Jupman

The webpage of the Scientific Programming Lab for QCB 2020

Luca Bianco

Oct 20, 2020

CONTENTS

Download: PDF[1] EPUB[2] HTML[3]

[1] http://readthedocs.org/projects/qcbsciprolab2020/downloads/pdf/latest/
[2] http://readthedocs.org/projects/qcbsciprolab2020/downloads/epub/latest/
[3] http://readthedocs.org/projects/qcbsciprolab2020/downloads/htmlzip/latest/

CHAPTER

ONE

GENERAL INFO

The contacts to reach me can be found at this page[4].

1.1 Timetable and lecture rooms

Due to the current situation regarding the Covid-19 pandemic, Practicals will take place ONLINE this year. They will be held on **Mondays from 14:30 to 16:30** and on **Wednesdays from 11:30 to 13:30**.

Practicals will use the Zoom platform (https://zoom.us/) and the link for the connection will be published on the practical page available in this site a few minutes before the start of the session.

This first part of the course will tentatively run from **Wednesday, September 23rd, 2020 to Monday, November 2nd, 2020**.

1.2 Moodle

In the moodle page of the course you can find announcements and videos of the lectures. It can be found here[5].

1.3 Zoom links

The zoom links for the practicals can be found in the Announcements section of the moodle web page.

1.4 Slides

Slides of the practicals will be available on the top part of each practical page.

[4] http://www.fmach.it/CRI/info-generali/organizzazione/Biologia-computazionale/BIANCO-LUCA
[5] https://didatticaonline.unitn.it/dol/course/view.php?id=25445

1.5 Acknowledgements

I would like to thank Dr. David Leoni for all his help and for sharing Jupman with me. I would also like to thank Dr. Stefano Teso for allowing us to use some of his material of a previous course.

CHAPTER

TWO

PRACTICAL 1

The aim of this practical is to set up a working Python3.x development environment and will start familiarizing a bit with Python.

2.1 Slides

The slides shown in the introduction can be found here: Intro

2.2 Setting up the environment

We will need to install several pieces of software to get a working programming environment suitable for this practical. In this section we will install everything that we are going to need in the next few weeks.

Python3 is available for Windows, Linux and Mac, therefore you can run it on your preferred platform.

Note:

Altough for this course you will be fine with any operating system, my advice, if you are interested in pursuing a bioinformatics career, is to get familiar with Linux.

The following section explains how to install Linux on a windows machine. This is for your reference, you can read the following instructions before the next practical and try to instally Linux if you want to test it out.

2.2.1 Linux on windows

If your computer has Windows installed but you want to learn Linux you have several options to get it to run Linux:

1. This video tutorial (only in Italian) shows you how to set up a usb stick to run Linux from it: https://youtu.be/8_SK8iEMyJk
2. You can install a virtualization software like vmware player[6] and download the .iso image of a linux distribution like ubuntu.[7] and install/run it from vmware player. For more information you can look at this tutorial.[8] Another option is to install virtual box[9].

[6] https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/15_0%7CPLAYER-1550%7Cproduct_downloads

[7] https://ubuntu.com/#download

[8] https://www.youtube.com/watch?v=9rUhGWijf9U

[9] https://www.virtualbox.org/wiki/Downloads

Here[10] you can find some VDI images that you can load in virtual box or in vmware player with several different operating systems including Linux distributions like Ubuntu, Debian, Centos, Fedora, etc. Please refer to this guide[11] (for information on vmware please click on **VM IMAGES –> VMware IMAGES** in the menu of the page).

2.2.2 A dual boot system

You can also install **Linux and Windows on the same machine** and every time you boot your system up **you can decide on which one of the two operating systems you want to use**. Unlike the case described above in which Linux runs **within** Windows, in this case to switch from one operating system to the other you will always have to reboot the machine.

The installation of a dual boot system is easy, in principle, but there are a few things that you have to be careful on, like creating a partition of the hard disk on which you want to install Linux. If you make a mistake here you might end up losing Windows for example. My advice is to read carefully one of the following (or other guides) before attempting this:

- How To Install Ubuntu Along With Windows[12]
- How to Dual Boot Ubuntu 20.04 LTS and Windows 10[13]
- How to Dual boot Windows 10 and Linux (Beginner's Guide)[14]

2.3 Our toolbox

If you decide to work on Windows or Mac, you can safely skip the following information and go straight to the section **"Installing Python3 in Windows/Mac"**. Note that, regardless your operating system, a useful source of information on how to install python can be found here[15].

