

Practical Manual for Data Analysis & Programming for Operations Management

Part A - Python

2022-09-04

Nick Szirbik, Wout van Wezel

University of Groningen, Department of Operations

Contents

1.	Introduction.....	3
2.	On the Basics of Programming.....	4
2.1	Starting with the first questions regarding the basics of programming.....	4
2.2	The Python language and some early syntax rules (for comments).....	4
2.4	Writing a small Python program using PyCharm, and executing it	6
2.5	Your next program	8
2.6	Compound statements and the role of indentation in Python's syntax	12
2.7	For the advanced: From “simple” variables to collections – and a bit about Objects/Classes in programming	13
3	On the basics of built-in Data Structures	15
3.1	Starting to analyze data sourced from external storage (csv files)	15
3.2	First, how is data stored in a csv file	15
3.3	Bring data records from a CSV file into a list-only based structure.....	15
3.4	Iterating through the data structure	17
3.5	Selecting records and data points from the table.....	19
3.6	Changing the content of the table and saving the new data into another file	20
3.7	For the advanced: solve a tricky (for this level) programming problem	20
4.	On the basics of Data Frames (using the pandas library)	22
4.1	Using the pandas to structure stored data and explore its content.....	22
4.2	What is importable code and what does it offer?	22
4.3	Create a DataFrame object directly in the memory	25
4.4	Manipulating DataFrame objects	27
4.5	Reading data from a file into a DataFrame object	28

4.6	Homework.....	29
4.7	For the very curious student only	30
5.	More about Data Frames.....	31
5.1.	Changing and manipulating pandas ’ data frame objects.....	31
5.2.	Why knowing pandas is important.....	31
5.3.	To start with a simple example.....	31
5.4.	Some simple statistics using data frames from pandas	35
5.5.	Graphic visualization methods of pandas data frame content	36
5.6.	Measure and visualize a real data file	37
5.7.	For the curious student.....	39
6.	Preparing “Raw” Data for Analysis.....	40
6.1.	The Data Science things that everybody does but no one really talks about: Cleaning/Filtering/Formatting.....	40
6.2.	Cleaning real-life data points: It is mostly a “String affair”	40
6.3.	Identifying the problems with the given data and making initial decision for the cleaning process 41	
6.4.	How to clean a field that has some string variety, but can have only simple (binary) values.....	43
6.5.	How to clean a data field that is needed in a computation, but it is a messy string	44
6.6.	More to do	44
6.7.	For the curious student.....	45
7.	Learning Good Programming Practice.....	46
7.1.	How can you fix your code?.....	46
7.2.	Every annoying coding problem has an explanation.....	46
7.3.	Debugging syntax errors	47
7.4.	Debugging runtime errors.....	48
7.5.	Debugging semantic errors	49
7.6.	You need also the right attitude	50
7.7.	DEMO error – how to find the cause of a difficult bug	51
8.	To conclude: how to continue from this point onwards, if you want to learn more Python and Data Science.....	56
8.1.	This could be not the end, this could be the beginning.....	56
8.2.	Online resources for a Python skills advanced skills learning path – that won’t cost a cent	56
8.3.	Online resources for a learning path in Data Science.....	57

1. Introduction

In the DAPOM course of the TOM Master Programme, you have to perform three activities per week for seven weeks: study and practice the weekly course material in the practical manuals, then attend a flipped-on (Q&A) plenary session, and finally, attend the weekly assisted practical session with your practical group. During the practical sessions, you will work in teams of two, assisted by instructors. Some of these lecturing and practical sessions are intended to teach you the basic skills of programming (by using Python as a programming tool).

This Manual is meant to be the support documentation for those parts of practical sessions which are mostly focusing on the learning of programming skills in Python. The other parts of the practical sessions will be focused either on programming-based modelling of OM problems, usage of solvers and optimization, or on specific data analysis techniques and tools (in our case, ElasticSearch). This manual can be studied gradually as the block progresses. Each chapter has 7 sections: first a list of topics to be covered, then explanations about specific technical issues related to the topics, and up to section *.5, next-to-screen material that will allow you to progress in writing and testing code. If you do not manage to finish the exercises in the allocated time of the practical, you will have to work individually throughout the week. Ask help from the instructor, but only if you really need to. The best way to learn programming is struggle alone. Avoid to work in bigger groups with other students, especially by mimicking what they do. Spar with your team mate, and avoid being a passive learner. Section *.6 of each chapter is for “homework”, i.e., the individual work planned to be done outside scheduled classes.

For those students who progress fast, and finish the first 6 sections before the time budgeted, section *.7 is an optional extra to explore. We wish that all students will be willing to invest time in this extra work proposed. **Try to keep ahead all weeks, do not fall behind**, because our previous experience with the programming learning curves is that once fallen behind, it is quite difficult to catch up. You will need all the skills you can acquire in order to make your individual assignments for the end grade, and also keep in synchronous pace with the work to be done with the rest of the work allocated to the practical sessions (for linear programming, ElasticSearch) of the 7 weeks.

Besides what we offer here, there is a plenty of material out there to study Python programming and Data Science. Look at the final chapter (8) of this manual for pointers in those directions.

We wish you success!

The instructors

2. On the Basics of Programming

2.1 Starting with the first questions regarding the basics of programming

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

- What are the **TOOLS** that we use (Python as a language, PyCharm as an IDE, libraries)
- What is the **SYNTAX** of a programming language (and in particular, Python's syntax)
- What is a built-in **FUNCTION** in Python
- How to **DISPLAY** simple output
- What is a **LITERAL** and how many types of literals we will use
- What is a Python **VARIABLE** and how to give a value to a variable (simple variables)
- How to **READ** values from the user input into variables
- What is the **TYPE** of a variable and how this can be changed dynamically
- What is a **COLLECTION** type variable (structured variables)
- What is a **LIST** in Python and how can it be constructed
- What is a **STATEMENT** and how Python code is executed
- What is the (very important) role of **INDENTATION** in Python
- What is a **COMPOUND** statement (an example of using the **with** statement)

2.2 The Python language and some early syntax rules (for comments)

Like all existing programming languages, Python has a **formal** syntax (i.e. the rules of building the **code**), which fully determines its **semantics** (i.e. what happens when the code is **executed**).

Like almost all programming languages, the Python code (i.e. programs) is formed by lines of text. There are two types of code text: **comments**, and **executable** lines (for which the computer environment for Python will have to do something). Commented lines are not executed. The role of the comments is to explain details of the code, for example to help other programmers to understand quickly your code, or to help you to understand your own code after a time when you did not use or changed it. Do not underestimate the usefulness of comments, and try to insert in your code as many as you have time for.

TO REMEMBER : a comment starts always with the **#** character. An alternative, typically used for multiple comment lines is to write **'''** (3 successive apostrophe characters) at the beginning and end of the commented text. Or **"""** (3 successive quote characters).

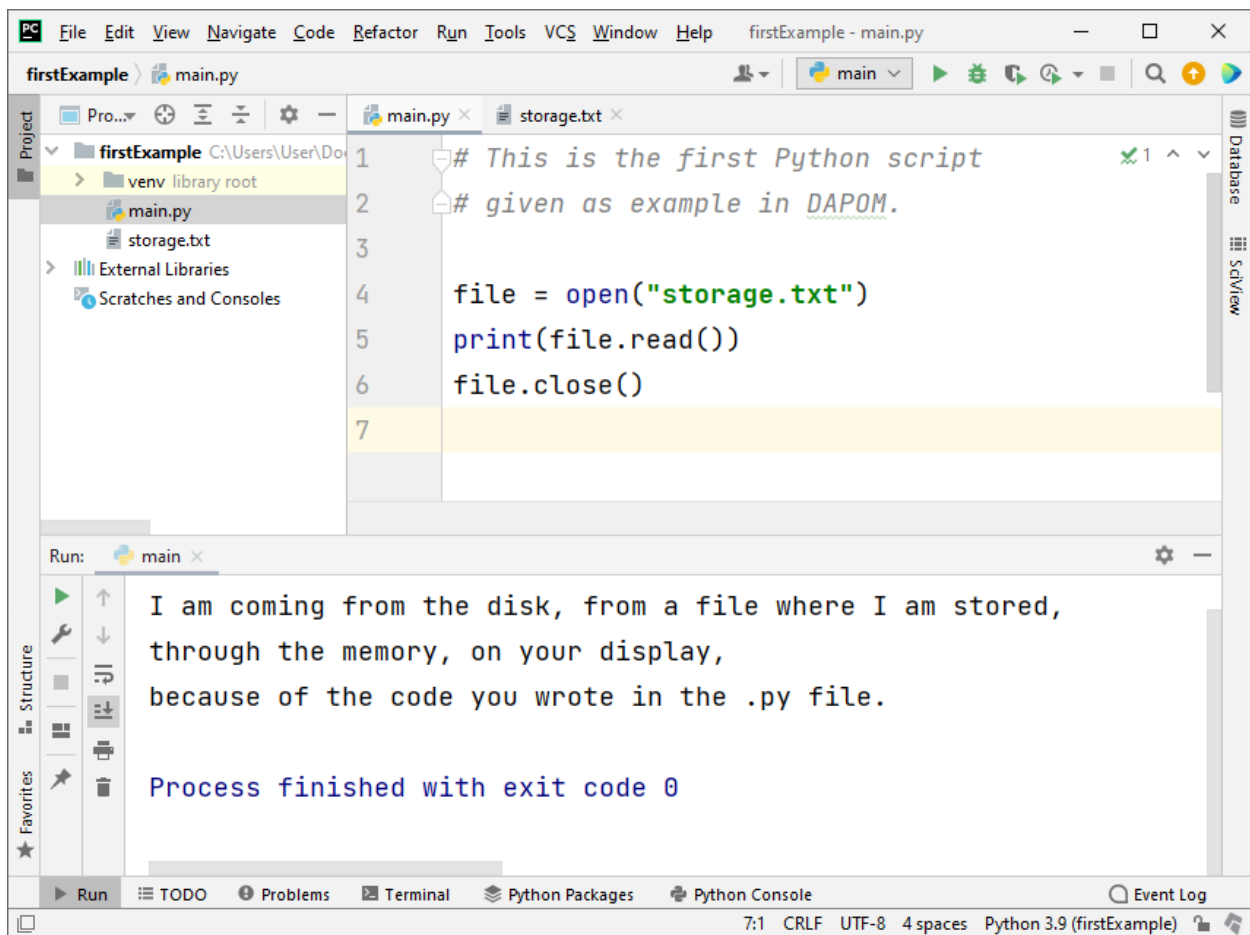
And a bit of history, for the curious student. Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands. **Fun fact**: Python is not named after the snake. It's named after the British TV show Monty Python. Python, like other languages, has gone through a number of versions. Python 0.9.0 was first released in 1991. In 2000, Python 2.0 was released. This version of Python included list comprehensions, a full garbage collector, and it supported Unicode. Python 3.0 was the next version and was released in December of 2008 (the latest stable version of Python is 3.9.10 as for February 2022, and there are beta releases for 3.10). Although Python 2

and 3 are similar there are subtle differences. Perhaps most noticeably is the way the **print** statement works, as in Python 3.0 the **print** statement has been replaced with a **print()** function.

