4. Practical 3a (on the basics of Data Frames)

4.1 Using the **pandas** to structure stored data and explore its content

Content to be covered during this practical (questions to be answered):

- How to import `pandas` or any other package in your program
- What is a `DataFrame` object and how to populate it with data
- How to select rows and/or columns in a `DataFrame` object
- How filter for specific records
- What is a module, package, and library?

4.2 What is importable code and what does it offer?

`pandas` is one of the most popular libraries for the programmers that want to do Data Science and Analytics in Python. The `pandas` library helps the programmer to manage elegantly and efficiently one-dimensional series, and also two-dimensional data tables. We enacted already a table-like two dimensional data structure in the previous practical, but only by using a mere list of lists. By using **classes**, objects, and **methods** from the `pandas` library we can be more effective programmers, achieving quickly analytical code that is less error prone and easier to manage.

NOTE: Data sets with more than two dimensions in `pandas` used to be called `Panels`, but these formats have been deprecated, and cannot be used anymore. Today the recommended approach for multi-dimensional (>2) data is to use the `Xarray` Python library – even if this was initially developed for homogenous matrices/mathematic purposes. It may be that in the future, a new library will be created for multi-dimensional heterogeneous tables. Keep yourself constantly informed about the novel developments in Python for Data Science.

Before we dwell into `pandas`, you should be aware that Python has a Standard Library (PSL) with dozens of built-in modules (i.e. always available when installing only the Python interpreter on your computer). For example, the module `csv` (used already in the previous practical) is part of the PSL. For data analytics, amongst the most commonly used are: `random`, `statistics`, `math`, and `datetime`. All these have to be explicitly imported in your program. The reason that they are not automatically included in your running code is one of performance; that is, a program runs better if it contains only the necessary code to be executed and nothing else. Adding the whole PSL to all Python programs will add all the functionality that exist there, but by this it will increase the memory-size needs, and will inevitably decrease execution speed, which is already considered an issue with Python programs. These performance indicators are crucial on special hardware, like micro-controllers, which have limited local memory and computing power, or on computing servers that are powerful enough, but have to execute concurrently (i.e. in the same time) thousands of programs permanently.

To include an external piece of code in your program, you have to use the `import` directive, which has the following generic syntax:

```
import <module_name>
```

```
# e.g.
import random # this will import code that allows you to generate
              # random numbers and other related functions
```

This way will import the entire module (and it will reduce performance). It is possible to import only a part of a module by, using: `from <module_name> import <item_name>`

```
# e.g.
from csv import reader
from random import random
# you include only the code for a single method and nothing else
# this method below just generates a random value

random_seed = random() # this will generate a float between 0 and 1
```

IMPORTANT: If we use the first import line above to import functions/methods from the `csv` module, we can use for example the reader function without needing to write the wholly dot-qualified method call `csv.reader(handler)`, but only `reader(handler)`.

We can import functionality from Python files that we wrote ourselves. For example, the `import timing`, in the case that we wrote a Python script named `timing.py` imports functionality from this own Python program, which has to placed in the same directory with the program that imports it – or we set the Path environment variable to have it visible for the Python interpreter..

If we import the module `statistics`, we have access to a number of useful methods, very often used in Data Analytics problems.

```
# e.g., consider the vector a, represented as a list in Python
a = [-16, 9, -4, 5, 66, 34, 28, 22, -90]

av = statistics.mean(a)
med = statistics.median(a)
mod = statistics.mode(a)
sigma = statistics.stdev(a)
var = statistics.variance(a)
print("vector", a, "\nhas ", av, med, mod, sigma, var)
```

EXERCISE: This code is correct, but still, it gives one wrong answer, why? HINT: it's mathematics (specifically statistics, look up the definition of "mode"), not programming that gives the answer. When does it work? What do you have to do to make it run?