2.4 Installing Python3 in Linux

1. The Python interpreter. In this course we will use python version 3.x. A lot of information on python can be found on the python web page[16]. Open a terminal and try typing in:

```
python3
```

if you get an error like "python3 command not found" you need to install it, while if you get something like this (note that the version might be different):

```
biancol@bludell:~$ python3
Python 3.6.8 (default, Aug 20 2019, 17:12:48)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

you are already sorted, just press Ctrl-D to exit.

Installation on a debian-like linux distribution (e.g. Ubuntu) can be done by typing the following commands on a terminal:

```
sudo apt-get update
```

[10] https://www.osboxes.org/virtualbox-images/
[11] https://www.osboxes.org/guide/
[12] https://itsfoss.com/install-ubuntu-dual-boot-mode-windows/
[13] https://www.youtube.com/watch?v=-iSAyiicyQY
[14] https://averagelinuxuser.com/dualboot-linux-windows/
[15] http://docs.python-guide.org/en/latest/
[16] https://www.python.org/

```
sudo apt-get install python3
```

While **if you are using Fedora** you can use:

```
sudo dnf install python3
```

2. Install now the package manager pip, which is a very convenient tool to install python packages, with the following command (**on Fedora, the command above should have already installed it**):

   ```
   sudo apt-get install python3-pip
   ```

   Note:

   If pip is already installed in your system you will get a message like: python3-pip is already the newest version (3.x.y)

3. Finally, install the Integrated Development Environment (IDE) that we will be using. This is called Visual Studio Code and is available for all platforms. You can read about it here[17]. Downloads for all platforms can be found here[18]. On a debian-like distribution go to the folder where you downloaded the .deb package and type:

   ```
   sudo dpkg -i code*.deb
   ```

   While **if you are using Fedora** you can use:

   ```
   sudo dnf install code*.rpm
   ```

2.5 Installing Python3 in Windows/Mac

Two options are available, please read them both **CAREFULLY** and then pick the one you are more comfortable with.

