Session 5 - Dataframes

May 27, 2019

Table of Contents

1 Pandas Dataframes

Python is very good for data analysis. Much of this is thanks to the `pandas` library, which contains a wealth of powerful functions to load data and manipulate it.

In the `pandas` environment what we normally refer to as a table is called a ***DataFrame***. Is the data has only a single column, it is called a ***Series***. These are the core objects in the library.

As with many libraries, there is a convection for renaming when importing. In `pandas` the convention is to import as `pd`:

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

1.1 Creating a simple DataFrame

A simple DataFrame can be created as with `pandas.DataFrame()`:

```
In [2]: # Create a simple DataFrame
        df = pd.DataFrame({'Column1': [11, 12, 13],
                           'Column2': [21, 22, 23],
                           'Column3': [31, 32, 33]})
        df
```

```
Out[2]:    Column1  Column2  Column3
        0       11       21       31
        1       12       22       32
        2       13       23       33
```

- **Note 1:** The input argument for creating the DataFrame is a *dictionary*. I.e. a data structure with keys-value pairs.
- **Note 2:** It automatically creates and *index* column as the leftmost column.
- **Note 3:** The displayed DataFrame looks nicer than the it would have in an editor. This is because it is styled with HTML. In an editor, the printed DataFrame would look like this:

```
In [3]: # DataFrame as it would look without HTML-styling
        print(df)
```

```
   Column1  Column2  Column3
0       11       21       31
1       12       22       32
2       13       23       33
```

1.2 Presenting results with DataFrames

If we have a dictionary from a previous calculation of some kind, we can quickly turn it into a DataFrame with the same pricinple as above:

```
In [4]: # Define normal force and cross sectional area
        normal_force = [85, 56, 120]
        area = [314, 314, 314]

        # Compute stress in cross section for all normal forces
        stress = [n/a for n, a in zip(normal_force, area)]

        # Gather calculation results in dictionary
```

```
results = {'N [kN]': normal_force, 'A [mm2]': area, 'sigma_n [MPa]': stress}

# Create a DataFrame of the results form the dictionary
df2 = pd.DataFrame(results)
df2
```

Out[4]:

	N [kN]	A [mm2]	sigma_n [MPa]
0	85	314	0.270701
1	56	314	0.178344
2	120	314	0.382166

1.2.1 Adjusting the index column

The default index (leftmost column) is not really suited for this particular scenario, so we could change it to be "Load Case" and have it start at 1 instead of 0

In [5]:
```
# Set the name of the index to "Load Case"
df2.index.name = 'Load Case'

# Add 1 to all indices
df2.index += 1

df2
```

Out[5]:

Load Case	N [kN]	A [mm2]	sigma_n [MPa]
1	85	314	0.270701
2	56	314	0.178344
3	120	314	0.382166

1.3 Extracting a subset of data

We can extract specific columns from the DataFrame:

In [6]:
```
# Extract only the stress column to new DataFrame
df2[['sigma_n [MPa]']]
```

Out[6]:

Load Case	sigma_n [MPa]
1	0.270701
2	0.178344
3	0.382166

- **Note:** The use of two square bracket pairs `[[]]` turns the result into a new DataFrame, with just one column. If there had been only a single square bracket, the result would be a *Series* object. See below.

In [7]:
```
# Extract stress column to Series object
df2['sigma_n [MPa]']
```

```
Out[7]: Load Case
        1    0.270701
        2    0.178344
        3    0.382166
        Name: sigma_n [MPa], dtype: float64
```

Most of the time, we want to keep working with DataFrames, so remember to put double square brackets.