To find out more about the PSL and what it does offer, the best location is:
https://docs.python.org/3/library/

In addition to the PSL, there is an ever growing ecosystem of thousands of stable and reliable modules/packages/libraries freely available. To see what is on offer, one has to search at:
https://pypi.org/

If you have not installed the necessary software on your computer, you have to use the `pip` terminal command:

```
C:\>Python –m pip install pandas
```

(on the Mac, refer to the python executable that you are using in Atom, e.g., python3.7 or python 3.8)

Pandas is built on top of the `numpy` package, which is a powerful library for scientific computing. The `pip` command (for Python3.8.x) recognizes such dependencies and will always install all libraries that are needed for your requested library. Therefore, the command above will not only install `pandas`, but also `numpy` and `pytz` (which is a quick time zone calculation package). It is a typical Python programming convention to import both `numpy` and `pandas`, because many programs use the functionality of both these libraries. An additional convention is to use the following aliases when importing the libraries in your program:

```
import numpy as np
import pandas as pd
```

Now you can refer to `pandas` as `pd`, and to `numpy` as `np` in your program. You can of course also use other aliases, but these are the most commonly used in the programming community. So if you encounter code snippets in examples or websites such as StackOverflow, you can assume that if they use `np` or `pd`, they refer to these libraries.

Here are a few things that `pandas` can do for you:

- Easy handling of missing data
- Easy change of table structure (insert, delete columns)
- Easy conversion of data
- Label-based data slicing, "fancy" indexing, subsetting of large data sets
- Merging and joining of large data sets, etc.

Of course, all these can be programmed by you, without using pandas. The main advantage of using pandas is that the code is already written, tested, it is quite fast, and very reliable.

The library offers two import data structure "types": `Series`, and `DataFrame`. These are both implemented as classes and if you write:

```
specificDataFrameObject = pd.DataFrame(myData, myColumns)
```

by invoking the constructor of the class `DataFrame` you will created a `DataFrame` typed object named `SpecificDataFrameObject`. Obviously, the variables `myData` and `myColumns` will have to be defined before the code line above. The next example illustrate how this is done.

4.3 Create a `DataFrame` object directly in the memory

To understand how to describe the input for the operation above (i.e. the variables `myData`, `myColumns`), we will create a small data set coded as a Python dictionary data structure (if you

have not studied the dictionary type yet, now it is a good moment to start, for example in W3SCHOOLS, see the link in the slides of the third's week video lectures).

Consider the following information: the structured data about some former formula one champions in the 1950s. In a Python built-in dictionary data structure, we can capture some information about them as follows – very much like in a database table:

```
data_in_dict = { "year" : [
            1950, 1951, 1952,
            1953, 1954, 1955,
            1956, 1957, 1958, 1959
            ],
         "champ" : [
            "Farina", "Fangio", "Ascari", "Ascari",
            "Fangio", "Fangio", "Fangio", "Fangio",
            "Hawthorne", "Brabham"
            ],
         "wins" : [
            3, 3, 6, 5,
            6, 4, 3, 4, 1, 2
            ],
         "points" : [
            30, 31, 36, 34,
            42, 40, 30, 40, 42, 43
            ]
        }
# end of dictionary data structure
```


HINT: you can copy/paste this in your Atom editor, this time I made sure that it will not generate errors.

Observe that we have four dictionary key names (year, champ, wins, and points), and for each key, we have a list of ten values. We can for example add later to the dictionary a new key with values, for example, the gender, like this:

```
data_in_dict["gender"] = ["m", "m", "m",
                  "m", "m", "m",
                  "m", "m", "m", "m"
                  ]
```

If you print this data structure:

```
print("printing first as a Python dictionary: \n", data_in_dict)
```

You notice that this will "spill" out a rather unfriendly stream of characters, strings, data, and delimiters mixed-up together. One of the first advantages of using a `DataFrame` from `pandas`

is to have nicely formatted output for the content of our data structures. Now, we will transform this dictionary into a `DataFrame` object of the class `DataFrame` in `pandas`:

```
formula_One = pd.DataFrame(data_in_dict, columns = [
        "year", "champ", "wins", "points", "gender"
        ]
    )
```

This will or course only work if you already imported in your program the `pandas` under the alias `pd`. If you print this object instead of the initial dictionary:

```
print("printing as a pandas DataFrame object:")
print(formula_One)
```

…you will notice the difference in formatting and user-friendliness. Most of the time, we collect lots of data and we may not know the structure of our `DataFrame` object. The class offers a number of methods that help us to identify the size and nature of the data structured as a bi-dimensional table. Try for example these methods and identify the type of returned value in each case:


```
print("the size of the table is rows * columns:")
print(formula_One.shape)
print("the rows are organized as:")
print(formula_One.columns)
print("the Python type of the values on the columns are:")
print(formula_One.dtypes)
```


Finally, we can store easily this table in a file, using another method of this very useful `pandas` class.

```
formula_One.to_csv('f1_fifties.csv')
print('data frame written to csv file')
```

Open the newly created file with the Atom editor (or Notepad), and observe that each row has now an index, which becomes the first column of the data structure.