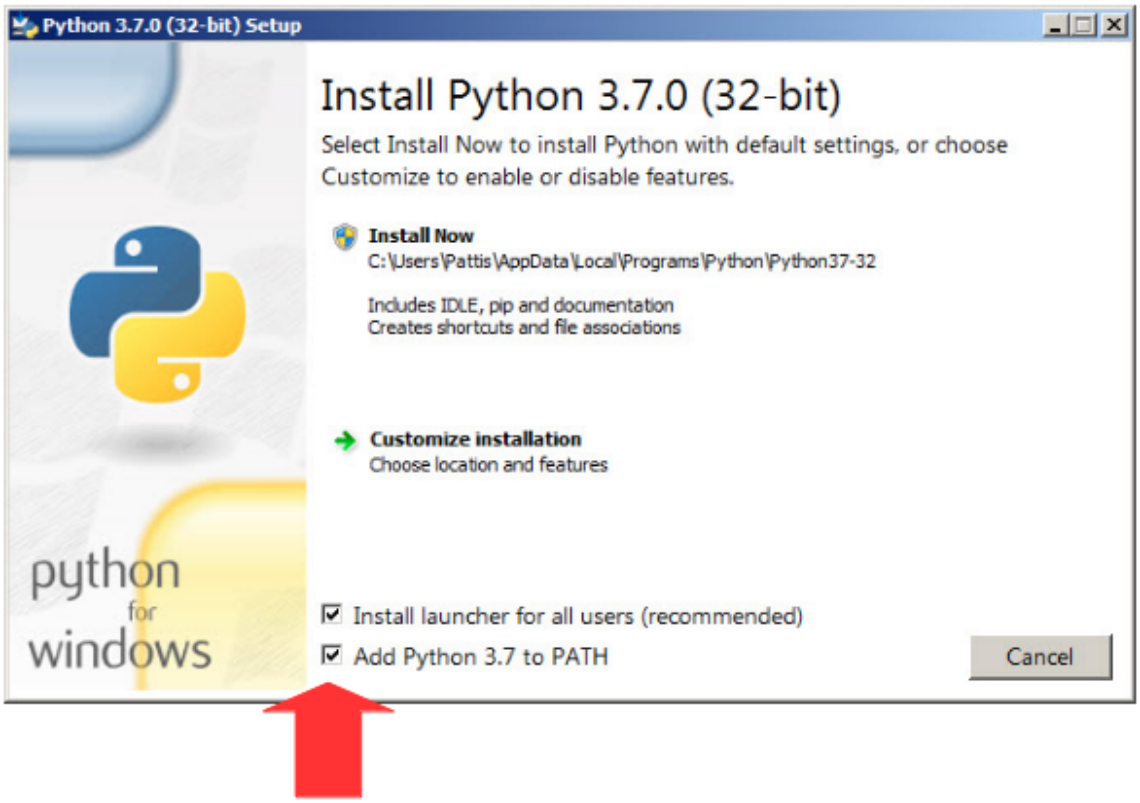
2.5.1 OPTION 1:

1. The python interpreter. In this course we will use python version 3.x. A lot of information on python can be found on the python web page[19]. Installers for Windows and Mac can be downloaded from this page[20]. Click on Download Python 3.8.x. **PLEASE REFRAIN FROM DOUBLE-CLICKING ON THE INSTALLER LIKE THERE IS NO TOMORROW AND READ BELOW FIRST.**

   Attention! Important note

   When executing the installer, please remember to tick the flag "Add Python 3.8.x to PATH" and then click on Install now (see picture below noting that the current version might differ from the picture).

[17] https://code.visualstudio.com/
[18] https://code.visualstudio.com/Download
[19] https://www.python.org/
[20] https://www.python.org/downloads/



2. Install now the Integrated Development Environment (IDE) that we will be using. This is called Visual Studio Code and is available for all platforms. You can read about it here[21]. Downloads for all platforms can be found here[22].

2.5.2 OPTION 2 (easier):

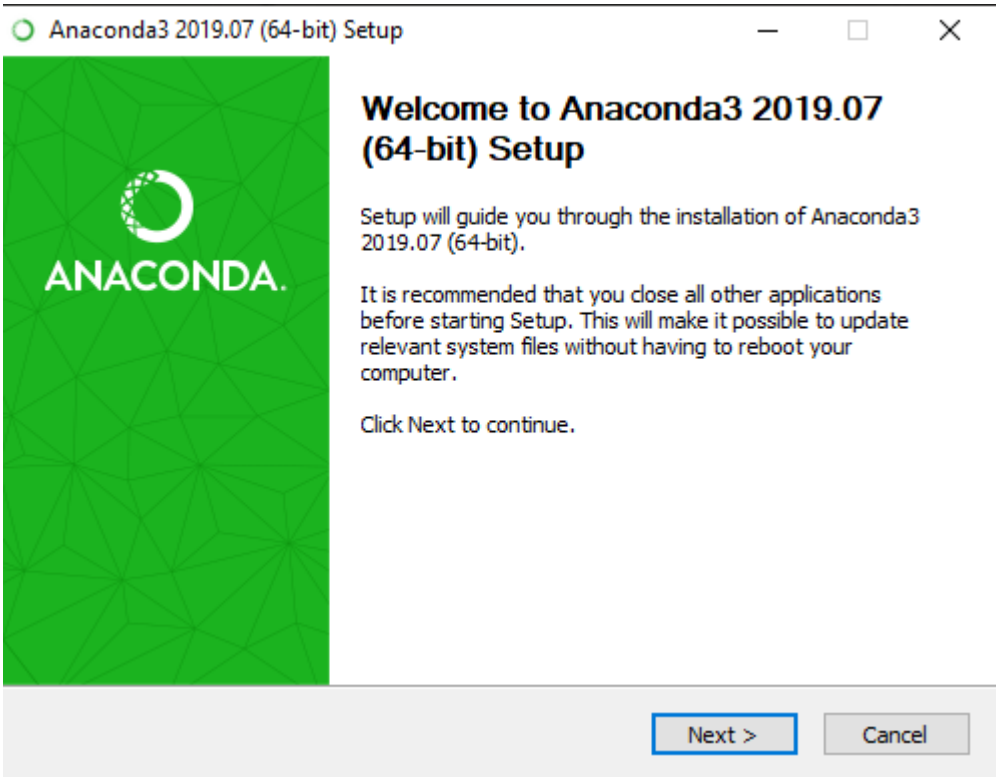