The official full documentation of the language can be found at: <https://docs.python.org/3/>

2.3 Our Integrated Development Environment (or IDE), used to write and execute programs - PyCharm

A combination of a programming language interpreter (Python in this context), a script editor, and integrated output visualization (and many more features in fact) is named in the programmers' lingo an Integrated Development Environment, or IDE. An IDE can be "assembled" by experienced programmers, or can be acquired as a single software package. For Python programming, we could use simple IDEs like IDLE (a very basic and effective one), Thonny (developed for kids, it is simple but powerful), Spyder (used extensively in the Data Analysis community), Atom (used for its online advantages), Eclipse (used by professional programmers that use multiple programming languages), etc. Experienced programmers can switch easily from one IDE to another, because in essence they are all the same set of tools, but the look and feel can be slightly different. In Dapom we use PyCharm (the community edition). A screenshot of the PyCharm IDE where a programmer runs a simple program looks like below:



It is useful to have three panes open. One for editing the file you are currently working on (in the image above: the right-up one). One for your project (left-up corner), where you can see the files and subfolder in a project folder (which is named **firstExample** here). These two are by default opening when you create a new project. The third pane (below the other two) is where the executed programs will show their output. You can run your **main.py** script either by pressing on the green “play buttons” (triangles) or by the Run menu, or by the short cut key combination <Shift+F10>.

2.4 Writing a small Python program using PyCharm, and executing it

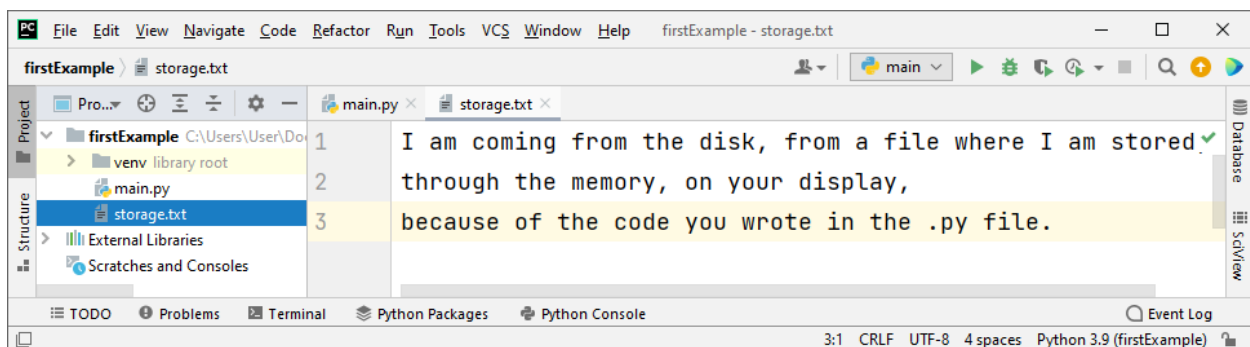
The right-up pane is named the *editor pane*, and offers some advanced features for programmers like syntax highlighting, on-demand automatic completion, code-cells, etc. We will slowly learn to use many of these. You can see at the top of the pane the name of the file you will edit (observe that Python files have the extension .py). You can have multiple files opened in PyCharm (some can be files of other types, like .txt, or .csv – however, you will be able to execute only .py files). When you create a new project in PyCharm, the IDE creates automatically a **main.py** file. Do not rename this file, it will be the central script of your projects. That is, all projects have a **main.py** file.

Delete the content which is automatically generated for the **main.py** file, then add the following executable code – exactly as written below. Do not try to understand it yet, albeit it is rather self-explanatory, and you already wrote something along this line in the comments. Do not copy paste from this document, write it by typing it:

```
file = open("storage.txt")
print(file.read())
file.close()
```

Create a new file (use the menus File/New file...), named **storage.txt** and write inside the following plain text (you can alter it a bit, it does not matter):

I am coming from the disk, from a file where I am stored, through the memory, on your display, because of the code you wrote in the .py file.

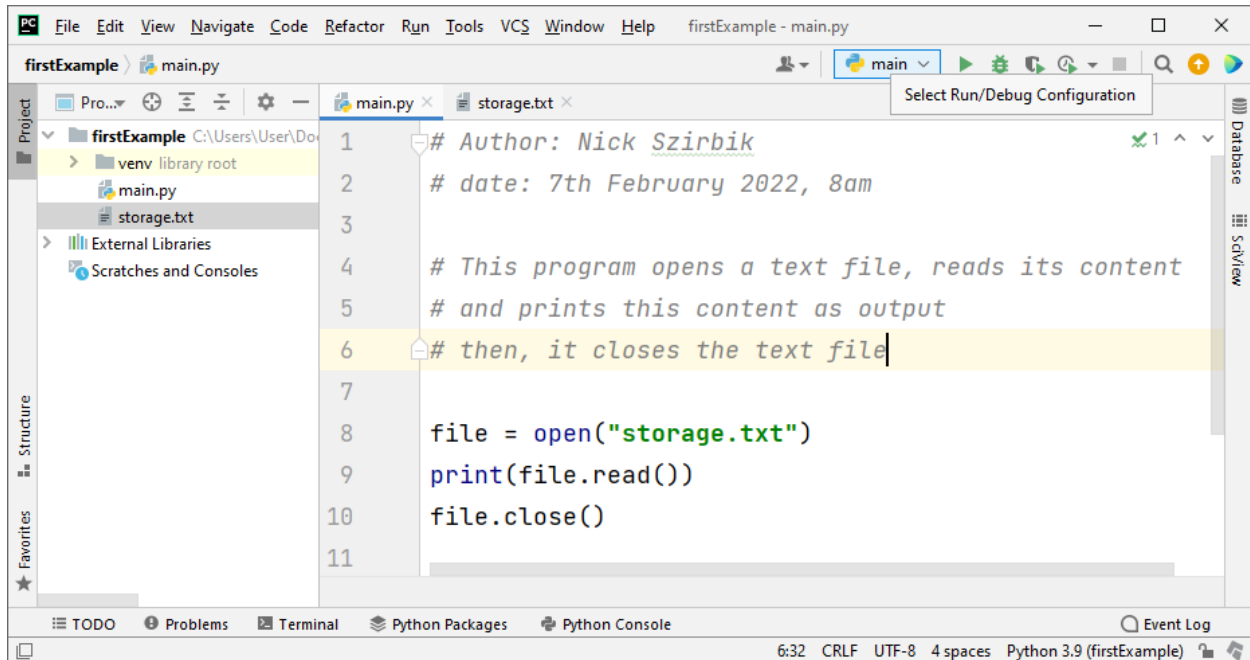


Save this file (the shortcut is Ctrl+s on Windows or Cmd+s on Mac). Close the editing tab of the **storage.txt** file. You can open it again by double clicking on the file name in the left pane.

In the **main.py** file, write your name as author, the date and hour of creating the file, and add a few comment lines where you explain that this program reads a text from a file on the disk and will display it on your screen (somewhere).

Make sure that the executable code lines are aligned to the left margin of the editing space.

Remember that nice looking Python program looks like below where the comments are longer than the executable code, and that is a good thing!



Execute the code by trying the various manners to execute. Do it multiple times. What do you think it happened? (btw, if you are a total beginner, you have now run your first Python program!) (cynical NOTE: if you copy/pasted, it fails for sure ☹).

The pane below where the content of the text file is displayed is the Output Area for the executed scripts. Your Python code is executed by a piece of software named **interpreter**, which is activated by PyCharm when you ask for running the program. Each line of code is executed separately. The interpreter ignores the comments in a program.

The executable code you wrote has a rigorous syntax. It contains a complex **variable** definition (an **object** like **file**), some function calls (like **read()**) and a **literal**.

After the line `file = open("storage.txt")`, insert the following line in your code:

```
print('The text file "storage" is now open')
```

and add the following line after the last line:

```
print('The text file "storage" is now close')
```

Execute the program again.

The code constructs `"storage.txt"`, `'The text file "storage" is now open'`, and `'The text file "storage" is now close'` are so-called **string** literals. These are values in your code that cannot change during the execution of your program. Literals can be also numbers, like, 256 (which is an **integer** literal, or **int** in programming jargon), or 3.141592 (which is **real** number literal, or **float** in programming jargon).

EXERCISE: display in the console the numbers 256 and 3.141592. Use only literals.

2.5 Your next program

Create a new project folder (via File/New Project...) and name it for example **mySecondPythonApplication**. Good practice dictates that you write again in `main.py` a program header like above with new comments, indicating that this program will do some simple arithmetic.

Literals are not sufficient for programming. To allow that the program takes repeatedly different values from its environment (e.g. files on external storage like disk, or from the user via a keyboard, or from sensors, etc.) we need **variables** (like **file** in the previous example).

