2 Practical week 2, part A (on the basics of built-in Data Structures)

2.1 Starting to analyze data sourced from external storage (csv files)

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

- Reading a table in a list of lists from a csv file
- Using some list operations
- Introducing the **for** statement to display the file row by row
- Introducing the **if** statement to select data points from the table
- Adding a new row with data from the keyboard
- Saving the table in a csv file
- Using methods for other types: string operations

3.2 First, how is data stored in a csv file

A CSV (**c**haracter, or more typically **c**omma, **s**eparated **v**alues) file is a fairly simple text file containing structured information. The values that are stored are separated by a special character, typically a comma or <TAB> character. The simplicity of this file organization is its main feature. CSV files are designed to be a way to easily export data and import it into other programs, and they are the bread-and-butter of data science storage means. The data in such a file is human-readable and can be easily viewed in Windows with a text editor like Notepad or a spreadsheet program like Microsoft Excel, and with similar tools in MAC OC and Linux(es). The convention to name these files is in to give them the extension ".csv".

For our programming exercises, today we will use a file that contains data about restaurants in our city (groningenRestaurants.csv). The file is in fact a stored table (you may visualize it in Excel if you want, but do not save it after). Each line (row) of this file – except the first line - is a "data record", where individual "data points" are separated by commas. It is important to notice that the last data point on any line is not followed by a comma.

The first line contains the names giving the meaning of the data points on the following lines – basically, it is a table header. All data records keep the number and order of the data points in accordance with this table header.

The information contained in the external storage has to be brought into the memory in order to be analyzed with the help of your programs. Normally, if a file has a consistent and correct structure like this one, it would be straightforward to use a special data structure from the `pandas` module. However, many files do not have a consistent structure, for example the number of data points on different lines can differ ("incomplete data records"). Therefore you have to be able to read

information from files that are not necessarily correctly structured. The `.csv` file is provided to you on Nestor.

3.3 Bring data records from a CSV file into a list-only based structure

To read a .csv file, you have to use the `csv` module ("import" it in your program), open the file with the `with` statement (and creating a "file handler" object). Then, by using the function `reader(<file handler>)` from the `csv` module, you transfer the content of the file into an object/variable in the memory – which is a collection of characters only, and cannot be easily be investigated. Finally, you can transform this raw data object into a common data structure, for example a list of lists, by using the built in function `list()`. In the code that follows, the names of the file handler, the raw data object, and the list of lists object are user given, therefore, we encourage you to write your code with different names for these objects.

```
import csv
with open("groningenRestaurants.csv") as handler_csv_file:
    raw_content_file = csv.reader(handler_csv_file)#this object is a
                                                   #collection of characters
    table = list(raw_content_file)                 #table is a list of lists
```


NOTE: If Python cannot find the file, the full path should be specified. For example, in Windows, right-click on the filename in the project explorer, and choose 'copy full path'. Then paste this as filename. After that, change all backward slashes (\) to forward slashes (/).

The first investigation is to see how many data records are in the table.

```
print("data records = ", len(table)) #run now Program.py
```


This number tells you how many lines the file contains (58, in this particular case), by identifying how many elements are in the list table. These elements are lists themselves. If we print the first element of the list table, we will have the table header presented as a list.

```
print("table header = ", table[0])
```

REMINDER: in Python, like in the majority of programming languages, the indexing of collections starts always with 0. Python also does inverse indexing (starting from end of a collection), by using negative indices. We can easily print the last data record in the table, without bothering about its length, by:

```
print("last record in the table = ", table[-1])
```

Because the table is a list of lists, we can index it as a two dimensional matrix. For example, if we want the name of the restaurant name from the last record, and we know that the name data point is in the first position, we do:

```
print("rest. name in the last record in the table = ", table[-1][0])
```

We can also investigate the length of a record (how many data points), by:

```
print("data points in a record =", len(table[0]))
```

However, this gives only the length of the table header – the rest of the entries can be a total mess still. Next, we will check if all the records are of the same length.

FOOD FOR THOUGHT: there are more than two commas on a line of the file, not only the ones separating the name from the address, and the address from the geo-coordinates. There are more in the address, and one inside each lon-lat coordinate. Why we have only THREE data points on a (correct) line then?

3.4 Iterating trough the data structure

The typical Python statement used to iterate through collections is the `for` statement. We can print the length of all the records in the table, by:

```
for record in table: print(len(record))
```

However, it is preferably to code such a compound statement on two lines of code, the second being indented:

```
for record in table:
    print(len(record))
```


This style is necessary if we have multiple lines of code in the "body" of the `for` statement. If we want to print for all the records, the restaurant address, the latitude and longitude coordinates, and the name (in this particular order), we execute:


```
for record in table[1:]: #note the indexing, it skips the header
    address = record[1]
    geolocation = record[2]
    name = record[0]
    print("\nAt:", address,"\ncoord.:",geolocation,"\nis:", name)
```


It would be shorter to write:

```
for record in table[1:]:print(record[1], record[2], record[0])
```

…which prints the same information. However, it is always better to make the code as explicit and easy to read as possible, and the output of displaying as user friendly as possible. Here the printing of one record results in the format imposed by the pretty printing code above:

```
At: Verlengde Hereweg 46, 9722AE, Groningen
coord.: 6.5798117,53.1987885
is: Alice Restaurant
```

For example, printing the length of all records will make the job of the user visually checking - if all are the same - rather difficult here (with 58 records), and impossible if the data has thousands or

millions of records. To be able to check the length consistency, we may use logic expressions and the `if` statement.

```
expected_record_length = len(table[0])
consistent = True
for record in table[1:]:
    consistent = (len(record) == expected_record_length)
    if not consistent:
        print("ALERT, ALERT, ALERT")
        print(record, "has", len(record), "data points")
        break
    else:
        continue
```

First, the length of the header is store in the `expected_record_length` variable. The code above uses a Boolean variable (`consistent`), which is set on `True` before we start to iterate through the table. That means that we assume that the table has all records of the same length with the header.