Additional Information:

It is also possible to install python through the Anaconda package manager. You can install Visual Studio Code together with Anaconda(the Anaconda installer will ask if you want it, just say yes!).

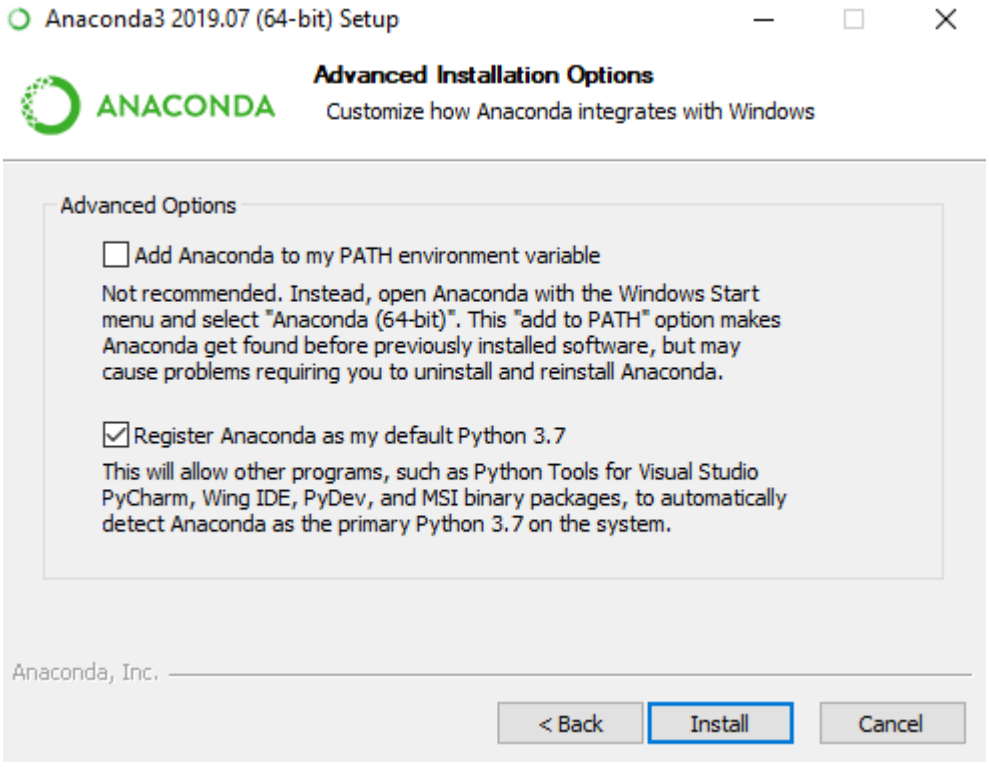
Anaconda is available here[23]

Upon launching the installer you should be prompted something like:

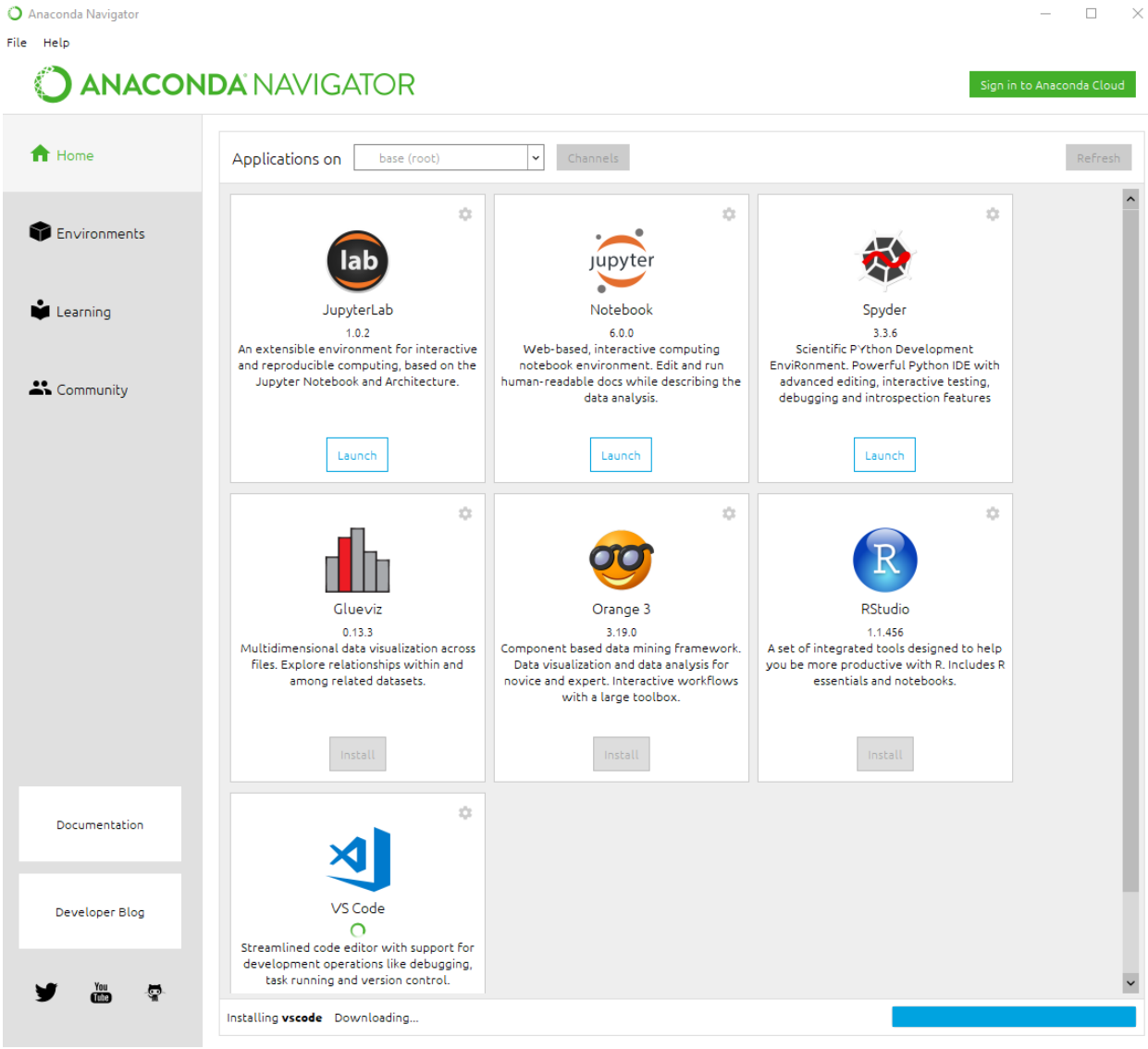
[21] https://code.visualstudio.com/
[22] https://code.visualstudio.com/Download
[23] https://www.anaconda.com/distribution/



at the next step flag the correct items as in the figure below (i.e. **Flag Register Anaconda as my Default Python 3.x**):



When installation is complete, start anaconda through the **Anaconda Navigator** in the windows menu. When the navigator starts, you should see a screen similar to:



from which you can install Visual Studio Code as IDE (by clicking on Install).

For more information please have a look here[24].

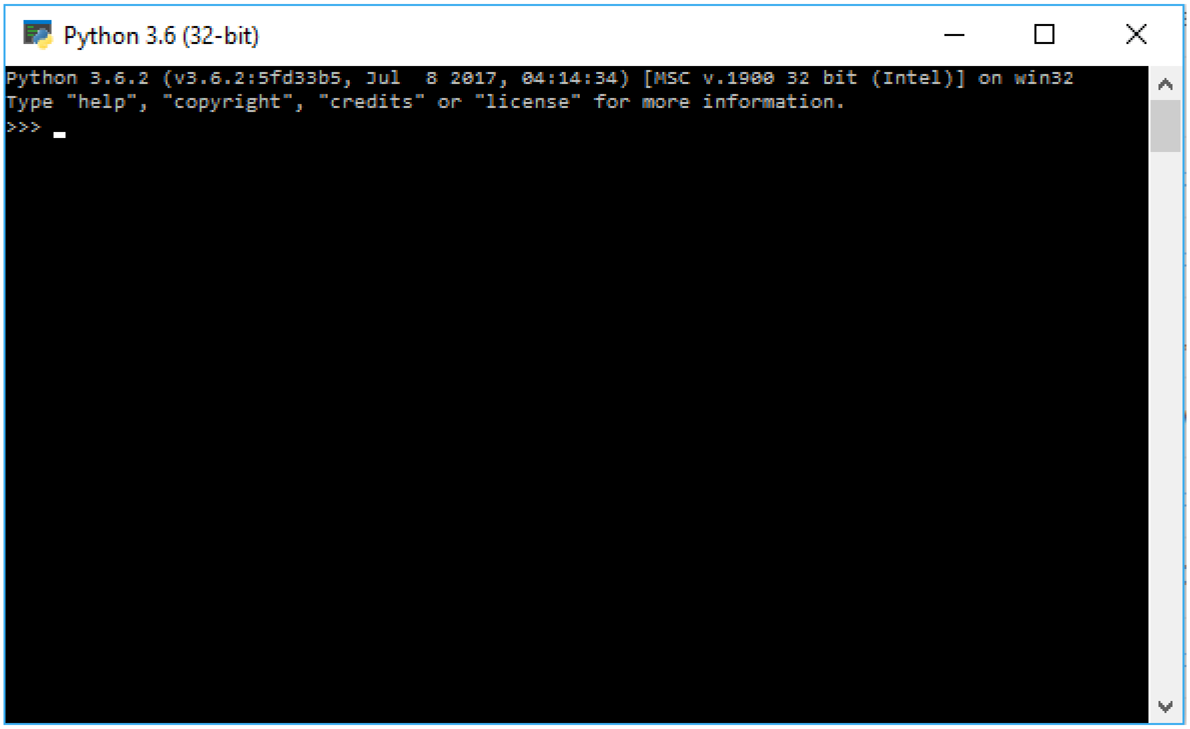
2.6 The console

To access the console on Linux just open a terminal and type:

```
python3
```

while in Windows you have to look for "Python" and run "Python 3.x". The console should look like this:

[24] https://docs.anaconda.com/anaconda/user-guide/getting-started/#open-nav-win



Now we are all set to start interacting with the Python interpreter. In the console, type the following instructions (i.e. the first line and then press ENTER):

```
[1]: 5 + 3
```

```
[1]: 8
```

All as expected. The "In [1]" line is the input, while the "Out [1]" reports the output of the interpreter. Let's challenge python with some other operations:

```
[2]: 12 / 5
```

```
[2]: 2.4
```

```
[3]: 1/133
```

```
[3]: 0.007518796992481203
```

```
[4]: 2**1000
```

```
[4]: 10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581
```

And some assignments:

```
[5]: a = 10
     b = 7
     s = a + b
     d = a / b

     print("sum is:",s, " division is:",d)
```

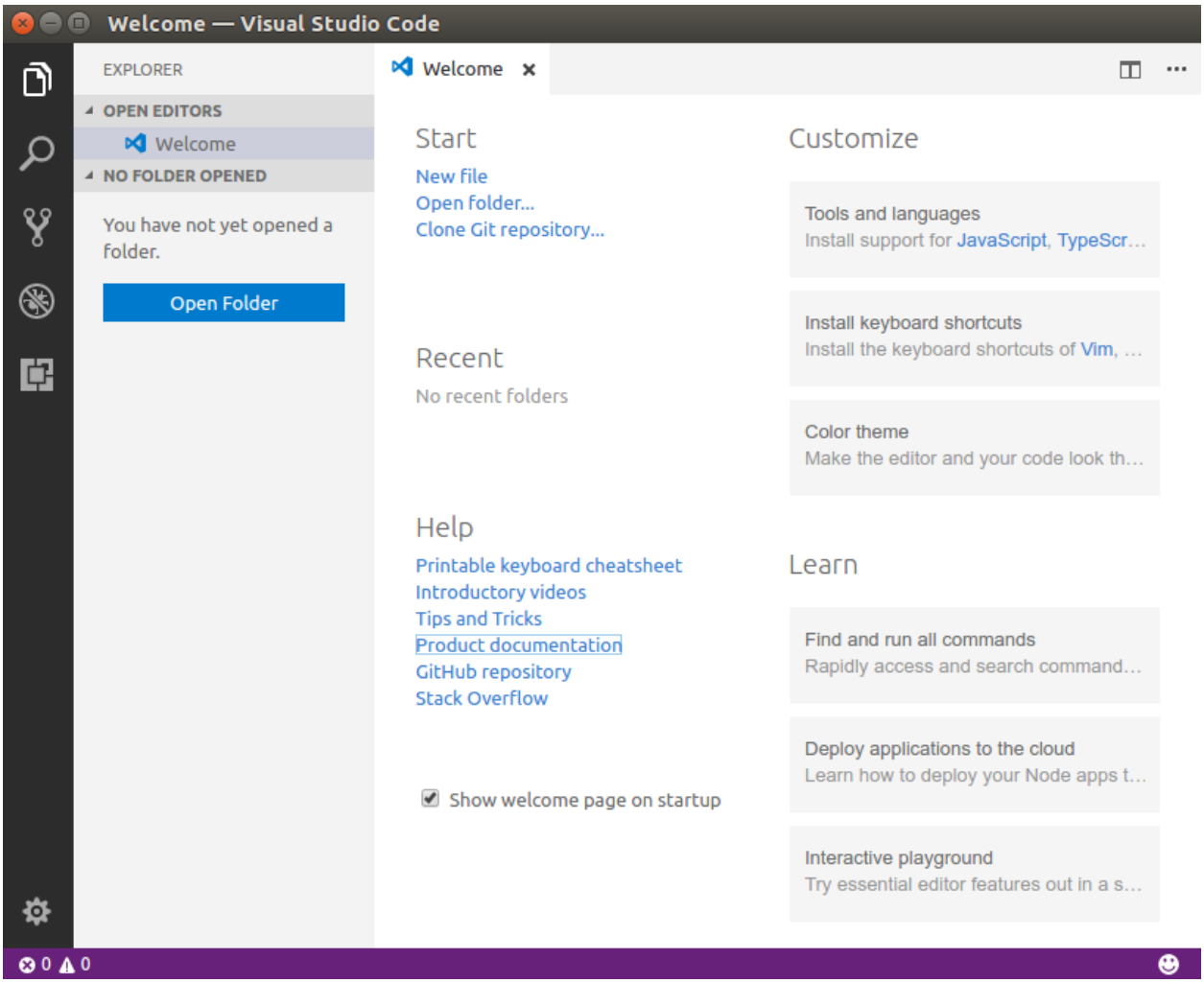
```
sum is: 17  division is: 1.4285714285714286
```

In the first four lines, values have been assigned to variables through the = operator. In the last line, the print function is used to display the output. For the time being, we will skip all the details and just notice that the print function somehow

managed to get text and variables in input and coherently merged them in an output text. Although quite useful in some occasions, the console is quite limited therefore you can close it for now. To exit press Ctrl-D or type exit() and press ENTER.

2.7 Visual Studio Code

