# 5. Practical week 4, part A (More about Data Frames)

## 5.1. Changing and manipulating **pandas'** data frame objects

In this practical we will cover some basic operations with data frames:

- How to add new (computed) columns in a data frame
- How to sort data frame tables by column values
- How to do simple statistics on data frame columns
- How to make simple visualizations of the data
- How to generate test (fake) data

## 5.2. Why knowing **pandas** is important



Increasingly, packages are being built on top of `pandas` to address specific needs in data preparation, analysis and visualization. This is encouraging because it means `pandas` is not only helping users to handle their data tasks but also that it provides a better starting point for developers to build powerful and more focused data tools. The creation of libraries that complement `pandas`' functionality also allows `pandas` development to remain focused around its original requirements.



For example `Statsmodels` is the prominent Python "statistics and econometrics library" and it has a long-standing special relationship with `pandas`. `Statsmodels` provides powerful statistics, econometrics, analysis and modeling functionality that is out of `pandas`' scope. `Statsmodels` leverages `pandas` objects as the underlying data container for computation.



Even if not built on top of pandas, other libraries are part of the "ecosystem" of panda. For example `Seaborn` is a Python visualization library based on the library `matplotlib`. It provides a high-level, dataset-oriented interface for creating attractive statistical graphics. The plotting functions in `Seaborn` understand `pandas` objects and leverage `pandas` grouping operations internally to support concise specification of complex visualizations. `Seaborn` also goes beyond `matplotlib` and `pandas` with the option to perform statistical estimation while plotting, aggregating across observations and visualizing the fit of statistical models to emphasize patterns in a dataset.

## 5.3. To start with a simple example

First, we consider a csv file that contains the restaurant list we used before (download it from Nestor, '`restRanking.csv`'), structured only on two columns (restaurant name, and address), using this time the '`;`' (semicolon) as a separator. (NOTE: csv's are many times called "comma separated value files", but in fact, the abbreviation stands for "CHARACTERS separated value

files", meaning that almost any character or set of characters can be used as separators – of course, commas and semicolons are make the textual content of the files easier to read with the human eye if necessary).

EXERCISE 1: (write the code in a .py file named for example practical_week4_ex_1):

Read the content of this file in a pandas data frame object (you should know now how to do it). The `read_csv()` method has multiple parameters, and you need now the `sep` (indicating which separator character is used in the file), which has the default value ',' (comma). As you would expect, this parameter has to be set for this file on the semicolon character. Display in the console the `shape`, `columns`, and `dtypes` properties of the newly created data frame. Display the whole data frame (all rows).

We want now to add some information to this data frame. For example, two columns, one representing the number of positive reviews by customers, and another one, representing the negative reviews, this giving us enough information to compute a 0-5 ranking of each restaurant. We consider here that customers respond to a review request, and they can answer in three ways: "good", "bad", "not sure". That means that we need a third column for this kind of responses.

Initially, we do not know these results, and we need a placeholder – like the number 0.



A solution would be to generate a list full of zeroes as long as the table's length in rows (you have to complete the name you gave to your data frame object, without the <> characters below):



```
zeroes = [0] * <the name you gave to your object>.shape[0]
```

…and we simply use the same technique as used in a dictionary Python type. Just assign the zero values in the list to the data frame, indexed with a new column name.



```
<the name you gave to your object> ['nrGoodReviews'] = zeroes
<the name you gave to your object> ['nrBadReviews'] = zeroes
<the name you gave to your object> ['nrUndecided'] = zeroes
```

Try this, and print the data frame's content again. However, doing it like this is a bad idea and ugly programming style in Python. Due to that, comment out these 4 lines above in your code.

However, customers might have responded to the review request only with "I am not sure", and therefore we could have leave in the dataset values like 0 good and 0 bad reviews. It is obvious that 0 is not a good value to use as a default for any of these columns. Programmers tend to use -1 in these situations, but `pandas`' users typically employ in such situations a special value named '`NaN`' (meaning Not a Number, and not granny ☺). To add three columns filled with these values, we can write code that is more elegant than the above:

```
for newCol in ['nrGoodReviews','nrBadReviews', 'nrUndecided']:
      <the name you gave to your object>[newCol] = np.nan
      # where np is the alias for the numpy library
      # which explicates the .nan value for Python
```

There are many ways to add new columns, but this has the advantage that takes a list of new column names as a sort of input, and it works for a list of any length.