4.4 Manipulating `DataFrame` objects

For example, we would like to add a new column to our `formula_One` table-like object. The syntax of this operation is very similar to the syntax we used to add a new key (gender) in the previous Python-only dictionary. We will add a column for the team (car brand). First, we create a list with exactly 10 items, in the exact historical order of wins – in a manner that may be new for you (via list multiplication and list concatenation using arithmetic-like notation, possible only in Python and in a few more languages):

```
team_wins = ["Alfa"] * 2 + ["Ferrari"] * 2 + ["Mercedes"
      ] * 2 + ["Ferrari", "Maserati", "Ferrari", "Cooper"]
```

EXERCISE: rewrite the statement for the gender list, based on what you see is possible above.

The statements that adds the new column and print the object are:

```
formula_One["team"] = team_wins
print(formula_One)
```

The column that shows the gender is rather useless (unfortunately, for long debated reasons, formula one remains still a male dominated sport). We can safely remove this monotone column, at least for a data repository that contains only the year fifties.

```
del(formula_One["gender"])
print(formula_One)
```

We can select from a `DataFrame` columns or rows. For example, the methods `.head()` and `.tail()` select by default the first 5 and the last 5 rows respectively. This 5 is a default number, you can indicate another explicit value, as below.


```
print(formula_One.head())
print(formula_One.tail(3))
```


We can also select a part of the table (last four rows, and only the champion's name):

```
print(formula_One["champ"][-4:])
```


This will select only one specific column and the last 4 rows. We can select more columns (pay attention, the notation becomes a bit tricky, note the double square brackets for the list of lists):

```
print(formula_One[["champ","year"]][2:4])
```

You notice that if you change here the order of the column names, the order of the returned columns will change also. Such a selection creates another object of the class `DataFrame`, with less columns and rows (the use of `head()` and `tail()` and the limited row selection above return the same kind of objects, but with different sizes and content).

If you import the average computing function from the `statistics` module, you can compute the average score of the fifties, like this:

```
print("average point score in the fifties:", mean(formula_One["points"]))
```

You can filter out specific rows:

```
only_Maserati_seasons = formula_One[formula_One['team'] == 'Maserati']

print(type(only_Maserati_seasons))

print(only_Maserati_seasons)
```

Here we create first another object of the class `DataFrame`, we print its type (just to convince ourselves) and we print it separately. In the selections we made above, the objects created were

anonymous and lost after their print statement, but in this case, we can use the variable /object `only_Maserati_seasons` later in our code..

We can successively filter our data set into more and specialized selections. Below, we look only for seasons won in a Ferrari by Juan Manuel Fangio.

```
only_Fangio_seasons = formula_One[formula_One['champ'] == 'Fangio']

only_Fangio_driving_Ferrari_seasons = only_Fangio_seasons[

    only_Fangio_seasons['team'] == 'Ferrari']

print(only_Fangio_driving_Ferrari_seasons)
```

4.5 Reading data from a file into a `DataFrame` object

We have seen that the structured information content of a DataFrame object can be stored in a file (in this case, a .csv file, but other types are also possible). Let us load from our previous practical data file the information about Groningen restaurants into a DataFrame object (this time, make sure that you are not touching the original file with Excel).

For this exercise, make a new Python file. Make sure that the original data file (with records that have exactly 3 data points in each line) is in your working directory or you know exactly its directory path and you imported `pandas` in your program. After we read the data from the file in the DataFrame object restaurants, we invoke another useful method of this class, which randomly selects a sample of rows of a given number (the default size is 1), and we print only a few columns we want to see.


```
restaurants = pd.read_csv("groningenRestaurants.csv")
# shorter and easier than the with, open, read, list() construct
random_selection = restaurants.sample(12)
print(random_selection[["restaurant","lonlat"]])
```


Notice how simple is the statement to read the information from the file. However, this is rather deceptive, because the method `read_csv()`, besides the obligatory parameter that is the string giving the file path and its name, can have quite a few (very useful) parameters.