The first statement compounded inside the `for` is assigning a new value to the Boolean variable, based on the result of the logic expression comparing the current record length with the expected one. If this comparison yields `False`, the `if` statement will execute the first branch, because the `not consistent` expression is true. The iteration stops after the alert display because of the break statement. The `else` branch is not absolutely necessary, but it is added for clarity, showing explicitly that the iteration through the table continues if the length of the record compared is the expected one.

To test this code, open the csv file, and add a comma at the end of a record (line) in the middle. You may alter more than one line. Save the csv file before running the code. You will remark that the code stops at the first incorrect record (because of the `break` statement). But we can have more than one incorrect records.

If we want to catch have all the records that have an incorrect length, then we iterate through the whole table, printing each record with an incorrect length. This can be achieved easily by commenting out the `break` statement. We can also count the number of records with the incorrect length and display the total counted. Make sure that the csv input file contains more than one record with incorrect length (say, 3-4).

```
expected_record_length = len(table[0])
wrong_length_record_counter = 0
for record in table[1:]:
    if not len(record) == expected_record_length:
        wrong_length_record_counter += 1
        print(record, "has", len(record), "data points")
print("In total, ", wrong_length_record_counter,
                            "times, the record length is wrong")
```

An alternative to this approach is to make a separate list with the incorrect records, to be analyzed separately, or if very few, corrected by hand in the file. This separate list can be used to eliminate the incorrect records from the table.

```
incorrect_records = []
expected_record_length = len(table[0])
for record in table[1:]:
    if not(len(record) == expected_record_length):
           incorrect_records.append(record)
print("In total", len(incorrect_records),
                           "times, the record length is wrong")
for bad_record in incorrect_records:
     print(bad_record, "bad length =", len(bad_record))
```

EXERCISE for you to figure out (difficulty: medium, you can leave it for the homework): Based on the `bad_record` list, eliminate the bad records from `table`. To solve this, you have to discover which method for the Python list data type is adequate to remove elements from a list.

3.5 Selecting records and data points from the table

Based on the pattern of iteration (with for) and check (with if), we can display only records and data points that are interesting for our analysis. For example, we can display only the partial records of the restaurants that have a name that starts with the letter "D", and an address that starts with the letter "R" (there is one restaurant in the file that fulfils this requirements):


```
for row in table[1:]:
    name = row[0]
    address = row[1]
    if name[0] == "D" and address[0] == "R":
        print(row)
```

The variables `name` and `name` are strings. In Python, we can check if a string is part of another string by using the `find()` "method". For example, if we execute to following code:

```
sentence = 'geeks for geeks'
# returns first occurrence of Substring
result = sentence.find('geeks')
print ("Substring 'geeks' found at index:", result)
```

The variable result will give us the position of the first occurrence of the substring. We will learn more about string related methods, because these are very useful.

To only check if a substring is part of a string, we need to use an `if` statement like this:

```
if (sentence.find('padawan') != -1):
    print ("Contains given substring ")
else:
    print ("Doesn't contains given substring")
```

EXERCISE(s):

Display all the restaurants that contain the substring "Pizz" or "pizz" in their name.

Add the condition that they also contain "Eet" or "eet".

Find only those restaurants that have "Zuid", or "Noord", or "West", or "Oost" in their address, irrespective of capital letters or not ("zuid", etc, shoud be also found).

3.6 Changing the content of the table and saving the new data into another file

Look at the following code:


```
name = input("restaurant name is: ")
address = input("restaurant address is: ")
lonlat = input("coordinates are: ")
new_row = list((name, address, lonlat))
table.append(new_row)
print(table[-1])
```


We can introduce in this way a new restaurant. Moreover, we can save the new list of records as a new csv file:

```
with open("groningenRestaurants_v1.csv", mode="w") as handler:
    file_writer = csv.writer(handler)
    for row in table:
        file_writer.writerow(row)
```

Inspect the output file after running this code to see the difference.

EXERCISE (this may be homework if you do not finish during the pratical): write code that allows the users to introduce more than one restaurant (like in the code above), and allows him to stop when they want, by asking "do you want to input one more restaurant? (y/n)". By answering "n", the process stops and the file is saved.

HINT: better use a Python function (find out on the WWW how to define and use a function). It is not necessary to use a function, but the code will look much better, and will be easier to manage.

3.7 For the advanced: solve a tricky (for this level) programming problem

From an operations management perspective, where we are interested in routing, distances, etc. the information that is the most interesting for us in the table are of course the coordinates.

Try to answer the following questions by writing the Python code for the data analysis:

- What is the average latitude and longitude of all these restaurants (the geographical "center of gravity point")
- How many restaurants are north of a user-inputted parallel?
- Take a lon-lat coordinate from the user (or for example, just use the newly found "center of gravity" point) and make four separate files, each containing the restaurants in a quadrant (NW, NE, SE, SW) defined by the given coordinate.