Double square brakcets must be used if we want to extract more than one column. Otherwise, a `KeyError` will be raised.

```
In [8]: # Extract multiple columns to DataFrame
        df2[['N [kN]', 'sigma_n [MPa]']]
```

```
Out[8]:            N [kN]  sigma_n [MPa]
        Load Case
        1              85       0.270701
        2              56       0.178344
        3             120       0.382166
```

1.4 Importing data from file to DataFrame

Data can be imported from various file types. The most common ones are probably standard text files (`.txt`), comma separated value files (`.csv`) or Excel files (`.xlsx`)

Some common scenarios

```
# Import from .csv (comma separated values)
pd.read_csv('<file_name>.csv')

# Import from .txt  with values separated by white space
pd.read_csv('<file_name>.txt', delim_whitespace=True)

# Import from Excel
pd.read_excel('<file_name>.xlsx', sheet_name='<sheet_name>')
```

The above assumes that the files to import are located in the same directory as the script. Placing it there makes it easier to do the imports.

The functions above have many optional arguments. When importing from an Excel workbook it will often be necessary to specify more parameters than when importing a plain text file, because the Excel fil is a lot more complex. For example, by default the data starts at cell A1 as the top left and the default sheet is the first sheet occuring in the workbook, but this is not always what is wanted.

See docs for both functions here:

- `panda.read_csv()`: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
- `panda.read_excel()`: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

1.5 Import example

```
In [9]: # Import data from 'HEA.txt', which has data separated
        # by white spaces
        df = pd.read_csv('HEA.txt', delim_whitespace=True)
        df
```

Out[9]:

	Profile	h[mm]	b[mm]	Iy[mm4]	Wel,y[mm3]	g[kg/m]
0	HE100A	96	100	3490000	72.8	16.7
1	HE120A	114	120	6060000	106.0	19.9
2	HE140A	133	140	10300000	155.0	24.7
3	HE160A	152	160	16700000	220.0	30.4
4	HE180A	171	180	25100000	294.0	35.5
5	HE200A	190	200	36900000	389.0	42.3
6	HE220A	210	220	54100000	515.0	50.5
7	HE240A	230	240	77600000	675.0	60.3
8	HE260A	250	260	104500000	836.0	68.2
9	HE280A	270	280	136700000	1010.0	76.4
10	HE300A	290	300	182600000	1260.0	88.3

1.6 Filtering data

Data filtering is easy and intuitive. It is done by conditional expressions.

For example, if we want to filter the HEA-DataFrame for profiles with moment of inertia I_y larger than some value:

```
In [10]: df[df['Iy[mm4]'] > 30000000]
```

Out[10]:

	Profile	h[mm]	b[mm]	Iy[mm4]	Wel,y[mm3]	g[kg/m]
5	HE200A	190	200	36900000	389.0	42.3
6	HE220A	210	220	54100000	515.0	50.5
7	HE240A	230	240	77600000	675.0	60.3
8	HE260A	250	260	104500000	836.0	68.2
9	HE280A	270	280	136700000	1010.0	76.4
10	HE300A	290	300	182600000	1260.0	88.3

1.6.1 Understanding the filtering process

The inner expression of the filtering

```
df['Iy[mm4]'] > 30000000
```

returns the column `Iy[mm4]` from the DataFrame converted into a ***boolean Series***. I.e. a Series with `True/False` in each row depending on the condition being fulfilled or not. See the printout below.

```
In [11]: # Inner expression returns a boolean Series of Iy[mm4]
         df['Iy[mm4]'] > 30000000
```

```
Out[11]: 0     False
         1     False
         2     False
         3     False
         4     False
         5      True
         6      True
         7      True
         8      True
         9      True
         10     True
         Name: Iy[mm4], dtype: bool
```

This boolean Series is used to filter the original DataFrame, which is done in the outer expression by `df[boolean_series]`.

The outer expression picks only the rows from the orignal DataFrame where the boolean series is `True`.