Variables in programming have a **name**, a **type**, and a **value** (which can change). They also exist as zones in the memory of the computer, where their value is stored temporarily – when the computer is switched off, their values are lost. This is why the information is kept in files on external memory storage, like HD disks or SSD disks, or the cloud. The main memory (which is directly accessed by the processor) of a computer is transient and never used for long term data storage. Its main feature is speed of access to data residing in this memory, and it is orders of magnitude faster than for external storage access.

When the Python interpreter executes a line (in programming jargon, “statement”) like:

```
two_Power_16 = 256
```

it does three things:

1. It creates a new variable, with the name **two_Power_16**
2. Decides that the type is integer (or **int**), based on the value of the literal given, and **allocates** enough space in the memory for an integer value
3. Assigns to the variable the value on the left side of the “=” (which is called the **assignment** operator in programming jargon)

EXERCISE: create two variables, one with the value 512 and the name **two_Power_17**, the second with the name **pi_6decimals** and the value 3.141592. Display the names and the values of these two variables in the console, to obtain the following output:

```
2^17 is 512
```

```
Pi with 6 decimals is 3.141512
```

Hint: **print()** is a function that can have multiple parameters, literals and variable names, separated by commas.

The variable names (in programming jargon we call them **identifiers**, and this applies to any object defined by you in Python) have to adhere to strict rules. They have to start with an alphabetic character (or an underscore `_`), and after that, only letters, digits (0-9) and underscores are allowed.

One of the typical tasks of a computer program is take inputted values (from various sources, like keyboard, sensors, files, cloud) into variables, and compute results based on mathematical procedures which are expressed as executable code. The simplest mathematical procedure is the mathematical expression, which is the coded version of a formula. As you already learned, the mathematical formula to compute the solutions of the quadratic equation is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$D = b^2 - 4ac$$

D > 0, Two real solutions

D = 0, One real solution

D < 0, Complex Solutions

This procedure can be easily transformed into Python code:

```
d = b**2 - 4*a*c
square_root_of_d = d**0.5
sol1 = (-b - square_root_of_d)/(2*a)
sol2 = (-b + square_root_of_d)/(2*a)
```

EXERCISE: in the current program, add code that assigns the values 1, 5, and 6, to a, b, and c respectively, and add the code above (again, do not cut and paste, write it yourself), and display the results in a user friendly way (by using string literals in the print statements).

The full list of mathematical operators in Python is given in the table below. Assume that variable **a** holds the value 10 and variable **b** holds the value 20.

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$, $-11 // 3 = -4$, $-11.0 // 3 = -4.0$

In accordance to mathematical notation, the exponent, multiplication, division, floor division, and modulus operations are executed **before** the addition and subtraction – from left to right. If the formula evaluation necessitates another order of execution, parentheses are needed.

The parentheses always dictate the order in which the operators are executed. This is why it is necessary to write $/(2*a)$, instead of $/2*a$ – because otherwise the multiplication with variable **a** will occur after the division. The evaluation of any expression with parentheses starts from the innermost parentheses and continues outwards.

If we want to run this program with different input values for this numerical procedure, we have to edit the program itself (i.e. the lines where **a**, **b**, and **c** are assigned with literal values). However, this is not the appropriate way to assign input values to variables – because it is very cumbersome and error-prone to edit again and again a program before it runs. If we want to input numbers from the keyboard when a Python program already runs, we can use the **input()** function:

```
a = input("Enter the value of a: ")
```

This statement displays first in the Console the text given as parameter (this is called in jargon a “prompt message”), reads from the keyboard the digit characters introduced by the user (displaying

them in the console), and when the user hits Enter, it transforms these characters into the value to be stored in the memory location of variable **a**.

The problem here is that the programmer cannot be sure that the user of the program will always input a number (necessary for example if **a** is the first parameter of the quadratic equation). To force the user to input a specific type of value, we can force this via a number of functions:

```
a = float(input("Enter the value of a as a real number: "))
i = int(input("Enter the value of i as an integer number: "))
j = complex(input("Enter the value of j as an complex number: "))
boo = bool(input("Enter the value of boo as True or False): "))
t = str(input()) # any text can be entered, actually...
```

Beside integers, real (i.e. float) numbers, and strings of characters, you notice that we can have in Python complex numbers (you may not have the mathematical background to understand them, but remember that they exist), and also Boolean variables, as in many other programming languages, which can take only two values: **True** or **False**. These variables are useful for logic operations and expressions, which are very important for programming, and will be studied next week.

EXERCISE: First, comment out the lines of code that assign the variables **a**, **b**, and **c** to the values edited as code literals. Replace the assignment of values for **a**, **b**, and **c** with value assignment statements using the **input()** function instead, making sure that the user (in this case you) is forced to input numerical values when the program runs (Hydrogen pop-ups for you each time a data entry mini-window). Use prompt messages for each variable, and run the program.

You will probably get an error if you did not force the result of the input to a numerical type (Python will consider that the numbers you input are strings). If so correct this error.

Introduce various numerical values, trying to get complex roots as results (e.g. a is 1, b is 2, c is 3). How is the result printed in this case? Try to input characters that will not express a number. What is happening?

You notice that we use more and more built-in functions. The Python 3.* language version has a number of functions that are always available. The list for the current version is given below:

		Built-in Functions		
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

You already have used a few from this list, like `open()`. You do not have to remember them all, and you do not have to use many of these in our course. For references and explanation, use:

<https://docs.python.org/3/library/functions.html>

2.6 Compound statements and the role of indentation in Python's syntax

The first program you run was opening a file from the disk, reading its content, and printing this content (a string) in the console, explaining also to the user what is happening. It makes sense to write it differently, using a string variable to keep the content of the file in the memory:

```
file_handler = open("storage.txt")
file_content = str(file_handler.read())
file_handler.close()
```

It is also considered good programming practice to make the variable identifiers as self-explanatory as possible. For example, above, the first variable is a complex object variable (try to display it and make sense of what you see) with a specific purpose (therefore the “handler” in the name) and the second is in fact a string variable that will store in memory the *content* of the file as a succession of characters.

Python practice also recommends to manage input and output files in a different manner, by using a statement that contains the `with` keyword. If we use it, the lines of code above will become:

```
with open("storage.txt") as file_handler:
    file_content = str(file_handler.read())
# the file is now closed automatically
```

The Python executable code above is actually a single statement – to be executed by the interpreter as a batch, even if it goes on multiple lines – these are called **compound or multi-line** statements. To indicate to the Python interpreter that we want a batch execution of multiple lines, the first line has to be defined by a compound statement keyword (like **with**, but also **if** and **for** allow for this, as you will learn later). The next lines that are in the batch should be indented with a <TAB>. The first line is typically finishing with the “colon” character ‘:’. A compound statement ends when the next executable line after the last is not indented.

The main advantage of the **with** statement over the first given code is that the file will be automatically closed after the compound statement ends, and only the statements that are “nested” inside the **with** compound can manage the file. This is not always desirable for specific tasks (you will learn about this later), but it is considered robust programming practice, especially for beginners.

EXERCISE: change the previous file content reading code with code that uses **with**.

2.7 For the advanced: From “simple” variables to collections – and a bit about Objects/Classes in programming

All variables in Python are actually “objects”. This term is related to a programming paradigm called Object-Oriented Programming (or OOP), where Python belongs, albeit it is not absolutely necessary for the programmers to use explicitly the OOP features of the language. Nevertheless, you have already used a “method” (i.e. typical OOP jargon) when you invoked the function **file_handler.read()**. All objects are instances of “classes”. For example, if you want to know the name of the class of an object, you can invoke the function **type()**, like for example:

```
print(type(file_content))  # this visualizes the type, or class of
                           # the variable file_content
print(type(file_handler))  # this visualizes the type, or class of
                           # the variable file_handler, which is a
                           # very complex object
```

Now, we will only shortly introduce the **list** type, which allows the declaration of variables that are collection of more than one value. Below, an example of a declaration of a list variable that are composed of 12 values that are all strings – given as literals.

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',      # Dutch names
               'Juli',    'Augustus', 'September', # for the months
               'Oktober', 'November', 'December']  # of the year
```

However, a list can be formed by values with different types. We can explicitly construct a list differently from above, by using the built-in function **list()**, and using already declared variables with assigned values, like:

```
born = int(1963)
weighs = float(94.3)
personal_data = list(("Nick", born, weighs))
print(personal_data)
```

Observe that the distinctive syntax feature of a list is the use of square brackets to delimit its content. However, when created with the help of the built-in function `list()`, then the parameter of the function has to be a so-called tuple, delimited by parentheses. It is possible to declare initially an empty list, which is still an object of the type (or as we say in OOP - class) `list`. Try the following code:

```
to_grow = []
print(type(to_grow))
print(to_grow)
```

A class is an abstract concepts that combines some data and the operations on that data in a single “capsule”. These operations are the “class’ methods”, and in Python they are just like any functions. The most typical list method/function is `append()`, which can have a single parameter. If we have already a declared object of the `list` class (like `to_grow`), we can use this to add new elements to the `list` class object (which is initially empty). Try the following code, added to the code cell above:

```
to_grow.append("first element added")
to_grow.append(int(2))
to_grow.append("IIIrd element added")
print(to_grow)
```

Notice the syntax of the notation using the dot (`'.'`) character *between* the name of the object and the name of the method/function. Some of these functions will give back a value, and some, like `append()` will give back nothing (actually, a special value, called a **signal value**, and named explicitly `None` will be “returned”, which is the jargon for “give back”). Try the following code, added to the code cell above:

```
x = to_grow.__class__ # is this a function call?
print("the attribute __class__ of the 'to_grow' object is:", x)
y = to_grow.append("4th element added")
```

3 On the basics of built-in Data Structures

3.1 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 (character, or more typically **c**omma, **s**eparated **v**alues) file is a fairly simple text file containing structured information. A special character, typically a comma or <TAB> character, separates the values that are stored. 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 OS 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 (do not visualize it Excel! it will add content that will make its reading in Python more difficult). Each line (row) of this file – except the first line - is a “data record”, where individual commas separate “data points”. It is important to notice that a comma does not follow the last data point on any line.

The first line contains the names giving the meaning of the data points on the following lines – 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 Brightspace.

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 easiest way to avoid complicated paths for data files to be used, is to operate these files from the “project folder” that is available via working in PyCharm. In this way, they are in the same directory with the Python scripts that access them.

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

```
print("data records =", len(table)) #run your program now
```

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 through 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 preferable 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, should 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 practical): 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.

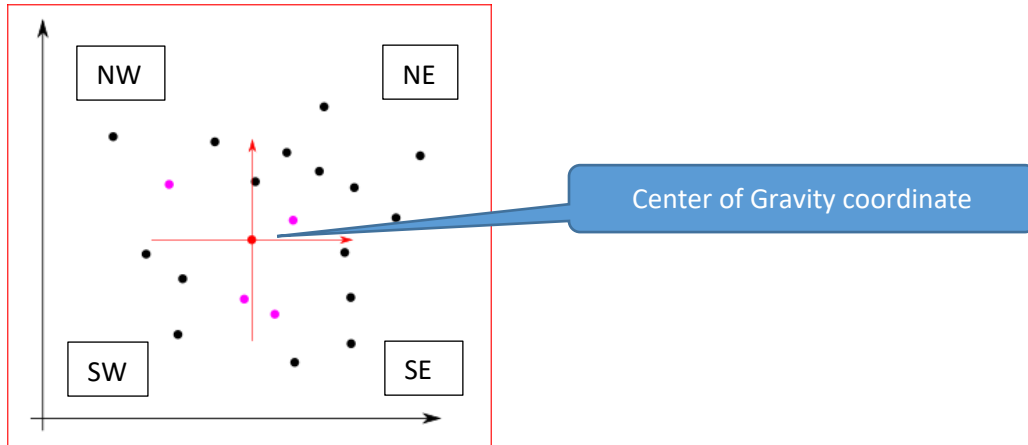
HINT: better use a Python function (find out on the WWW how to define and use a function). It is not necessary to use a function, but the code will look much better, and will be easier to manage.

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.



4. On the basics of Data Frames (using the **pandas** library)

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 be 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 PyCharm, 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 on 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 create 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 illustrates 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
]
}
```

HINT: you can copy/paste this in your PyCharm 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 of 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 PyCharm (or simply with 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 PyCharm'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”, “cet”, 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>

5. 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 Brightspace, ‘**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=bjvlp1-1w84>

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_object> = <object>.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 PyCharm 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 Brightspace), 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/>):

$$\text{score} = (g / (g + b + u) - b / (g + b + u)) * 5.0 + 5.0$$

...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 of 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 PyCharm to visualize it!), 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 methods of pandas 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 Brightspace, 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** or the **folium** 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.

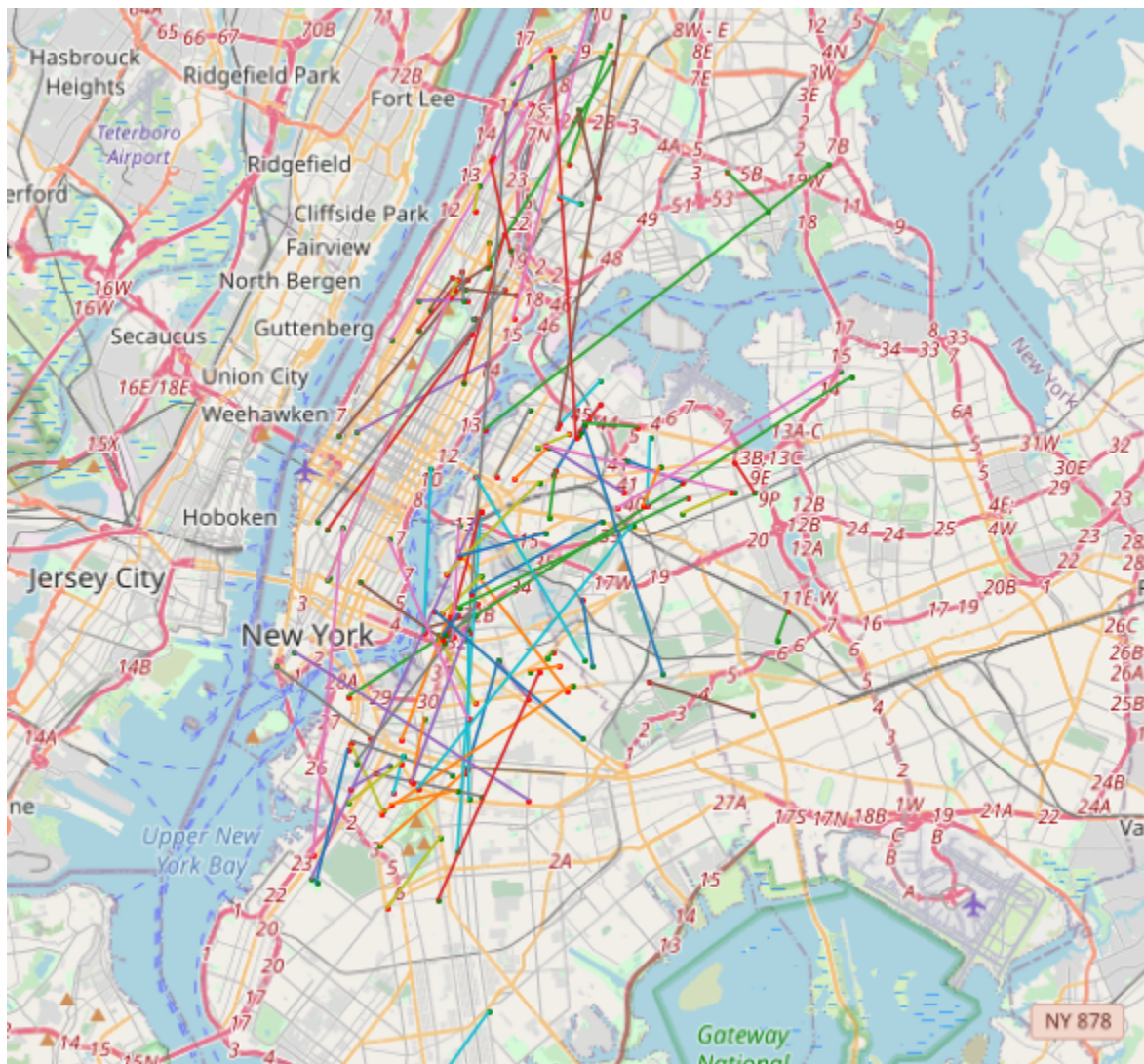
HINT: to show a line on the map, use the plotting statement:

```
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 browser (you cannot see directly the output immediately when running the program in Python, because PyCharm does not visualize html directly – in PyCharm you have a menu that allows to open a browser of your choice with the local map showing file).

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.

6. Preparing “Raw” Data for Analysis

6.1. The Data Science things that everybody does but no one really talks about: Cleaning/Filtering/Formatting

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

- How to identify the “dirty” (problematic) areas of a data repository
- How to identify the magnitude of the problem
- How to decide about what needs to be done
- A few tricks of how to “clean/fix” the fields that need “repair/small changes”
- Why this part of Data Science is important

6.2. Cleaning real-life data points: It is mostly a “String affair”

Like Cinderella for outsiders in the fairy tale, maybe the most important things are not so obvious in the Data Science “household”. However, without her, the bad-mother’s household will collapse. Similarly, data cleaning can make or break a Data Science project. Indeed, the professional data scientists spend a very large portion of their time on this task (sometimes 90% - and btw, do not worry, we will give you a clean data repository for your assignment – however, out in the real world this would be completely unrealistic).

There is a simple reason for this: collecting the data is always a messy and error prone process. One of the wisdom sayings in Data Science is that “Better data beats fancier algorithms”. There is also a very old adage in Computer Science: “Garbage in... garbage out”. If one has a clean data repository, many more tools can be applied to analyze it, and also simpler algorithms need to be devised for further analysis. In this chapter/practical, we will focus on some of the cleaning operations – that will use mostly the Python string class methods. But before that, we will shortly summarize the other previous steps that are important in preparing data for analysis.

The first step is to remove unwanted data points. For example, typical unwanted data are the **duplicates**, which can result when various separate datasets are combined. Duplicates can refer to rows in the table, but also to columns. Another example of unwanted data is the **irrelevant** data, which do not contribute to our analysis. Or, it can be that some entries are just unlikely to be real data (**outliers** or out-of-range data). If these can be identified, they should be eliminated (however, one has to have a good reason to remove an outlier).

The second step is to handle **missing** data. One may have records or columns that contain too often the **NaN** or **None** values. Two decisions can be taken: to **drop** the record or column altogether, or to input values that are **inferred** based on the distribution of the other existing values.

Because this is important, in the exercise below, we will discuss about such an inference, related to one specific data value.

However, the bulk of the work in this practical will be about changing strings of the same data field into a uniform format. You have to download first from Brightspace a .py generator file (**fifthWeek_generate_data.py**) and run it; that will generate the .csv file that is named: **phdThesesFranceThirteesToNineties.csv**. You notice that this is a relatively big file. Our newly created data repository has a “**neat structure**”, that is, we have a .csv file that has **all its records of the same length** and encoding, and it is (mostly) complete. That means that we can directly read the data into a data frame object in the memory. Before we do that, for the sake of just showing what happens, try to read this file into Excel, by using the import facility via the menu entry **Data/From Text**, indicating the comma as separator. You will have an Excel sheet with the file’s records as rows. Observe the columns and try to understand their meaning (as usual for a .csv file you have the column names in the first row of the file/table). Observe the total number of rows/records. Is this the total number? (check with Notepad if Excel coped with all the rows in the file – Notepad also indicates the line number).

NOTE: as you probably can immediately notice, the data is generated by using the **Faker** class code, presented in the previous chapter. You can study later the fake data generation code in **fifthWeek_generate_data.py** and play with it, changing it to create other kinds of fake test data.

Any data analysis exercise has a **purpose**, based on a research goal, business problem or opportunity, human curiosity or boredom, etc. Let us assume that our purpose here is to determine, based on the data available:

1. *What is the average age of when a PhD candidate defended its thesis?*
2. *What is the average age of female candidates, and also what is the average age of male candidates?*

Let’s proceed!

6.3. Identifying the problems with the given data and making initial decision for the cleaning process

First, the file’s content has to be read into a data frame object. Start a .py file named **fifthWeekEx1** (for example). Write the code that displays the shape, the columns, and the data types of the data frame. Run the code. This will enable you to see the (true) size and structure of data (number of rows/records, number of columns, and the name and types of the columns). To answer the analysis questions above, we need data only from three of these columns:

- the gender, and

- the year of birth, and
- the year of the PhD thesis defense.

From the data types of the column, you can see that the year of defense is an integer number, which can be used directly in the analysis. For example we can explore the range of this year of defense (as usual, the chosen name of the data frame object is **df**):

