218]}
a
```

Out[17]:
```
{'name': ['Apple',
  'Apple',
  'IBM',
  'IBM',
  'Amazon',
  'Amazon',
  'Unilever',
  'Unilever'],
 'year': [2017, 2018, 2017, 2018, 2017, 2018, 2017, 2018],
 'net_income': [48.351, 59.531, 5.753, 8.728, 3.033, 10.073, 6.84
2, 11.088],
 'total_assets': [375.319,
  365.725,
  125.356,
  123.382,
  131.31,
  162.648,
  68.14,
  70.218]}
```

The dictionary `a` can then be passed to pandas's `DataFrame()` function, which will read the dictionary as a DataFrame:

In [18]:
```
b = pd.DataFrame(a, index = ['US', 'US', 'US', 'US', 'US', 'US', 'NL
', 'NL'])
b
```

Out[18]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	11.088	70.218

As you can see above, the resulting DataFrame has four columns: **name**, **year**, **net_income** and **total_assets**. (Note that **name**, **year**, **net_income** and **total_assets** correspond to the keys in dictionary `a`, and that the associated lists in that dictionary are the values in the resulting DataFrame.)

We also observe that there is a labelled index (the left-most "column"). We have created this index in the command above: `index = ['US', 'US', 'US', 'US', 'US', 'US', 'NL', 'NL']`. We will see how this can be useful later on. We will use this DataFrame for much of this introduction.

Selecting parts of the DataFrame

There are *many* ways to select parts of a DataFrame (check here (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html) for more information and details). Here are some of them:

`head()` and `tail()`

The `head()` and `tail()` functions show the first and last 5 rows of the DataFrame, respectively:

In [19]:
```
b.head()
```

Out[19]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310

In [20]:
```
b.tail()
```

Out[20]:

	name	year	net_income	total_assets
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	11.088	70.218

If you pass an integer to either of these functions, you can specify the number of rows displayed. For example, if we want to select the last 7 rows of the DataFrame:

In [21]:
```
b.tail(7)
```

Out[21]:

	name	year	net_income	total_assets
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	11.088	70.218

Selection by `[]`

You can access columns by using dictionary-like syntax:

In [22]:
```
b['name']
```

Out[22]:
```
US        Apple
US        Apple
US          IBM
US          IBM
US       Amazon
US       Amazon
NL    Unilever
NL    Unilever
Name: name, dtype: object
```

You can of course also pass a list of column names to retrieve several columns at once:

In [23]:
```
b[['name', 'year']]
```

Out[23]:

	name	year
US	Apple	2017
US	Apple	2018
US	IBM	2017
US	IBM	2018
US	Amazon	2017
US	Amazon	2018
NL	Unilever	2017
NL	Unilever	2018

You can also access rows with `[]` :

In [24]:
```
b[0:3]
```

Out[24]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356

The code above gives the first 3 rows of the DataFrame. As a reminder: `b[:3]` would have given the same result.

You could also use a negative index:

In [25]:
```
b[-1:]
```

Out[25]:

	name	year	net_income	total_assets
NL	Unilever	2018	11.088	70.218

which shows only the last row, or:

In [26]:
```
b[:-1]
```

Out[26]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140

which shows all rows except for the last row.

We could also be looking for the names and years of the companies whose net income is larger than 9:

In [27]:
```
b[b['net_income'] > 9][['year', 'name']]
```

Out[27]:

	year	name
US	2017	Apple
US	2018	Apple
US	2018	Amazon
NL	2018	Unilever

Let's examine what the command above does. It chains together two operations:

1. It selects, from all columns, all rows of the DataFrame that match the expression `b['net_income'] > 9`
2. Of the DataFrame that results from Step 1., it selects the two columns `year` and `name` (achieved by `['year', 'name']`)

Above, we have used a condensed command to achieve this, but considering the two steps involved, we can achieve the same with a more verbose approach: We could first save to a new DataFrame all of `b` that corresponds to `b['net_income'] > 9`:

In [28]:
```
c = b[b['net_income'] > 9]
c
```

Out[28]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	Amazon	2018	10.073	162.648
NL	Unilever	2018	11.088	70.218

And then, from `c`, select only the columns **year** and **name**:

In [29]:
```
c[['year', 'name']]
```

Out[29]:

	year	name
US	2017	Apple
US	2018	Apple
US	2018	Amazon
NL	2018	Unilever

Of course, for such a simple example, it does not make a large difference, but a good goal to keep in mind is to try to keep your code succinct.

`iloc[]` and `loc[]`

Other ways of selecting parts of a DataFrame are by using `loc` and `iloc`. With `loc`, we can use the index labels to select parts of a DataFrame, and with `iloc` we can use integers for positional indexing. `loc[]`'s and `iloc[]`'s syntax is basically: `loc/iloc[rows, columns]`. Before the comma, we specify the rows that we want to select, and after the comma we specify the columns that we want to select.

Let's use `loc[]` first:

```
In [30]: b.loc['US',]
```

Out[30]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648

The command above has returned all of the DataFrame that corresponds to the axis label we have provided (`US`). That is, we have retained only U.S. companies. Because we have not provided arguments after the comma, `loc[]` returns all columns. We could alternatively have used:

```
b.loc['US'] # i.e., not provide the comma
```

or

```
b.loc['US', :] # i.e., provide a comma and a colon (which indicates all
columns)
```

which would both have returned the same.

If we wanted to select only the part of the DataFrame corresponding to the net income of Dutch companies, we could use:

```
In [31]: b.loc['NL', 'net_income']
```

```
Out[31]: NL     6.842
         NL    11.088
         Name: net_income, dtype: float64
```

The `NL` before the comma specifies the rows, `net_income` after the comma specifies the columns to be considered.

If we wanted both the year and the net income for Dutch companies, we could pass a list of columns after the comma:

```
In [32]: b.loc['NL', ['year', 'net_income']]
```

Out[32]:

	year	net_income
NL	2017	6.842
NL	2018	11.088

We can also use `loc[]` to select (or, filter) rows of a DataFrame that match a Boolean expression. Suppose we are looking for all rows of the year 2017 and only the net income column. We can do this with:

```
In [33]: b.loc[b['year'] == 2017, 'net_income']
```

```
Out[33]: US    48.351
         US     5.753
         US     3.033
         NL     6.842
         Name: net_income, dtype: float64
```

With `b['year'] == 2017` (that is, before the comma), we select all rows in which the year column contains the value 2017. After the comma, we provide only `'net_income'` and `loc[]` correspondingly returns the net income column only.

We can also supply several expressions to filter the rows of the DataFrame as well as ask for several columns to be returned:

```
In [34]: b.loc[(b['year'] == 2017) & (b['net_income'] > 5), ['year', 'net_inc
         ome']]
```

Out[34]:

	year	net_income
US	2017	48.351
US	2017	5.753
NL	2017	6.842

Let's look at the code above more closely:

1. If you provide several Boolean arguments to filter the DataFrame, then each argument has to be enclosed in round brackets (i.e., `()`). You see above that `b['year'] == 2017` and `b['net_income'] > 5` are both enclosed in round brackets. The arguments are combined with a `&`, meaning that **both** expressions have to be matched (i.e., the year has to be 2017 **and** net income has to be larger than 5). You could add more arguments by adding them in round brackets and another `&`. If you want either the year to be 2017 **or** (i.e., not **and**) net income to be larger than 5, you could use the `|` operator, which indicates **or**.
2. Whenever you ask for more than one column to be returned, you need to supply a list of columns after the comma (i.e., in square brackets: `[]`). In the code above, you see that we have provided `['year', 'net_income']` after the comma, and `loc[]` returns these two columns.

If we want to use a positional argument (e.g., the second row of the DataFrame), we can use `iloc` instead of `loc`:

```
In [35]: b.iloc[1]
```

```
Out[35]: name              Apple
         year               2018
         net_income       59.531
         total_assets    365.725
         Name: US, dtype: object
```

Or, if we are interested in the third column (which is **net_income**):