EXERCISE: write code that identifies the size, columns, and data types for the `restaurants DataFrame` object. HINT: you did this already for the formula one example.

We can read into a `DataFrame` object information from other types of files, like Excel (.xls or .xlsx) files, or even SQL database files. The names of the methods that read these files are unsurprisingly `pd.read_excel()` and `pd.read_sql()` – for their use, you can find easily various on-line documentation if you need it.

If you try to display in the Atoms's console the content of the `restaurants` object, Python and the `pandas` code will display the restaurant names first, and then the addresses and the lonlats. Moreover, if the number of lines is also big (>100), only the first and last five records are displayed by `print(restaurants)`. The same is true for the number of columns, if they are too many, only the first 5 and last 5 are printed.

These are only default settings, which can be changed easily, for example like below, by changing the settings in a manner that will display all the columns (known in the `shape` attribute of the object) and all the lines (also known in the `shape` attribute). For the width of the console screen, it is possible to set a large number of characters, and scroll to see the whole line if not visible:

```
pd.set_option('display.max_rows', restaurants.shape[0])
pd.set_option('display.max_columns', restaurants.shape[1])
pd.set_option('display.width', 1000) #characters in one line width
print(restaurants)
```

Which will set the display parameters to values that fit exactly the size of this particular table. Remember that the `shape` attribute contains a two-value tuple: first the number of rows and then the number of columns.

EXERCISE: Try to filter out the same information you have filtered out in the previous practical (e.g. for names with "pizz", "eet", and addresses with compass point names).

HINT: search for solutions on the web, using for example the phrase 'select by partial string from a pandas DataFrame'.

4.6 Homework

Find an interesting (for you) .csv file on the internet and import it with the new methods studied. Study its structure and size. Filter information you find interesting. Format the displayed output in a way that makes it user friendly and easy to look at it. There are many large data repositories publicly available for both study and research. For example you can download csv files from the Data Science competition website, Kaggle:

https://www.kaggle.com/datasets

A useful possible example to start with in Kaggle is the Global Food production data, which contains 21478 records, each corresponding to a food source from a specific country. The first columns represent information about the country and food/feed types, and the remaining columns represent the food production for every year from 1963 to 2013. Download this file at:

https://www.kaggle.com/dorbicycle/world-foodfeed-production

4.7 For the very curious student only

Students have a hard time understanding what exactly is imported. In various on-line and printed documentations these things are referred as "modules", in some cases as "packages", in other cases as "functions", and sometimes as "libraries". If one wants to keep up a meaningful discussion with a developer (or even just ask a question on Stackoverflow), one has to have at least a clue about what is what. A short and incomplete explanation is given below:

- **Function:** it is a block of code that one can (re-)use by calling it with a keyword. E.g. `median()` is a function. In a Object Oriented Programming (a way of programming, possible in Python also), if a function is part of a class definition, some programmers, especially those who also working in Java and C++, speak about these functions as "methods".
- **Module:** it is a .py file that contains a list of functions (it can also contain variables/objects, and class definitions). E.g. in `statistics.mean(a)`, `mean()` is a *function* that is found in the `statistics` *module*.
- **Package:** it is a collection of Python modules. E.g. for the function call `numpy.random.randint(2, size=10)` the `randint()` is an imported *function* in the `random` *module* of the `numpy` *package*.
- **Library:** it is a more general term for a collection of Python codes. E.g. if you install manually (with `pip`) the `pandas` (considered a library itself) this shell command will install `numpy` also (which is considered a package), therefore distinctions become blurred.

In the end, you have to be aware that sometimes the programmers jargon is rather flexible, even if they tenuously try to be very precise about what they mean. In this particular case of naming the chunks of code that are importable, they still continue to debate terms. If have time, and you are of a curious nature and also patient, an intriguing experts' debate on the subject can be found at:

https://stackoverflow.com/questions/19198166/whats-the-difference-between-a-module-and-a-library-in-python