1.7 Filtering by multiple conditions

Filtering based on multiple conditions can be quite powerful. The syntax is only slightly more complicated

```
In [12]: df[(df['Iy[mm4]'] > 30000000) & (df['h[mm]'] < 260 )]
```

```
Out[12]:   Profile  h[mm]  b[mm]    Iy[mm4]  Wel,y[mm3]  g[kg/m]
         5  HE200A    190    200   36900000       389.0     42.3
         6  HE220A    210    220   54100000       515.0     50.5
         7  HE240A    230    240   77600000       675.0     60.3
         8  HE260A    250    260  104500000       836.0     68.2
```

Filtering can also be based on lists of values:

```
In [13]: # Valid profiles to choose from
         valid_profiles = ['HE180A', 'HE220A', 'HE260A', 'HE280A']

         # Filter DataFrame based in Iy and valid profiles
         df[(df['Iy[mm4]'] > 30000000) & (df['Profile'].isin(valid_profiles) )]
```

```
Out[13]:   Profile  h[mm]  b[mm]    Iy[mm4]  Wel,y[mm3]  g[kg/m]
         6  HE220A    210    220   54100000       515.0     50.5
         8  HE260A    250    260  104500000       836.0     68.2
         9  HE280A    270    280  136700000      1010.0     76.4
```

If we want to rule out some profiles, we could put a ~ in front of the condition to specify that values must ***not*** be present in the list:

```
In [14]: # Invalid profiles
         invalid_profiles = ['HE180A', 'HE220A', 'HE260A', 'HE280A']

         # Filter DataFrame based in Iy and valid profiles
         df[(df['Iy[mm4]'] > 30000000) & (~df['Profile'].isin(invalid_profiles) )]
```

```
Out[14]:    Profile  h[mm]  b[mm]    Iy[mm4]  Wel,y[mm3]  g[kg/m]
         5   HE200A    190    200   36900000       389.0     42.3
         7   HE240A    230    240   77600000       675.0     60.3
         10  HE300A    290    300  182600000      1260.0     88.3
```

1.8 Exporting a DataFrame to a file

Exporting a DataFrame to a new text file could not be easier. Saving to a `.txt`:

1.9 GroupBy

`groupby` provides a way to ***split*** a DataFrame into groups based on some condition, ***apply*** a function to those groups and ***combine*** the results into a new DataFrame that is returned.

1.9.1 An example

```
In [15]: # Create a dataframe to work with
         dff = pd.DataFrame({'Fruit': ['Pear', 'Apple', 'Apple', 'Banana',
                                       'Lemon', 'Banana', 'Banana', 'Pear'],
                             'Amount_sold':  [3, 6, 7, 2, 4, 7, 1, 6]})
         dff
```

```
Out[15]:    Fruit  Amount_sold
         0    Pear            3
         1   Apple            6
         2   Apple            7
         3  Banana            2
         4   Lemon            4
         5  Banana            7
         6  Banana            1
         7    Pear            6
```

The `DataFrame.groupby` method itself returns a *groupby object*, ***not*** a DataFrame. So printing that on its own will just show you the object.

```
In [71]: # The gropuby will return a groupby object
         dff.groupby('Fruit')
```

```
Out[71]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001B716413978>
```

The object contains metadata about how the data is grouped. The powerful operations are visible only after we apply a certain function to the groupby object, like `sum()`:

```
In [17]: dff.groupby('Fruit').sum()
```

```
Out[17]:         Amount_sold
         Fruit
         Apple            13
         Banana           10
         Lemon             4
         Pear              9
```

We could say that we first ***split*** the DataFrame in fruit groups, ***applied*** a function to those individual groups and ***combined*** and returned the results.

Note that by default the column that was grouped by becomes the new index, since these are now unique values.