```
In [36]: b.iloc[:, 2]
```

```
Out[36]: US    48.351
         US    59.531
         US     5.753
         US     8.728
         US     3.033
         US    10.073
         NL     6.842
         NL    11.088
         Name: net_income, dtype: float64
```

Note, that we have to supply a colon (`:`) to indicate that we wish to retain all rows of the DataFrame; after the comma, we then indicate which column to extract (the third here, indicated by the `2`).

Last, if we are interested in the third net income figure in `b`, we can use:

```
In [37]: b.iloc[1, 2]
```

```
Out[37]: 59.531
```

Note: the `1` corresponds to the second row, and the `2` identifies the third column.

Replacing

With `loc[]`

We can of course use the selection operators from above to change values in a DataFrame. Suppose we would like to set to 0 any net income that is larger than 40:

In [38]:
```
b.loc[b['net_income'] > 40, 'net_income'] = 0
b
```

Out[38]:

	name	year	net_income	total_assets
US	Apple	2017	0.000	375.319
US	Apple	2018	0.000	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	11.088	70.218

Let's break this down:

1. `b['net_income'] > 40` selects all rows of the DataFrame for which the net income is larger than 40
2. `'net_income'` (after the comma) selects, of the part of the DataFrame selected in Step 1., the net income column
3. `= 0` sets the selected parts of the DataFrame to 0

The result, as you can see, is that all net income figures that were previously larger than 40 (Apple in 2017 and 2018) have been set to 0.

Sorting

You can easily sort a DataFrame with `pd.sort_values()`. Let's take our initial dataset:

In [39]:
```
a = {'name' : ['Apple', 'Apple', 'IBM', 'IBM', 'Amazon', 'Amazon', '
Unilever', 'Unilever'],
     'year' : [2017, 2018, 2017, 2018, 2017, 2018, 2017, 2018],
     'net_income' : [48.351, 59.531, 5.753, 8.728, 3.033, 10.073, 6.
842, 11.088],
     'total_assets' : [375.319, 365.725, 125.356, 123.382, 131.310,
162.648, 68.140, 70.218]}
```

In [40]:
```
b = pd.DataFrame(a, index = ['US', 'US', 'US', 'US', 'US', 'US', 'NL
', 'NL'])
b
```

Out[40]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	11.088	70.218

Let's sort it by year:

In [41]:
```
b = b.sort_values(by = 'year')
b
```

Out[41]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	IBM	2017	5.753	125.356
US	Amazon	2017	3.033	131.310
NL	Unilever	2017	6.842	68.140
US	Apple	2018	59.531	365.725
US	IBM	2018	8.728	123.382
US	Amazon	2018	10.073	162.648
NL	Unilever	2018	11.088	70.218

Calculations

Let's look at calculations with DataFrames. Getting back to our sample DataFrame:

In [42]:
```
a = {'name' : ['Apple', 'Apple', 'IBM', 'IBM', 'Amazon', 'Amazon', '
Unilever', 'Unilever'],
     'year' : [2017, 2018, 2017, 2018, 2017, 2018, 2017, 2018],
     'net_income' : [48.351, 59.531, 5.753, 8.728, 3.033, 10.073, 6.
842, 11.088],
     'total_assets' : [375.319, 365.725, 125.356, 123.382, 131.310,
162.648, 68.140, 70.218]}
```

In [43]:
```
b = pd.DataFrame(a, index = ['US', 'US', 'US', 'US', 'US', 'US', 'NL
', 'NL'])
b
```

Out[43]:

	name	year	net_income	total_assets
US	Apple	2017	48.351	375.319
US	Apple	2018	59.531	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	11.088	70.218

Let's first see how many observations there are in the DataFrame using `len()`, which returns the length of the DataFrame:

In [44]:
```
len(b)
```

Out[44]: 8

Then, we can get a broad overview of the data using `describe()`, which will print several statistics for the columns in `b`:

In [45]:
```
b.describe()
```

Out[45]:

	year	net_income	total_assets
count	8.000000	8.000000	8.000000
mean	2017.500000	19.174875	177.762250
std	0.534522	21.811035	123.074511
min	2017.000000	3.033000	68.140000
25%	2017.000000	6.569750	110.091000
50%	2017.500000	9.400500	128.333000
75%	2018.000000	20.403750	213.417250
max	2018.000000	59.531000	375.319000

We could use `sum()` to obtain the sum of the **net_income** and **total_assets** columns:

```
In [46]: b[['net_income', 'total_assets']].sum()
```

```
Out[46]: net_income       153.399
         total_assets    1422.098
         dtype: float64
```

Note: We have supplied a list to `b` above, indicated by the extra `[]`: `b[['net_income', 'total_assets']]`. If we wanted the sum of only one column, we would not have needed to pass a list: `b['net_income'].sum()`.

We might be interested in a scaled net income measure. We can do this as follows:

```
In [47]: b['net_income'] / b['total_assets']
```

```
Out[47]: US    0.128826
         US    0.162775
         US    0.045893
         US    0.070740
         US    0.023098
         US    0.061931
         NL    0.100411
         NL    0.157908
         dtype: float64
```

The output, as you can see, is printed to the notebook, but has not been saved to our DataFrame. If we want to add a new column with scaled net income, we have to specify this:

```
In [48]: b['net_income_scaled'] = b['net_income'] / b['total_assets']
```

Inspecting `b` shows that the new column **net_income_scaled** is now in `b`:

```
In [49]: b
```

Out[49]:

	name	year	net_income	total_assets	net_income_scaled
US	Apple	2017	48.351	375.319	0.128826
US	Apple	2018	59.531	365.725	0.162775
US	IBM	2017	5.753	125.356	0.045893
US	IBM	2018	8.728	123.382	0.070740
US	Amazon	2017	3.033	131.310	0.023098
US	Amazon	2018	10.073	162.648	0.061931
NL	Unilever	2017	6.842	68.140	0.100411
NL	Unilever	2018	11.088	70.218	0.157908

In the same spirit, we can add (subtract, multiply, ...) columns:

```
In [50]: b['net_income'] + b['total_assets']
```

```
Out[50]: US    423.670
         US    425.256
         US    131.109
         US    132.110
         US    134.343
         US    172.721
         NL     74.982
         NL     81.306
         dtype: float64
```

Note: Adding net income and total assets is probably not a very useful measure - we do this just for demonstration purposes.

We can also perform calculations on subsets of a DataFrame:

```
In [51]: b.loc[b['name'] == 'Apple', 'net_income'].max()
```

```
Out[51]: 59.531
```

The code above first selects all rows of Apple and only the net income column, and on that selection of the DataFrame we then apply the `max()` function.

In the same spirit, we could get descriptive statistics of our variables for a particular subset of the data. Suppose we are interested in the descriptive statistics for the year 2018:

```
In [52]: b.loc[b['year'] == 2018,].describe()
```

Out[52]:

	year	net_income	total_assets	net_income_scaled
count	4.0	4.000000	4.000000	4.000000
mean	2018.0	22.355000	180.493250	0.113339
std	0.0	24.802842	129.166017	0.054430
min	2018.0	8.728000	70.218000	0.061931
25%	2018.0	9.736750	110.091000	0.068538
50%	2018.0	10.580500	143.015000	0.114324
75%	2018.0	23.198750	213.417250	0.159125
max	2018.0	59.531000	365.725000	0.162775

Calculations using `groupby()`

Suppose we are interested in the average net income *of each company* contained in our DataFrame. We could get these values with `groupby()`:

In [53]:
```
b['net_income'].groupby(b['name']).mean()
```

Out[53]:
```
name
Amazon       6.5530
Apple       53.9410
IBM          7.2405
Unilever     8.9650
Name: net_income, dtype: float64
```

Take a moment to see how this command works:

1. We specify the column for which we want to calculate a measure (the mean, in this case): `b['net_income']`
2. We specify along which column we want to group: `.groupby(b['name'])`
3. We then specify what we want to extract from this grouped data object: `.mean()`

Also note, an equivalent for the command above is:

In [54]:
```
b.groupby('name')['net_income'].mean()
```

Out[54]:
```
name
Amazon       6.5530
Apple       53.9410
IBM          7.2405
Unilever     8.9650
Name: net_income, dtype: float64
```

Here, we call `groupby()` on all of `b`, and then specify after the `groupby()` for which column (`['net_income']`) we want to calculate a certain statistic (`.mean()`).

We could also imagine that we want the maximum of net income and total assets for each year, and that we want to save these values to a new DataFrame. We could achieve this by:

In [55]:
```
c = b[['net_income', 'total_assets']].groupby(b['year']).max()
c
```

Out[55]:

year	net_income	total_assets
2017	48.351	375.319
2018	59.531	365.725

Again, what does the command do step-by-step?

1. We start by specifying the columns for which we want to calculate a measure (the maximum, in this case): `b[['net_income', 'total_assets']]`
2. We specify along which column we want to group: `.groupby(b['year'])`
3. We then specify what we want to extract from this grouped data object: `.max()`
4. Last, we save all this to a new object: `c =` (*Note*: this is the first step in the calculation, but listed as the last here to make the understanding easier)

Let's take this one step further.

Suppose we want to include a column which contains the difference between (i) a firm's net income in a given year and (ii) the maximum net income generated by any firm contained in the dataset. A command we could use to achieve that is:

In [56]:
```
b['deviation_from_max'] = b['net_income'] - b['net_income'].groupby
(b['year']).transform('max')
b
```

Out[56]:

	name	year	net_income	total_assets	net_income_scaled	deviation_from_max
US	Apple	2017	48.351	375.319	0.128826	0.000
US	Apple	2018	59.531	365.725	0.162775	0.000
US	IBM	2017	5.753	125.356	0.045893	-42.598
US	IBM	2018	8.728	123.382	0.070740	-50.803
US	Amazon	2017	3.033	131.310	0.023098	-45.318
US	Amazon	2018	10.073	162.648	0.061931	-49.458
NL	Unilever	2017	6.842	68.140	0.100411	-41.509
NL	Unilever	2018	11.088	70.218	0.157908	-48.443

As you can see, we now have an extra column that contains zeros for Apple in 2017 and 2018 (because Apple generated the highest net income in 2017 and 2018). For all other companies, we see the difference of their own net income from Apple's net income in either 2017 or 2018. Using such commands can be useful when trying to calculate differences from industry means, etc.

But, how does the command work?

1. We take the column from which we want to subtract something: `b['net_income']`
2. We then specify that we want to group the **net income** column by the **year** column: `b['net_income'].groupby(b['year'])`
3. Last, we use `transform('max')` to calculate the maximum per year
   A. We have used `'max'` here, but we could have used `'min'`, `'mean'`, `'median'`, etc.
   B. `transform(max)` ensures that the `max` function will be applied to the grouped DataFrame and that the output aligns with the length of the original DataFrame. More information on this can be found here (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transform.html).

Another way in which `groupby()` can be handy is in the calculation of year-on-year differences. We can use `diff()` for this. Now, suppose we want the increase in total assets from the last year to the current year. If we did not have `groupby()` at our disposal, we might do:

In [57]:
```
b['total_assets_diff'] = b['total_assets'].diff()
```

In [58]:
```
b
```

Out[58]:

	name	year	net_income	total_assets	net_income_scaled	deviation_from_max	total
US	Apple	2017	48.351	375.319	0.128826	0.000	
US	Apple	2018	59.531	365.725	0.162775	0.000	
US	IBM	2017	5.753	125.356	0.045893	-42.598	
US	IBM	2018	8.728	123.382	0.070740	-50.803	
US	Amazon	2017	3.033	131.310	0.023098	-45.318	
US	Amazon	2018	10.073	162.648	0.061931	-49.458	
NL	Unilever	2017	6.842	68.140	0.100411	-41.509	
NL	Unilever	2018	11.088	70.218	0.157908	-48.443	

As you can see above, IBM has a decrease in total assets of 240.369 in 2017, which is clearly wrong. The problem here is that pandas used Apple's 2018 value and subtracted it from IBM's 2017 value. Obviously, this is not what we want. We can solve this using `groupby()` (in combination with `transform()` and a so-called lambda function):

In [59]:
```
b['total_assets_diff'] = b['total_assets'].groupby(b['name']).transform(lambda x: x.diff())
```

In [60]:
```
b
```

Out[60]:

	name	year	net_income	total_assets	net_income_scaled	deviation_from_max	total
US	Apple	2017	48.351	375.319	0.128826	0.000	
US	Apple	2018	59.531	365.725	0.162775	0.000	
US	IBM	2017	5.753	125.356	0.045893	-42.598	
US	IBM	2018	8.728	123.382	0.070740	-50.803	
US	Amazon	2017	3.033	131.310	0.023098	-45.318	
US	Amazon	2018	10.073	162.648	0.061931	-49.458	
NL	Unilever	2017	6.842	68.140	0.100411	-41.509	
NL	Unilever	2018	11.088	70.218	0.157908	-48.443	

Now we see that pandas has produced `NaN` values (Not a Number - more on this below) for each company in the year 2017 (because 2016 figures are not available for the companies), and has created correct values for 2018.

Missing data in pandas

Until now, our datasets did not contain any missing values. In real datasets, however, this is often the case. pandas displays missing values as `NaN` (Not a Number).

Suppose we have this dataset (note that we create missing values using NumPy's `nan` function):

In [61]:
```
a = {'name' : ['Apple', 'Apple', 'Apple', 'IBM', 'IBM', 'Amazon', 'A
mazon', 'Unilever', 'Unilever', 'Unilever'],
     'year' : [2016, 2017, 2018, 2017, 2018, 2017, 2018, 2016, 2017,
2018],
     'net_income' : [45.687, 48.351, np.nan, 5.753, 8.728, 3.033, 1
0.073, 5.737, 6.842, np.nan],
     'total_assets' : [321.686, np.nan, 365.725, 125.356, 123.382, 1
31.310, 162.648, 62.444, 68.140, 70.218]}
```

In [62]:
```
b = pd.DataFrame(a, index = ['US', 'US', 'US', 'US', 'US', 'US', 'US
', 'NL', 'NL', 'NL'])
b
```

Out[62]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	Apple	2017	48.351	NaN
US	Apple	2018	NaN	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	NaN	70.218

Identifying and filtering missing data

We can identify the cells with missing data in the **net_income** column with:

In [63]:
```
b[b['net_income'].isnull() == True]
```

Out[63]:

	name	year	net_income	total_assets
US	Apple	2018	NaN	365.725
NL	Unilever	2018	NaN	70.218

Above, we haved passed the Boolean argument `b['net_income'].isnull() == True` (i.e., all rows of the colum **net_income** for which the function `isnull()` evaluates to `True`) to `b`. Note that we do not need to explicitly specify the `== True`:

In [64]:
```
b[b['net_income'].isnull()]
```

Out[64]:

	name	year	net_income	total_assets
US	Apple	2018	NaN	365.725
NL	Unilever	2018	NaN	70.218

Also note that next to the `isnull()` function, pandas also knows the `isna()` function, which does the exact same thing:

In [65]:
```
b[b['net_income'].isna()]
```

Out[65]:

	name	year	net_income	total_assets
US	Apple	2018	NaN	365.725
NL	Unilever	2018	NaN	70.218

You can achieve the opposite of `isnull()` with `notnull()` (and the opposite of `isna()` with `notna()`):

In [66]:
```
b[b['net_income'].notnull()]
```

Out[66]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	Apple	2017	48.351	NaN
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140

In [67]:
```
b[b['net_income'].notna()]
```

Out[67]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	Apple	2017	48.351	NaN
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140

Note: There are reasons for why both `isnull()` / `notnull()` *and* `isna()` / `notna()` *exist even though they achieve the same outcome. For now, just remember that both do the same.*

If we want to filter out missing values in two columns, we could use:

```
In [68]: b[(b['net_income'].notnull()) & (b['total_assets'].notnull())]
```

Out[68]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140

Take a moment to see above that we have passed two conditionals:

1. `b['net_income'].notnull()`, surrounded by `()`
2. `b['total_assets'].notnull()`, also surrounded by `()`
3. The two conditional statements both have to be `True` because we specified the `&`

Alternatively, you can use `loc[]` to filter in the same way:

```
In [69]: b.loc[(b['net_income'].notnull()) & (b['total_assets'].notnull()),]
```

Out[69]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140

Given `loc[]`'s syntax (i.e., `loc[rows, columns]`), note that we have added a comma. As we did not specify a specific column to be selected (i.e., we did not provide anything after the comma), `loc[]` returns all columns.

Another way to achieve the above is with `dropna()`, where we specify the columns in which pandas should look for `NaN` using the `dropna()`'s `subset` argument:

In [70]:
```
b.dropna(subset = ['net_income', 'total_assets'])
```

Out[70]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140

Replacing missing data

There may be instances in which you wish to replace `NaN` by a specific value. Using `fillna()`, we can replace missing values by zeros:

In [71]:
```
b.fillna(0)
```

Out[71]:

	name	year	net_income	total_assets
US	Apple	2016	45.687	321.686
US	Apple	2017	48.351	0.000
US	Apple	2018	0.000	365.725
US	IBM	2017	5.753	125.356
US	IBM	2018	8.728	123.382
US	Amazon	2017	3.033	131.310
US	Amazon	2018	10.073	162.648
NL	Unilever	2016	5.737	62.444
NL	Unilever	2017	6.842	68.140
NL	Unilever	2018	0.000	70.218

Or, we could wish to replace the missing **net_income** values by the means of the available **net_income** values *per company*:

In [72]:
```
b['net_income'] = b['net_income'].fillna(b['net_income'].groupby(b['
name']).transform('mean'))
b
```

Out[72]:

	name	year	net_income	total_assets
US	Apple	2016	45.6870	321.686
US	Apple	2017	48.3510	NaN
US	Apple	2018	47.0190	365.725
US	IBM	2017	5.7530	125.356
US	IBM	2018	8.7280	123.382
US	Amazon	2017	3.0330	131.310
US	Amazon	2018	10.0730	162.648
NL	Unilever	2016	5.7370	62.444
NL	Unilever	2017	6.8420	68.140
NL	Unilever	2018	6.2895	70.218

Let's unpack the code above:

1. We specify the column in which we want to replace missing values: `b['net_income'].fillna`
2. We specify that we want to replace these values by values from the **net income** column: `(b['net_income']`
3. And we specifically look for the means of available values *per company*: `.groupby(b['name']).transform('mean'))`

Loading (saving) data from (to) disk

Until now, we have created DataFrames ourselves. In practice, however, you are likely to be using datasets which are stored on your disk, for example datasets you have downloaded from Compustat. Let's see how we can import data for use in pandas.

In this tutorial, we use the file **compustat.xlsx**, which is a download from Compustat that includes **total assets**, **net income (loss)** and **sales**.

Note 1: If you are going to try this with a dataset of your own, you will have to provide the specific path to the dataset that you're trying to load. In the case below, the path is `data_files/compustat.xlsx` , meaning the file is located in a sub-directory (`data_files/`), and that sub-directory is in the same directory as the Jupyter Notebook that is used to write this tutorial.