Once you open the IDE Visual Studio Code you will see the welcome screen:



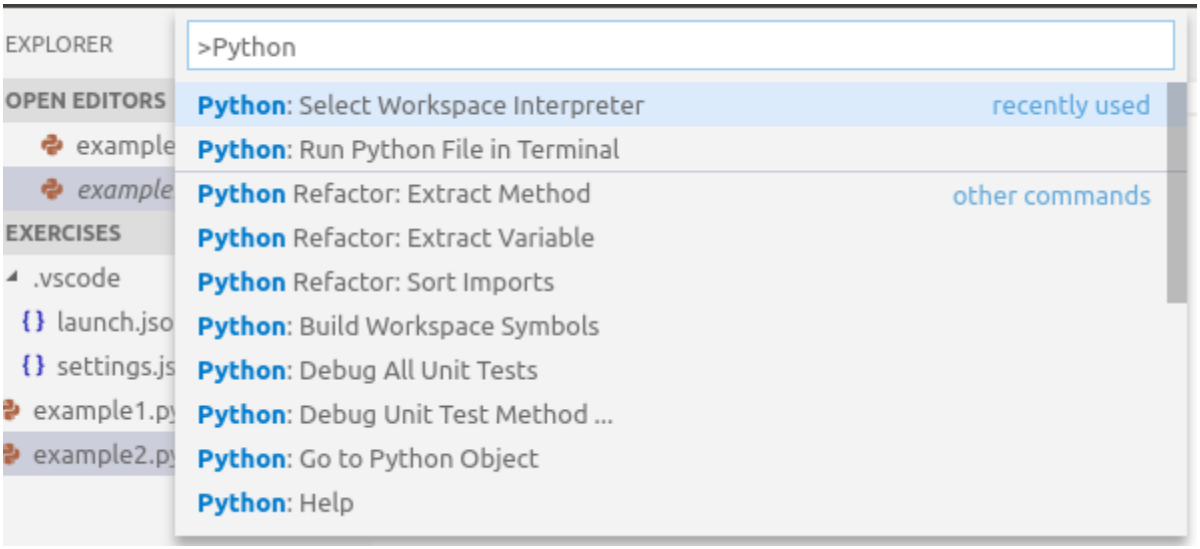
You can find useful information on this tool here[25]. Please spend some time having a look at that page. Once you are done with it you can close this window pressing on the "x".

Attention! Important note

The following procedure is quite important and you will need to remember it to do the exams on the PCs of the lab.

The first thing to do is to set the python interpreter to use. Click on **View –> Command Palette** and type **"Python"** in the text search space. Select **Python: Select Workspace Interpreter** as shown in the picture below.

[25] https://code.visualstudio.com/docs#vscode



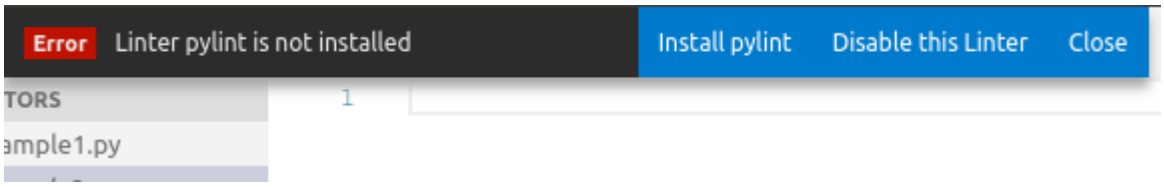
Finally, select the python version you want to use (e.g. Python3.x).

Now you can click on **Open Folder** to create a new folder to place all the scripts you are going to create. You can call it something like "exercises". Next you can create a new file, *example1.py* (as you might have guessed the **.py** extension stands for python).

Visual Studio Code will understand that you are writing Python code and will help you writing valid syntax in your scripts.

Warning:

If you get the following error message:



click on **Install Pylint** which is a useful tool to help your coding experience.

Add the following text to your **example1.py** file.

[6]:
```
"""
This is the first example of Python script.
"""
a = 10 # variable a
b = 33 # variable b
c = a / b # variable c holds the ratio