INTERESTING NOTE: You can also use the Python built-in placeholder `None` instead of the typical `numpy` and the `pandas` related `np.nan` value. We need these placeholders to show in data repositories that we do not have yet a value for that particular data point or record. In other programming languages (C, C++, Java), this "nothing" placeholder value is called `null`. Which is a very different thing from having a 0 or a -1. To understand better the nuances and implications of this particular kind of value for data repositories, watch the following excellent educational video: https://www.youtube.com/watch?v=bjvIpI-1w84

Display again the whole data frame content in the console. You see that the restaurants are in a random sequence, and if somebody would like to input manually the test results, by overwriting the `NaN` values, it would be slow to find the right row. For example, if the students appear in the alphabetical order of their surnames, the task to find a specific student's row would be easier. Sorting the content of a data frame, based on one or more columns, is very easy. The `DataFrame` class has a `sort_values()` method, and you can use it as to order the restaurants on two criteria, name first, and address second:

```
<ordered_dataset> = <pandas>.sort_values(by = ['name', 'address'])
```



Display the data, and then save the content of the newly ordered data frame into a new csv file – which you will name as you feel fit. This file should not have the index in the first column (search yourselves how to achieve that), and the separator should be the comma this time, and not the semicolon like in the initial file. Open this newly created csv file with Atom to see how it looks. You can remark that either with using `None` or using `np.nan`, the effect on the output file is the same.





To continue this exercise, we assume now that somebody has completed the missing values in the file, and we have a new file, named '`rankingsRaw.csv`' (download it from Nestor), where the number of good, neutral, and bad reviews are completed (NOTE: the numbers were generated randomly, these **are not** real restaurant reviews).

IMPORTANT NOTE: As you may expect, the file `rankingsRaw.csv` was not completed by hand. As data scientists, we should also be quite versed in generating test data (mostly randomly). The .py file used to generate the random results will be provided to you together with the solution files that are provided at the beginning of the next week. We also show later here how we can even generate a file with randomly generated names – we used real restaurant names this time, but for various purposes, due to reputation/liability/privacy legal reasons, we may want to use an "anonymous list" of thousands of fictional names – which have to be generated somehow.

We assume that the formula to compute the ranking score is (based on the net promoter score, or **NPS formula** – see details at https://www.checkmarket.com/blog/net-promoter-score/):

```
score = (g / (g + b + u) - b / (g + b + u)) * 5.0 + 5.0
```

…where `g` is the number of good reviews (surveyed customers who awarded a specific restaurant for example with a 9 or a 10), `b` is the number or bad reviews (graded with integer values between 0

and 5), and `u` the number of in-between (6, 7, or 8). The grade thresholds for good, bad, and undecided are up to the data analyst to decide. Note that the formula above will yield always a score (a real number) between 0 and 10. And it will generate a "division by 0 error", if there are 0 customer responses. Therefore, we should never try to compute a score when (g+b+u) is zero. Normally, marketing analysis is done only if there are enough answers (over a certain threshold) – because statistics is meaningless for small numbers.

EXERCISE 2 (write the code in NEW .py file named for example practical_week4_ex_2): This will read the content of (the given) `rankingsRaw.csv` into a data frame object, compute the score for all restaurant, add this to the data frame, and save the data into a new csv file.

First, write a function that computes the score:

```
def nps_formula(g, b, u):
# computes the Net promoter score in a range from 0 to 10
    value = (g / (g + b + u) - b / (g + b + u)) * 5.0 + 5.0
    return value
```



This function can be used, by invoking (calling it), to add a new score column to the data frame with all the computed scores:



```
<df>['score'] = nps_formula(df.nrGoodReviews,
                            df.nrBadReviews,
                            df.nrUndecided)
```



where `<df>` is the placeholder that in your code is the actual name of the data frame object you created when reading the file – which can be in fact be exactly `df` if you want (many examples in tutorials use for didactic reasons this short object identifier, that is, `df`, but it is recommended to use always a specific identifier that reflects the nature of the data, like `rest`[aurants], or `taxi`[Drives], for this practical for example).