```
print('year range:',
      df['year_of_defense'].min(),
      '-',
      df['year_of_defense'].max())
```

Let us explore the gender column. When looking into the data with Notepad or its similar tools on Mac or Linux (this is the so-called preliminary visual inspection) one can immediately notice that there is a variation of the ways this field was completed. It could be possible that such a file was collected from various universities, which each decided to use a specific coding for the gender. Because there are so many rows, we have to find out first how many ways of coding the gender as a string are in this whole data repository. Before we do that, we should sort the data frame by the values of the gender column:

```
odf = df.sort_values(by = ['gender'])
```

To identify the variety of the gender strings used, we can use two data frame class methods: one that counts how many kinds of representation are (a method named **.nunique()**) and another that gives the counts for each individual kind of representation (a method named **.value_counts()**):

```
print('ways the gender field appears:',
      df['gender'].nunique(), 'kinds\n',
      'each occurs:\n kind    occurrences\n',
      df['gender'].value_counts())
```

We can see now how many kinds of gender strings exists throughout the data, and how many of each.

For the date of birth, do the same thing as above, using the same methods for the respective column in the data-frame. You will notice that the variety of dates of birth is much higher than the gender (almost two degrees of magnitude). A wise decision is to start with the easiest task. Therefore, first we will clean the gender column (follow the guiding text in 6.4), and only after the year of birth column (explained succinctly in 6.5). To finish coding this first .py file, write code that saves the data frame object ordered by gender data into a new. csv file, without the index (index = False, if you remember this from the previous practical). Run the code and remember the name of this output .csv file.

6.4. How to clean a field that has some string variety, but can have only simple (binary) values

Start a new .py file, named for example **fifthWeekEx2**. To keep file sizes manageable, the code in this file will repair the gender field only. First, read the csv file you just saved previously into a data frame object. This cleaning operation is rather simple because the gender field has a **binary**-like value. The column for gender will contain either the string “Female”, either the string “Male” (in more modern times that the file contains data about, maybe more gender types are needed).

By looking at the possible values that are now in the column, we notice that there is a “?” value. That clearly says nothing useful about the gender. Because the number of these occurrence is rather low compared with the total number of records, for the time being we will ignore these rows (we come back to it for homework). To drop these few rows now is a typical decision for the Data Science process.

The rest of values can clearly indicate the gender. To replace the strings, we will use the **.apply()** method, like in the previous practical. This method needs always a function (either a defined one, either an anonymous **lambda** function). In the .py file, before the **read_csv()** containing line (as a good structured code should look like), you have to define a function named for example, **repairGender()**, that takes as argument a string named **gender**, and returns a string named **repairedGender**. The repair in this function, can be done by the following code:

```
repairedGender = gender
# if the string is not repaired after the code below,
# the same value that was sent is returned
if gender == 'f' or gender == 'female' or gender == 'F' or gender == 'fem.':
    repairedGender = 'Female'
if gender == 'm' or gender == 'male' or gender == 'M' or gender == 'man':
    repairedGender = 'Male'
```

In the “main” part of the code (after you have to code that reads the csv file), this function can be used as:

```
df['gender'] = df['gender'].apply(repairGender)
```

Which will replace all the strings in the column with the repaired strings.

QUESTION: what happens to the rows that have the '?' value for the gender field. Check.

Write code that saves the data frame into an indexless .csv file. Run the code and remember the name of this second csv file you created from the original one. For more string methods, see appendix.

HOMEWORK 1: the code above is rather clumsy and too big for the task. A good Python programmer could write a much more compact and elegant solution. Try to find a smaller solution. **HINT:** use the **title()** string method. Make a separate .py file for this solution.

6.5. How to clean a data field that is needed in a computation, but it is a messy string

By visually inspecting the year-of-birth field, you will notice that actually there are only a few characters that have to be cleaned around the year(s): ['c', 'C', '.', '(', ')', '-']

Start a new .py file, named for example **fifthWeekEx3**. The code in this file will repair the gender field only. First, read the file you just saved previously into a data frame object. Define a new function, named for example **repairYearOfBirth()**, very similar in structure with the previous function, which will return a repaired string representing the year of birth in a simple format, like **'1909'** – which can be easily converted into an integer usable in the age calculations. To clean unwanted characters, you can for example use the **replace()** string method, for example like this:

```
repairedYear = repairedYear.replace('c.', '')
```

That replaces the whole string **'c.'** with an empty string in the **repairedYear** string. Write code that replaces all the unwanted characters, except the **'-'**. This appears in the middle of a string in between two substring values that define a year range (the precise year of birth is not known). You can detect the dash (minus) character with the following statement:

```
if '-' in repairedYear:
```

...and use the **split()** string method to get the two strings that represent the year range. Compute the average year value (use **int()** and **round()**), and make sure that the final value is a string. Finish the defined function by returning the repaired year value – as a string.

In the main part of the program, use **apply()** to change the defective year of birth strings with the repaired ones. Write code that saves the data frame into an indexless csv file. Run the code and remember the name of this third file you created from the original one.

6.6. More to do

Start a new .py file, named for example **fifthWeekExercise3**. The code in this file will read the third csv file you saved previously and compute the answers to the two (actually three) questions for the intended data analysis:

1. *What is the average age of when a PhD candidate defended its thesis?*
2. *What is the average age of female candidates, and also what is the average age of male candidates?*

Finally, we have to address the ignored rows (those with '?' in the gender field). Given the amount of records and the relatively small number of these rows, the average values will not be really affected, therefore to ignore them is a rather sound Data Science cleaning decision in this particular case.

However, it may be that the number of these rows is higher - you can change the appropriate parameter in the faker code in `fifthWeek_generate_data.py` and have more of these. We could complete the field by hand, in Notepad (after the sorting by the gender value, these rows are nicely grouped together). For a few hundred entries, we could do this manually because we have a pretty good idea which first names denote a French female or male name – and it would take maybe a couple of hours. Naturally, this would be more difficult for a European if the names would be from the Indian Subcontinent, China, Japan, or Asia in general. Also, if the number of rows is in thousands, the manual task becomes untenable.

6.7. For the curious student

Automatic gender recognition is possible to some extent. You may study this further, for example one current project in the ever growing Python modules development ecosystems is:

<https://pypi.org/project/gender-guesser/>

You could attempt a third homework after this practical, intended for the ambitious students, to use the `Detector()` class developed by Jorg Michael (described for use on the webpage linked above) and automatically complete the gender field for the those rows where we can rely only on the name of the thesis author.

IN MORE DEPTH: We have written a paper in the past, based on our experience on a specific data aggregation process related to a healthcare research project in London. Here, the building of the data repository for the purpose of analysis is theorized into six steps. The paper is well cited and other researchers built their new ideas on our proposed aggregation process. As a student, you have free access to this paper, published in the Journal of Medical Informatics:

<https://www.sciencedirect.com/science/article/pii/S1386505606000864>

7. Learning Good Programming Practice

7.1. How can you fix your code?

The content of this practical is to be studied and executed in your individual study time. By reading the next sections and exercising with the proposed code, you will learn:

- What is “debugging”?
- What kind of software bugs exist?
- How to use the debugging techniques and guidelines
- Why debugging is so important and how to mentally approach this process

7.2. Every annoying coding problem has an explanation

Debugging software is similar for physical engineering troubleshooting (like identifying the problems with a malfunctioning new machine prototype) and it is an integral part of the entire software development cycle. Typical software “malfunctions” are: your code may not run at all, it crashes unexpectedly before its normal ending run, or it gives you incorrect results. The explanation is ALWAYS the same (thus, no magic): you have unwittingly included software errors (or more endearingly put, bugs) in your code.

That is nothing to be ashamed of, everybody can and will make mistakes. An old wise saying is “Errare humanum est, sed perseverare diabolicum”¹, and translates to: “To err is human, but to persist in error (out of pride) is diabolical”. Or, just plain stupid.

Debugging is indeed the tenuous but necessarily persistent process of finding the errors, and taking measurable steps to fix it, and lastly test and release the “error free” code². Software errors are typically classified (in most programming languages, not only Python) in three kinds:

- **Syntax errors** are reported by the Python interpreter when translating your code lines into executable code. They usually indicate that there is something wrong with the syntax of that particular line; an example is forgetting the colon (:) at the end of a **def** function definition; then the IDE stops the execution of your program and displays the message **SyntaxError: invalid syntax** in the output, indicating also the place where the error occurred (the line and column). Sometimes, the place is misleading, because the syntax

¹ Attributed to Seneca

² Wise programmers say that error free code does never exist, and it is foolish and dangerous to assume that a complicated piece of code is ever completely error free. Small pieces of code can be mathematically checked (like proving a theorem), but checking fully complex applications is beyond our current computational means. Moreover, most of the code out there interacts with humans, and that makes software behavior even more difficult to predict.

violation occurred on a line earlier, but the error is detected only when an unexpected (at least, for the interpreter) line starts.

- **Runtime errors** are reported if something goes wrong while the program is already running. Most runtime error messages include information about where the error occurred and what code line or functions were executing at that moment. For example, if you try to divide a value by 0, the IDE stop the execution and displays in the console the error message **ZeroDivisionError: division by zero**. Moreover, it shows the line that generated the error and its number in your code.
- **Semantic errors** are those with code that runs to the end but does not do what you expect. A typical example are mathematical expressions where you have incorrectly placed the parentheses and these are obviously not evaluated in the order you (mistakenly) expected, yielding an incorrect result for the problem to be solved by the code.

The first step in debugging is to figure out which kind of error you are dealing with. Once you identified what kind of error occurred, the next step is to find out where the error occurred.

7.3. Debugging syntax errors

For syntax errors, it should be simple to locate the error, because the system indicates a line that generated the error. However, sometimes it is not so easy, because the IDE shows you where Python finally noticed a problem, which is not necessarily where the real syntax error is. Many times the error is prior to the location of the error message, often on the preceding line (remember when you forgot a parenthesis?).

Here are some useful **guidelines** to avoid syntax errors:

- If you are building your program incrementally, running it after each statement you code, you should have a good idea about where the syntax error is. It will be in the last line you added.
- If you are copying/pasting code from a digital text, start by checking every character (the wrong type of apostrophes are typical culprits here). At the same time, remember that the source text might be plain wrong, so if you see something that looks like a syntax error, it might well be.
- Make sure you are not using a Python keyword (like **for**) for a variable name.
- Check that you have a colon at the end of the header of every compound statement, including **for**, **while**, **if**, **else**, **elif**, and **def** lines.
- Check that indentation is consistent. You may indent with either spaces or tabs but it is best not to mix them. Each level should be nested the same amount.
- Make sure that any strings in the code have matching quotation marks.
- If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!

- An unclosed bracket matching an open one, like (, {, or [, makes Python continue with the next line as part of the current statement. Generally, the syntax error occurs almost immediately in the next line.
- Check for the classic error of using assignment, i.e. = instead of equality comparison == inside a conditional expression.

7.4. Debugging runtime errors

For **runtime errors**, a first classic is that your program does not even start. This problem is mostly caused by the fact that your .py file consists only of functions (and/or classes) and does not actually invoke anything to start its execution. This may be intentional if you only plan to import this module to supply classes and functions to other .py files. If it is not intentional, make sure that you are invoking a function to start its execution, or execute one from the interactive prompt in the console.

The second kind of runtime error is when your program hangs. Programmers use the term hanging when the code starts but it seemingly stops at some moment and nothing happens anymore (but it is still running – in the IDE you can force the program to stop). This means that the program is caught in an infinite loop, typically caused by a **while** statement. How to fix this?

- If there is a particular while loop that you suspect is the problem, add a print statement immediately before the loop that says entering the loop and another immediately after that says exiting the loop.
- Run the program. If you get the first message and not the second, you've got an infinite loop.
- If you think you have an infinite loop and you think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition and the value of the condition. For example:

```
while x > 0 and y < 0:
    ... various statements in the while
    ... you do something to x
    ... you do something to y
    print("x: ", x)
    print("y: ", y)
    print("condition: ", (x > 0 and y < 0))
```

- Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be False. If the loop keeps going, that means that condition remains always True. You will be able to see the values of x and y, and you might figure out why they are not being updated correctly.
- Of course, all programming and debugging require that you have a good mental model of what the algorithm ought to be doing: if you don't understand what ought to happen to x and y, printing or inspecting its value is of little use. In this case, probably the best place to debug the

code is away from your computer, working on your understanding of what should be happening.

Another runtime problem is when the Python environment “throws” an exception (and the program stops). In this situation, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a “traceback”.

The traceback identifies the function that was currently running, and then the function that invoked it, and then the function that invoked that, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line numbers in your files where each of these calls occurs. The first step is to examine the last place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:

- **NameError**: You are trying to use a variable that doesn’t exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.
- **TypeError**: There are several possible causes:
 - You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.
 - There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.
 - You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is self. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.
- **KeyError**: You are trying to access an element of a dictionary using a key value that the dictionary does not contain.
- **AttributeError**: You are trying to access an attribute or method that does not exist.
- **IndexError**: The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a print statement to display the value of the index and the length of the sequence. Is the sequence the right size? Is the index the right value?

7.5. Debugging semantic errors

Finally, to catch a **semantic error** is rather more difficult (and the most challenging and stressful for the programmer). This is because the compiler and the runtime system provide no useful information about what is wrong. Only you know what the program is supposed to do, and only you know that it is not doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes that hard is that computers run so fast. You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed **print** statements is often short compared to setting up the debugger, inserting and removing breakpoints by adding **input()** statements, and walking the program to where the error is occurring. When your program does not do what you want, you should ask yourself these questions:

- Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
- Is something happening that should not? Find code in your program that performs that function and see if it is executing when it shouldn't.
- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other Python modules. Read the documentation for the functions/modules you invoke. Try them out by writing simple test cases and checking the results.

In order to program, you need to have a mental model of how programs work. If you write a program that does not do what you expect, very often the problem is not in the program; it is in your mental model of the program.

The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

This also comes down to your programming practice. To isolate any error easily, the best programming technique resides actually in the way the program you develop is structured. If the code is broken down into small separate modules and functions, identifying which part of the code yields wrong results is quite straightforward.

7.6. You need also the right attitude

Sometimes you can get really stuck. First, try getting away from the computer for a few minutes. Debugging requires a high mental concentration and it can affect the brain in unwanted ways, causing unwelcomed (and sometimes strange) effects:

- Frustration and/or rage.
- Magical thinking: the computer “has a will of its own”, or even “the computer hates me”.
- Random-walk programming: meandering aimlessly in finding a “silver bullet” solution on stackoverflow.com or trying to write another code that replaces the erroneous one – without understanding what went wrong in the first place.

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes some time to find out the cause of an annoyingly persistent bug. We often find the cause of bugs when we are away from the computer and let our minds wander. Some of the best places to discover the causes of bugs are trains trips, showers, and in bed, just before you fall asleep or when you wake up.

The most important thing is not to give up too easily. Programming is in the end an attitude of doggedness and persistence. You need lots of patience and endurance to become an effective programmer.

Nevertheless, it happens that after the best efforts, the cause of a bug is still not found. Even the best programmers occasionally get stuck badly. Sometimes you work on a program so long that you cannot see the cause of the error, even if it is plain to see. Then, a fresh pair of eyes and an unbiased mind of a fellow programmer is just what you need.

However, before you bring someone else in, make sure you have exhausted the techniques described before. Your program should be as modular as possible (with modules and functions as simple as possible), and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely. When you bring someone in to help, be sure to give them the information they need:

- If there is an error message, what is it and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

Good instructors and helpers will also do something that should not offend you: they won't believe when you tell them "I'm sure all the input statements are working just fine, and that I've set up the variable values correctly!" They will want to validate and check things for themselves. After all, your program has a bug. Your understanding and inspection of the code have not found it yet. So you should expect to have your assumptions challenged. And as you gain skills and help others, you will need to do the same for them.

Finally, when you find the cause of a time consuming bug, take a minute to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.

Remember, your goal as programmer is not just to make the currently developed program work. The goal is to learn how to make programs work.

7.7. DEMO error – how to find the cause of a difficult bug

This section gives a demo of how to use Pycharm for debugging. The code to debug is easy to understand – it emulates a poker card game dealer, the one that produces the cards from a deck in random order, when the players ask for some. The cause of the error in this sample code is rather difficult to find for any programmer. The program is as follows, first browse through and try to understand its logic:

```
import random

def create_all_the_cards():
    # generates a list
    # containing all the cards
    # in a standard, 52-card deck.
    suits = ["Spades", "Hearts", "Diamonds", "Clubs"]
    ranks = ["2", "3", "4", "5", "6", "7", "8", "9", "10",
             "Jack", "Queen", "King", "Ace"]
    deck = []
    for s in suits:
        for r in ranks:
            name = "%s of %s" % (r, s)
            deck.append(name)
    return deck

def draw_a_random_card_from_the(deck):
    # Draw one random card from the given deck
    remaining = len(deck)
    position = random.randint(0, remaining)
    card = remove_selected_card_from_the(deck, position)
    return card

def remove_selected_card_from_the(deck, x):
    # Return card no. x from the deck and
    # remove card from the deck
    card = deck[x]
    deck.remove(card)
    return card

def test_Dealer():
    print("Generating the 52-card deck")
    deck = create_all_the_cards()
    print("Initial deck ==", deck)
    number_to_draw = 12
    print("Drawing", number_to_draw, "random cards with no replacement:")
    drawn = set()
    for x in range(number_to_draw):
```

```

        card = draw_a_random_card_from_the(deck)
        print("  You drew", card)
        drawn.add(card)
    unique = len(drawn)
    print("Drew", unique, "different cards")
    print("Final deck ==", deck)