Note 2: Here (https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html) is a list of pandas methods to import different file types (e.g., .csv, .xls(x), .dta (Stata)).

In [73]:
```
data = pd.read_excel('data_files/compustat.xlsx')
```

In [74]:
```
data.head()
```

Out[74]:

	Global Company Key	Data Date	Data Year - Fiscal	Company Name	Assets - Total	Net Income (Loss)	Sales/Turnover (Net)
0	1004	20180531	2017	AAR CORP	1524.700	15.600	1748.300
1	1045	20181231	2018	AMERICAN AIRLINES GROUP INC	60580.000	1412.000	44541.000
2	1050	20181231	2018	CECO ENVIRONMENTAL CORP	392.582	-7.121	337.339
3	1062	20181130	2018	ASA GOLD AND PRECIOUS METALS	196.072	-48.789	1.629
4	1072	20180331	2017	AVX CORP	2672.766	4.910	1562.474

Let's add a new column, net income scaled by sales:

In [75]:
```
data['net_income_scaled'] = data['Net Income (Loss)'] / data['Sales/Turnover (Net)']
data.head()
```

Out[75]:

	Global Company Key	Data Date	Data Year - Fiscal	Company Name	Assets - Total	Net Income (Loss)	Sales/Turnover (Net)	net
0	1004	20180531	2017	AAR CORP	1524.700	15.600	1748.300	
1	1045	20181231	2018	AMERICAN AIRLINES GROUP INC	60580.000	1412.000	44541.000	
2	1050	20181231	2018	CECO ENVIRONMENTAL CORP	392.582	-7.121	337.339	
3	1062	20181130	2018	ASA GOLD AND PRECIOUS METALS	196.072	-48.789	1.629	
4	1072	20180331	2017	AVX CORP	2672.766	4.910	1562.474	

And save the DataFrame to a new file called **compustat_new.xlsx**:

In [76]:
```
data.to_excel('data_files/compustat_new.xlsx')
```

If you check the directory to which the file is saved, you should see that it is available there now.

Often, data does not come in Excel files but in text files in which values are separated by delimiters such as commas, semicolons, tabs, etc. `pd.read_csv()` can be used to load such files into memory. Let's use a tab-delimited text file with the same content as the .xlsx file we loaded before (this file was also downloaded from Compustat):

In [77]:
```
data = pd.read_csv('data_files/compustat.txt')
```

In [78]:
```
data.head()
```

Out[78]:

	gvkey\tdatadate\tfyear\tconm\tat\tni\tsale
0	001004\t20180531\t2017\tAAR CORP\t1524.7000\t1...
1	001045\t20181231\t2018\tAMERICAN AIRLINES GROU...
2	001050\t20181231\t2018\tCECO ENVIRONMENTAL COR...
3	001062\t20181130\t2018\tASA GOLD AND PRECIOUS ...
4	001072\t20180331\t2017\tAVX CORP\t2672.7660\t4...

As you can see above, the file has been loaded, but the format is not ready for use. If you check the official documentation (https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-read-csv-table) for `pd.read_csv()`, you will see that it has an argument `delimiter`, with which we can specify the delimiter pandas should use when reading the file. Let's set it to `'\t'` for tab-delimited:

In [79]:
```
data = pd.read_csv('data_files/compustat.txt', delimiter = '\t')
```

In [80]:
```
data.head()
```

Out[80]:

	gvkey	datadate	fyear	conm	at	ni	sale
0	1004	20180531	2017	AAR CORP	1524.700	15.600	1748.300
1	1045	20181231	2018	AMERICAN AIRLINES GROUP INC	60580.000	1412.000	44541.000
2	1050	20181231	2018	CECO ENVIRONMENTAL CORP	392.582	-7.121	337.339
3	1062	20181130	2018	ASA GOLD AND PRECIOUS METALS	196.072	-48.789	1.629
4	1072	20180331	2017	AVX CORP	2672.766	4.910	1562.474

Now, the file has been loaded correctly. In case we want to save this DataFrame, say in comma-separated format, we can use:

In [81]:
```
data.to_csv('data_files/compustat_new.txt', sep = ',')
```

As you can see, the argument to be used to specify the delimiter here is `sep =`, to which we supplied `','` to indicate separation by comma.