Order the rows of the data frame by the values of this new `score` column, descending (search the on-online documentations, e.g.: https://www.geeksforgeeks.org/python-pandas-dataframe-sort_values-set-1/ , to see how to achieve that, because the default sorting results is ascending).

Move the score column to make it the first (leftmost) column of the table. Display the ordered table and save the ordered data frame into an indexed (this time, because it will show ranking) csv file named for example '`finalRestaurantScores.csv`'.

HOMEWORK (in another .py file): take the first csv file you have created after extending the data frame with `NaN` or `None` values. This file has no values for the columns for good and bad answer counts. Complete the results in the file (by hand, with notepad) for a few (4-7) restaurants only, leaving the rest of the rows unchanged. Read the file into a data frame and compute the scores only for the rows which have raw results completed by you. Create two output files, one with restaurants

who had results and have a final score, and restaurants who do not have (the format of this one should be similar to the initial file, but the restaurants who have now a score should not appear here anymore). HINT: explore the on-line documentation to learn how `NaN` is used in Python and how the `isnull()` method is used.

### 5.4. Some simple statistics using data frames from **pandas**

After computing the scores, we want to get some statistical knowledge about the grades, and also their distribution. HINT: start a new .py file that reads the `finalRestaurantScores.csv` file into a data frame (for simplicity, in the code below, the name of this data frame object is `df` – however, in your code, better use a more specific identifier, like restaurants).

First, a very basic operation is to count how many non-`NaN/null` scores we have, by using the `count()` method of the `DataFrame` class:

```
print(df['score'].count())
```



The method can be invoked for the entire data frame:

```
print(df.count())
```



Delete some score values from the input csv file (use Atom!), leaving the commas untouched (by this you introduce `NaN/null` values) and run the code again. See the difference with the first run. The complete documentation of this method is at:
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.count.html#pandas.DataFrame.count



Other two useful methods of the `DataFrame` class are similar to built-in `min` and `max` Python functions that can be used for all built-in collections, like lists. Here, the methods can be applied either to a single column (or row, by changing the axis parameter), either to an entire data frame:

```
print(df['score'].min())
print(df['score'].max())
print(df.min())
print(df.max())
```

The `mean()` method is computing the average of a given column (or row, if asked accordingly):

```
print(df['score'].mean())
```

If applied to the entire data frame:

```
print(df.mean())
```

it will compute the average for all columns that are numerical and can be computed – obviously the columns 'name' and 'address' cannot be "averaged".
The statistical analysis of any numerical data set gets more meaning if the standard deviation of a sample is known. The method `std()` does this for a data frame:

```
print(df.std())
```

And it can be also applied for an individual column only (try it). You should also apply to the grade column the methods `sum()`, `median()`, and `nunique()`:

```
print(df['score'].sum())          # Total sum of the column values
print(df['score'].median())       # Median of the column values
print(df['score'].nunique())      # Number of unique entries
```

Finally, a powerful method that will give multiple answers in the form of a statistics summary of the data in a data frame or a part of it (like a single column) is `describe()`. This will summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding `NaN` values. You can parametrize the `percentiles`, like below:



```
print(df.describe())  # here the default is [0.25, 0.5, 0.75], which
                      # returns the 25th, 50th, and 75th percentiles
print(df.describe(percentiles = [0.15, 0.3, 0.45, 0.6, 0.85 ]))
```



As with the previous statistical methods, the result will include all the numerical columns only.



## 5.5. Graphic visualization of data frame content

A useful insight would be to see visually the distribution of the grades in this specific data frame. The easiest way is to use a specialized method named `hist()` (which displays a histogram or more, depending how many numerical columns can be analyzed). Try both – for an entire data frame, and for one column only:

```
df['score'].hist()
df.hist()
```

For this particular set of data, it is not necessarily a good idea to try a pie chart. To show how this would work, use the following code (in a separate .py file), which creates a data frame object from a simple dictionary with three entries for two keys. Each entry refers to a planet (the masses in kg are reduced by 10^24, and the radiuses are in kilometers):

```
df = pd.DataFrame({'mass' : [0.330, 4.87 , 5.972],
                   'radius' : [2439.7, 6051.8, 6378.1]},
                    index = ['Mercury', 'Venus', 'Earth'])
# we can select only one key/row if we want
df.plot.pie(y='mass', figsize=(5, 5))
df.plot.pie(y='radius', figsize=(5, 5))
# but we can also plot all in one shot
df.plot.pie(subplots = True, figsize = (6,3))
```