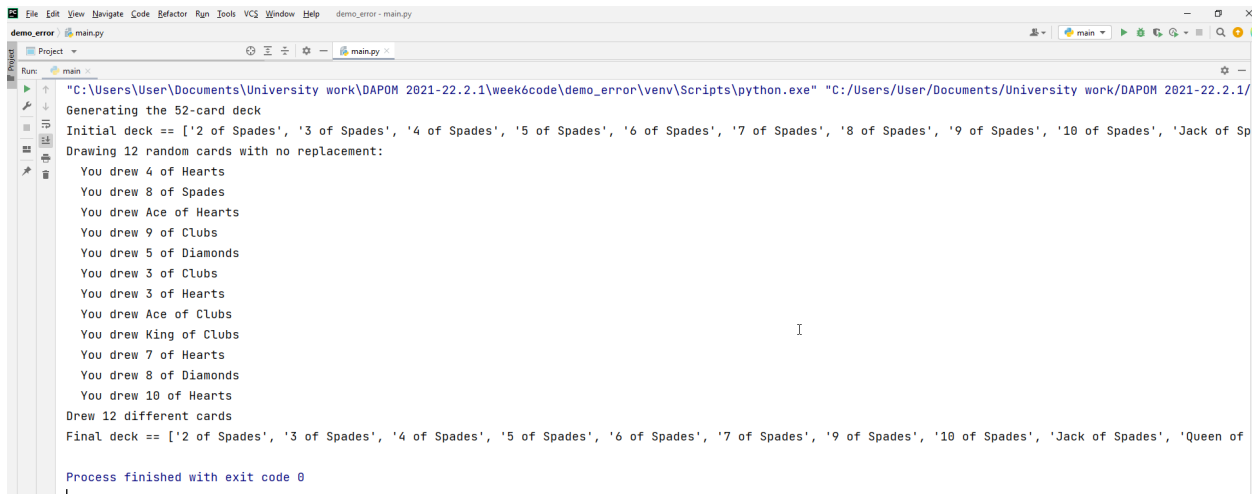
```

```

# main part
test_Dealer()

```

When executed, the program may display in the console:

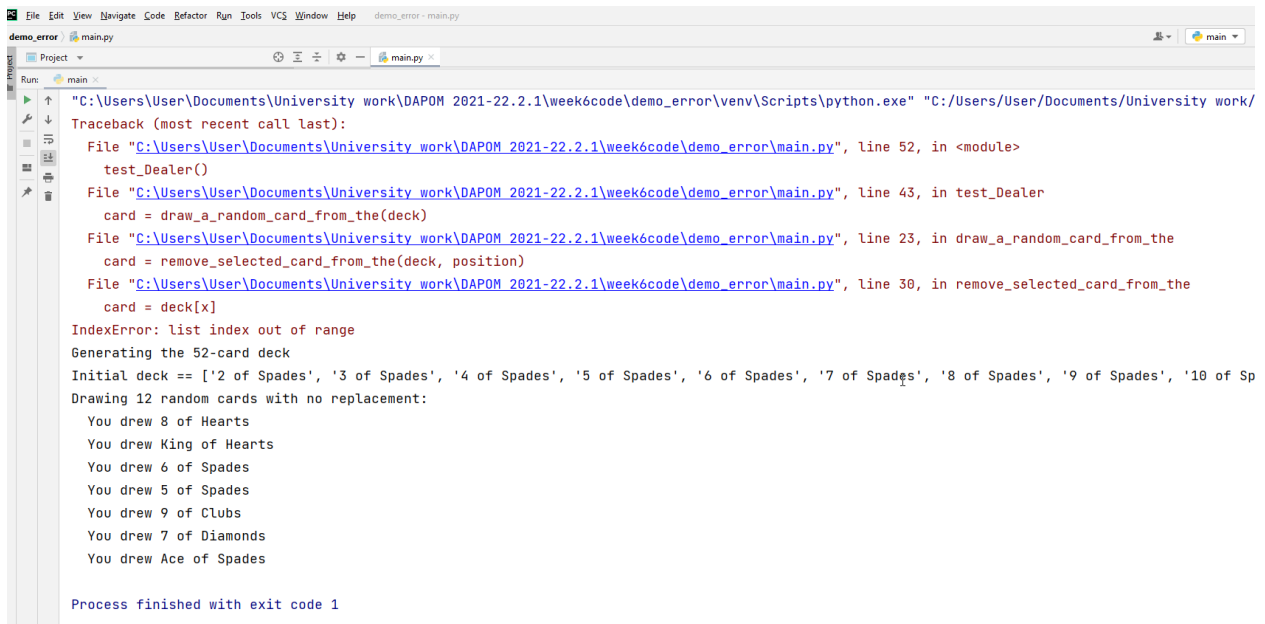


```

C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\venv\Scripts\python.exe "C:/Users/User/Documents/University work/DAPOM 2021-22.2.1/
Generating the 52-card deck
Initial deck == ['2 of Spades', '3 of Spades', '4 of Spades', '5 of Spades', '6 of Spades', '7 of Spades', '8 of Spades', '9 of Spades', '10 of Spades', 'Jack of Sp
Drawing 12 random cards with no replacement:
You drew 4 of Hearts
You drew 8 of Spades
You drew Ace of Hearts
You drew 9 of Clubs
You drew 5 of Diamonds
You drew 3 of Clubs
You drew 3 of Hearts
You drew Ace of Clubs
You drew King of Clubs
You drew 7 of Hearts
You drew 8 of Diamonds
You drew 10 of Hearts
Drew 12 different cards
Final deck == ['2 of Spades', '3 of Spades', '4 of Spades', '5 of Spades', '6 of Spades', '7 of Spades', '9 of Spades', '10 of Spades', 'Jack of Spades', 'Queen of Sp
Process finished with exit code 0

```

However, trying it repeatedly, we will get at some moment the following (apparently sudden and unexpected) output:



```

C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\venv\Scripts\python.exe "C:/Users/User/Documents/University work/
Traceback (most recent call last):
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 52, in <module>
    test_Dealer()
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 43, in test_Dealer
    card = draw_a_random_card_from_the(deck)
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 23, in draw_a_random_card_from_the
    card = remove_selected_card_from_the(deck, position)
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 30, in remove_selected_card_from_the
    card = deck[x]
IndexError: list index out of range
Generating the 52-card deck
Initial deck == ['2 of Spades', '3 of Spades', '4 of Spades', '5 of Spades', '6 of Spades', '7 of Spades', '8 of Spades', '9 of Spades', '10 of Sp
Drawing 12 random cards with no replacement:
You drew 8 of Hearts
You drew King of Hearts
You drew 6 of Spades
You drew 5 of Spades
You drew 9 of Clubs
You drew 7 of Diamonds
You drew Ace of Spades
Process finished with exit code 1

```

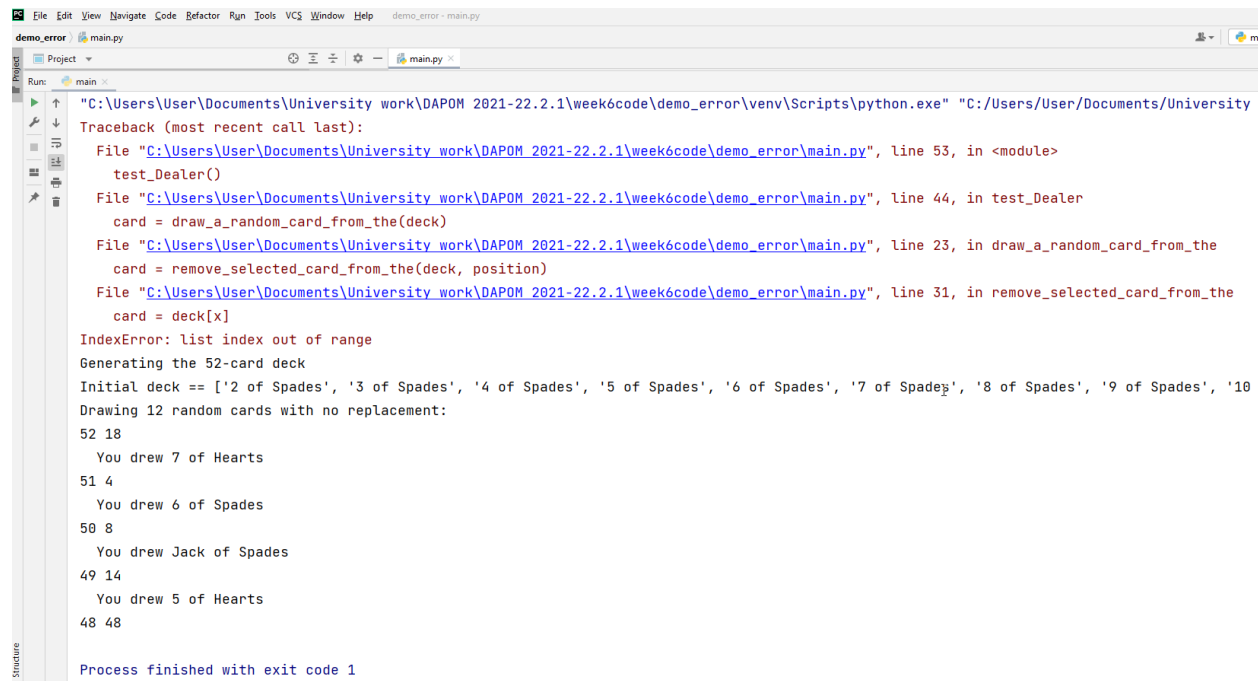
When the program crashes, we can use this information to find out the cause of the error. If we print variable values around the line where the error is reported (above, the program crashes finally in line 30, indicated by the arrow closest to the bottom of the trace) we are therefore able to inspect the contents of the variables used in the program, and we can also see where the function that crashed was called from, and which input it was given.

For example, just before line 30, we insert the following statement:

```
print(len(deck), x)
```

```
26
27 def remove_selected_card_from_the(deck, x):
28     # Return card no. x from the deck and
29     # remove card from the deck
30     print(len(deck), x)
31     card = deck[x]
32     deck.remove(card)
33     return card
```

Which will print the number of the remaining cards in deck, and the position from where we draw the card, just before that card is to be removed. The program may run for a few times normally, but then it crashes again. But this time, we have new information to look into:



The screenshot shows a Python IDE window titled 'demo_error - main.py'. The 'Run' tab is active, displaying the following output:

```
"C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\venv\Scripts\python.exe" "C:/Users/User/Documents/University
demo_error - main.py
Traceback (most recent call last):
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 53, in <module>
    test_Dealer()
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 44, in test_Dealer
    card = draw_a_random_card_from_the(deck)
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 23, in draw_a_random_card_from_the
    card = remove_selected_card_from_the(deck, position)
  File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 31, in remove_selected_card_from_the
    card = deck[x]
IndexError: list index out of range