1.9.2 Documentation

Documentation for `groupby`: http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html Documentation for `apply`: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.apply.html

1.10 Printing with `df.head()` and `df.tail()`

When DataFrames become very large, printing all the data to the screen becomes unwieldy. Printing is mostly done only to make sure that some operation worked as we expected it would. In that case, printing just a few rows will be sufficient, which the following methods will alloww for:

```
# Print the first 5 rows of df
df.head()

# Print the last 5 rows of df
df.tail()

# Print the first x rows of df
df.head(x)

# Print the last y rows of df
df.tail(y)
```

1.11 Something to be aware of

A potentially confusing thing about `pandas` methods is that it can be hard to know which mutates the DataFrame inplace and which needs to be saved to a new varaible. Consider the lines below:

You will most likely stumble across this when working with `pandas`. ***Note that there is no error when when executing the first line shown above, but when `df` is eventually printed it will just not be as intended.***

1.12 Much more functionality

There are numerous functions and methods available in `pandas` and the above mentioned barely scrathes the surface.

Practically anything that you would want to do to a dataset can be done. And quite possibly somebody has had the same problem as you before and found a solution or maybe even even contributed to the `pandas` library and put in that functionality for everyone to use. However, some functionality can be much harder to understand and use than the above mentioned.

The `pandas` library integrates well with other big libraries like `numpy` and `matplotlib` and other functionality in the Python language in general. For example, many DataFrame methods can take as input a customly defined function `def ...()` and run it through certian content of the DataFrame.

Plotting with `matplotlib` is directly supported in `pandas` via shortcuts so you can do `df.plot()` and it will create a plot of the DataFrame of a specified kind even without having to import `matplotlib`.

1.13 When and why to use `pandas`

- The manipulations that can be done with `pandas` are quite powerful when datasets become much larger than ones shown above. It is especially helpful when the dataset reaches a size where all data can not be viewed and understood well by simply scrolling down and looking at the data. If the number of rows go beyond just a couple of thousands, it is hard to get the overall feel for the data and its trends just by inspection. This is were typing logic commands to do manipulations becomes a great help.
- Use it when a very specific solution for data manipulation is desired. Especially when the solution is not trivially done in for example Excel.
- It is a good tool for combining multiple datasets, e.g. from different files.
- Last but not least, it is good for reproducibility and handling changes in data size.

2 Exercise 1.1

All exercises 1.x are working with the same DataFrame.

Create a DataFrame from the dictionary `d` below. Save it is a variable called `df`.
Print/display the entire DataFrame to see it if it comes out as you expect.
Remember to `import pandas as pd`.

3 Exercise 1.2

Print/display the only the first or last rows by using `DataFrame.head()` or `DataFrame.tail()`. You choose how many rows to print (default is 5).

Use these methods to test print the DataFrames from now on to avoid printing all rows.

4 Exercse 1.3

Filter `df` to only contain the rows where the uppercase letter is `'K'`. Save it to a new variable called `dfk`.

Print/display it to make sure it it correct.

If you were unlucky and did not have a `'K'` generated in the uppercase column, try re-running the code.

5 Exercise 1.4

When printing the filtered `dfk`, notice that the index from the original DataFrame is kept. This is often useful for back reference, but sometimes we want the index to be reset.

Reset the index of `dfk` to start from 0 by using `DataFrame.reset_index()`. This method does not modify the DataFrame inplace by default, so remember to either save to a new variable or give the input argument `inplace=True`.

By default, the orignal index will be added as a new column to the DataFrame. If you don't want this, use the input argument `drop=True`.

6 Exercise 2.1

All exercises 2.x are to be seen as the same problem. It has just been divided into smaller tasks.

```
# Remove column 'column_name' form 'df'
df = df.drop('column_name', axis=1)

# Plot dataframe contents
df.plot(kind='bar', x='column_for_x_values', y='column_for_y_values')
```

The method has many optional arguments, see https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html.

Try for example to change the figure size by `figsize=(width, height)`, rotate the x-ticks by `rot=angle_in_degrees` and change the color of the bars by `color='some_color'`.

7 If you are up for more

Create a loop that goes through all shear keys, creates a plot like the one from the previous exercise and saves each plot to a png-file.

```
In [ ]:
```