You can find more information about plotting various types of data graphics at:
https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

## 5.6. Measure and visualize a real data file

In the previous week, for practical B, you have been asked to visualize on a map output the track of a stroll which was traced via a GPS device where the data generated was saved in a .gpx file. This exercise continues that exercise, but the data this time is about taxi rides (in NY city). In the JSON file `taxiRuns.json`, provided on Nestor, there is detailed data about a limited number of taxi runs, in a record-oriented JSON format (each "{<one run data>}" structure in the file is a record of one taxi run, with keys and values similar to the dictionary type syntax of Python. The records are separated by commas, and the file starts with a [ and ends with a ], like a list in Python (however, this is a JSON file, and not Python code).



First, start a new Python program (named like `week4_ex5.6.py`). To read the content of the file in your program, use the simplest way possible:

```
taxi = pd.read_json(os.path.join(sys.path[0],"taxiRuns.json"),
                    orient = 'records')
```

This will create a pandas DataFrame object, where each row is a taxi run. You have to use the pandas methods you have already learned to investigate the shape, size, and statistics output of this data collection. You will see that the data for each run contains geolocation coordinates for the pickup and drop off points of the taxi run.

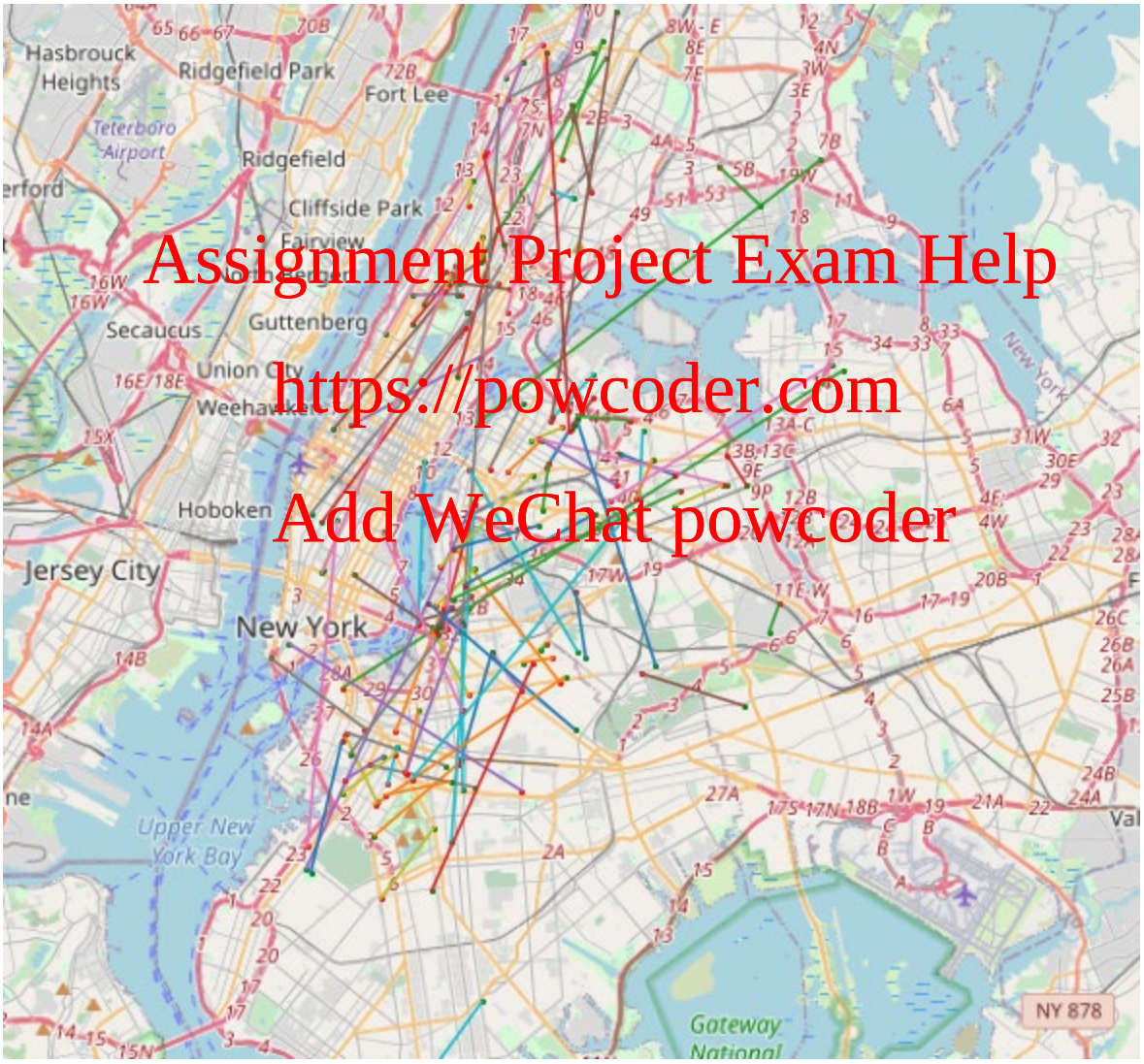
EXERCISE: By using the `smopy` library you have used in the previous B practical, visualize these points on the NY map as below, and also unite for each run the start and the end of the run.
HINT: to show a line on the map, use the plotting stament:

```
ax.plot([x1,x2], [y1,y2])
```