Generating the 52-card deck
Initial deck == ['2 of Spades', '3 of Spades', '4 of Spades', '5 of Spades', '6 of Spades', '7 of Spades', '8 of Spades', '9 of Spades', '10
Drawing 12 random cards with no replacement:
52 18
You drew 7 of Hearts
51 4
You drew 6 of Spades
50 8
You drew Jack of Spades
49 14
You drew 5 of Hearts
48 48
Process finished with exit code 1
```

The last printed information, just before the program crashing indicates that there are 48 cards remaining in the deck (list) and the random number created to ask for index (position) 48 in a remaining list of 48 cards. Seems normal at the first sight, but we have to remember that in Python lists, we unnaturally start counting from 0, instead of the natural 1.

METAPHORICAL LIGHTBULB SWITCHES ON! The deck list is indexed from 0 to 47!

Voila, here it is!!!

The index in the `x` variable is too big! (BTW, when you run this program yourselves, you may crash the program with a different value of `x` – but the cause is the same). Moreover, the cause of the error message we found is consistent with the error message we got in the first place:

IndexError: list index out of range

However, we have not yet fixed the cause of the bug. We have to find the root of the problem, and eliminate it. The value 48 has been calculated somewhere else in the code and send as parameter `x` to the function that crashed.

File "C:\Users\User\Documents\University work\DAPOM 2021-22.2.1\week6code\demo_error\main.py", line 23, in draw_a_random_card_from_the
card = remove_selected_card_from_the(deck, position)

We can trace in the error output up to the function that generated the value of this parameter, by going up in the function call cascade, at line 23.

```
19 def draw_a_random_card_from_the(deck):  
20     !# Draw one random card from the given deck  
21     remaining = len(deck)  
22     position = random.randint(0, remaining)  
23     card = remove_selected_card_from_the(deck, position)  
24     return card
```

By looking into the values of the variables `remaining` and `position` (print them) we can see that both have the value 48. That means that the `random.randint()` was called with the arguments (0, 48). That tells us that the value 48 (which should not be allowed, the maximum value should be 47 in this particular case) should have never been generated. The conclusion of this error cause finding process is that we are not using the `random.randint()` function properly, allowing it to generate values that can crash our program (sometimes, and randomly). Therefore, to correct the bug, we have only to effectuate a very simple change in the code line numbered 22 above (you may have a different line number for the same statement in you code):

```
position = random.randint(0, remaining-1)
```

From now on, the program will run each time properly.

(hold your breath for a second)

Until somebody discovers a hidden error...

8. To conclude: how to continue from this point onwards, if you want to learn more Python and Data Science

8.1. This could be not the end, this could be the beginning

In this chapter, we cover the following topics:

- What are good (in our opinion) online resources to continue (only) with Python?
- What are good online resources to deepen your knowledge and improve skills in Data Science?
-

8.2. Online resources for a Python skills advanced skills learning path – that won't cost a cent

We have covered in our few previous practicals only the first essential steps in learning Python-based programming skills. For example, we have not yet touched the parts related to the advanced software engineering paradigm of Object-Oriented (OO). Learning about defining classes, object instances, and how to create complex class hierarchies based on inheritance would be for you the next logical phase of learning Python and programming. OO is an approach that is also supported by other currently programming languages in widespread use like C++, Java, C#, Swift, and others. Switching from Python to one of these should be now rather easy for you, because the basic programming constructs and jargon are the same, only the syntax differs.

The online resource of W3SCHOOLS, which we used during our course as a quick learning and reference material offers some intro into classes, but this is our opinion not sufficient. A better resource to continue is the e-learning platform named **TechBeamers** (techbeamers.com). They offer programming and testing tutorials on various topics and languages, but the best they offer is by far the Python part: <https://www.techbeamers.com/python-tutorial-step-by-step/#tutorial-list>

They offer quizzes to test yourself and also examples of interviews (what kind of questions would you expect in a job interview where you assert knowledge of Python programming and the interviewers would check that on the spot). They cover very advanced subjects like for example multithreading and Socket programming, but everything is logically organized, with tutorials that slowly grow in complexity and allow for robust learning. Going through this material would take you like **40-80 hours**, depending on how fast you go, and what topics you select (not all topics are essential).

Another e-learning material which is more concerned with programming (albeit uses Python as a tool) is the excellent open e-book of Wentworth, Elkner, Downey, and Meyers, **“How to Think Like a Computer Scientist – Learning with Python 3”**. Allen Downey wrote initially the same kind of book for C++ and Java (but these are not open). The Python based resource is online

(under a GNU Free Document License) and it is also interactive:

<http://openbookproject.net/thinkcs/python/english3e/>

In our faculty, this e-book is used as the primary teaching material for the Learning Community for Programming in Python (for both the basic and the advanced tracks – this is an extracurricular activity organized by the Career Company, and each year 70 to 80 FEB students take and finish one or the other track). This is a much more extensive material, and would take you probably **160-200 hours** of work to cover most of the chapters. If you choose not to explore advanced topics like recursion, stacks, linked lists, trees and queues, you may finish the material in less than **120 hours** of work.

The important aspect of this material is given by its title. By doing it, you will start to think like a real programmer, developing a mindset that will allow you to converse easily with computer scientists and software engineers. A bit of this new way of thinking was already inculcated into your minds by our course, but this is only a beginning and it needs to be extended and strengthened.

8.3. Online resources for a learning path in Data Science

It is possible to train into a career in data scientist without taking any formal master education in this subject. The average cost of obtaining a Data Science MSc degree (at a bricks and mortar university) in US varies between minimum \$30,000 at low ranked universities, and \$120,000 at highly ranked universities. There are online degrees/distance learning as well at various universities, but they can cost up to \$9000 per degree.

However, there are some useful resources that will cost you much less. Of, course there is a price based hierarchy of online platforms that are accessible for a fee. On the bottom of the hierarchy (i.e. the cheapest) is **Udemy**, where Jose Portilla has a Data Science online course that is recognized universally to be exceptionally good value for money:

<https://www.udemy.com/course/python-for-data-science-and-machine-learning-bootcamp/>

Currently (evening of 7 March 2022) the price for access is EUR14.99, a normal price if you buy it during a promotion period. The course is intended for Python coders, and this is a good course to start if you have not yet decided to really invest time and financial resources on your path to become more of a Data Scientist. This course will give you a good introduction into what kind of work you are supposed to do as a data analyst and what is the minimum set of skills you will need to cope with such a job.

If you really want to take a serious interest in Data Science, then you should look at the midrange of this hierarchy (a few hundred euros per year, depending on promotions). The best examples here are **Data Camp** and **Data Quest**. Some of our colleagues (especially PhD candidates) in the faculty (FEB) took six month courses with these two, both in Python and R, the second language of choice for Data Science (the prices ranged from 25 euro to 50 euro per month). They offer lots of various tracks, and they slowly takes you through all the topics you need to learn, they are interactive, and have many quizzes, exercises, and explanatory videos.

If one makes a comparison between these two, the way Data Quest teaches is more interesting and engaging, because they are much more focused on project style learning. As anyone who is a programmer or a data scientist will tell you, the best way to learn is through projects. Going only through books, exercises, and following on-line courses passively is not enough. You may understand what is taught, but you will find it very difficult to apply to different settings. If you learn by doing projects, you have to go from start to finish with a complex programming and data analysis problem, and this is how you really learn to apply data science theory.

Data Quest also offers part of their courses for free – we would strongly recommend to have a look at their offers at: <https://app.dataquest.io/>. If you want to invest in paid courses, it is a better value to purchase the annual subscription, and you have access to all their guided projects, and you don't have to pay any extras as you go through them.

For Data Science **specializations** that are offered by well-known institutions, the best resource to start is **coursera.org**. Here, the top ranked online programmes in Data Science are:

- The Data Science Specialization offered by Johns Hopkins University, consisting of ten courses (using mostly R, unfortunately for you): <https://www.coursera.org/specializations/jhu-data-science>. This will bring you from beginner level to medium level in Data Science, it covers also statistics, machine learning, information visualization, text analysis, social network analysis techniques, and ends with a project. As an example of their work, they are the ones that provide the Covid-19 most used global dashboard (<https://coronavirus.jhu.edu/map.html>).
- The Applied Data Science with Python offered by University of Michigan, consisting of five courses (basic Python knowledge is required, and Python is used throughout): <https://www.coursera.org/specializations/data-science-python> (this is a bit more advanced than the first one).

Both will cost a bit less than one thousand euros, you can enroll for free, but you have to pay as you go.

At the top of this price hierarchy are the online “**micro-masters**”, which are offered through the **edx.org** platform (at <https://www.edx.org/course/subject/data-science>). These are considered the best, most demanding, and most expensive online learning environments – beside the full master distance learning programmes. The rankings indicate two at the top:

- Micro Masters in Data Science – at University of California, San Diego, uses Python and teaches all the necessary elements (statistics, machine learning, big data analytics) and also uses and teaches the open Apache platform Spark for Big Data projects. The number of courses is flexible, the minimum is four, and currently there is a big promotion reduction for €1,153 for this minimal package: <https://www.edx.org/micromasters/uc-san-diegox-data-science>
- Micro Masters Statistics and Data Science at MIT, with 5 courses (one is the final capstone project), uses Python also, and starts with Probability before Statistics (a very sound theory-based approach). It is focused on Machine learning, from simple linear models to Deep

Learning with unsupervised Artificial Neural Networks. There is also a promotion for €1,235 currently: <https://www.edx.org/micromasters/mitx-statistics-and-data-science>

All these on-line learning platforms provide for those who finish the tracks and projects a certificate and some even give credit points (like ECTs, but American style) that are recognized by other learning institutions – to complete the curricular programme requirements for various other degrees.