Where these pixel coordinates are generated (pair by pair, in a `for` statement) like in the previous exercise, from geolocation coordinates, using the `map_to_pixels(…)` method from `smopy`:

```
x1, y1 = map.to_pixels(pickup_points[i][latitude],
                       pickup_points[i][longitude])
x2, y2 = map.to_pixels(dropoff_points[i][latitude],
                       dropoff_points[i][longitude])
```

where `i` is the index used to iterate through all the runs, and the object `map` is generated by using `smopy.Map(…)`. The output of your program should look like:






Obviously, you will have first to find out the corner coordinates of this map, by identifying the minimum latitude and longitude and the maximums of the existing geolocation points in the data collection given.

## 5.7. For the curious student

EXERCISE 1:
`Smopy` is a rather limited and simplistic library, easy to use but which does not allow to add map place markers or zoom interactively after the map is generated by your script. A more powerful tool is folium (https://python-visualization.github.io/folium/), which generates an html map file, which you can visualize in a browse (you cannot see directly the output immediately when running the program in Python, because Atom does not visualize html). On Nestor, a `foliumExample.py` is provided, along its output file, `foliumTest.html` (try to use zoom zoom in this file, and see in which US city the place markers are placed).
Change your exercise 5.6 solution from using `smopy` to `folium`.

EXERCISE 2:
Horeca owners will feel uncomfortable when their restaurant name appears in files used for students, as this example file with randomly generated scores. This negative feeling is indeed normal and somehow expected, and in more formalized institutions it is not even allowed to use real data – even for test data. To solve this issue, luckily others worked on it, and we can use a special module named (appropriately) `faker` which offers a `Faker` class that can be instantiated in objects that can generate names of different kinds (more about this module at: http://zetcode.com/python/faker/). See an example of code below, generating people's names – you can use it in a new .py file (don't forget you have to install the `faker` module with `pip` on your computer):



```
from faker import Faker
import numpy as np

output_file = "fake_data.csv"
fake = Faker('nl_NL')

with open(output_file, mode='w') as output:
    output.write("first_name,last_name, scores\n")
    for _ in range(20):
        output.write("%s,%s,%f\n"%(
            fake.first_name(),
            fake.last_name(),
            np.nan))
```

You can change the parameter of the `Faker()` constructor to for example `'fr_FR'` or `'hu_HU'` and get French- or Hungarian-sounding names. Observe above also the use of a special (anonymous) variable in the `for` construct, denoted by a single underscore, and also the formatted-string write operation. More about this Python formatting feature you can study on a very nice free DataCamp webpage: https://www.learnpython.org/en/String_Formatting

Make a similar program, one that generates restaurant names and addresses.