# Let's print the result to screen.
print("a:", a, " b:", b, " a/b=", c)
```

```
a: 10  b: 33  a/b= 0.30303030303030304
```

A couple of things worth nothing: the first three lines opened and closed by """ are some text describing the content of the script. Moreover, comments are proceeded by the hash key (#) and they are just ignored by the python interpreter.

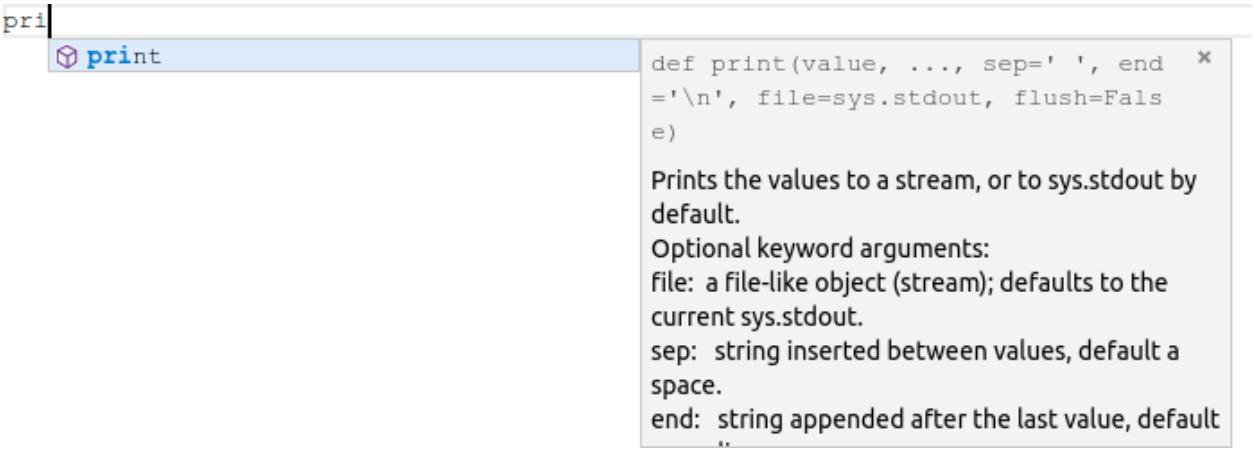
Note

Good *Pythonic* code follows some syntactic rules on how to write things, naming conventions etc. The IDE will help you writing pythonic code even though we will not enforce this too much in this course. If you are interested in getting more details on this, you can have a look at the PEP8 Python Style Guide[26] (Python Enanchement Proposals - index 8).

Warning

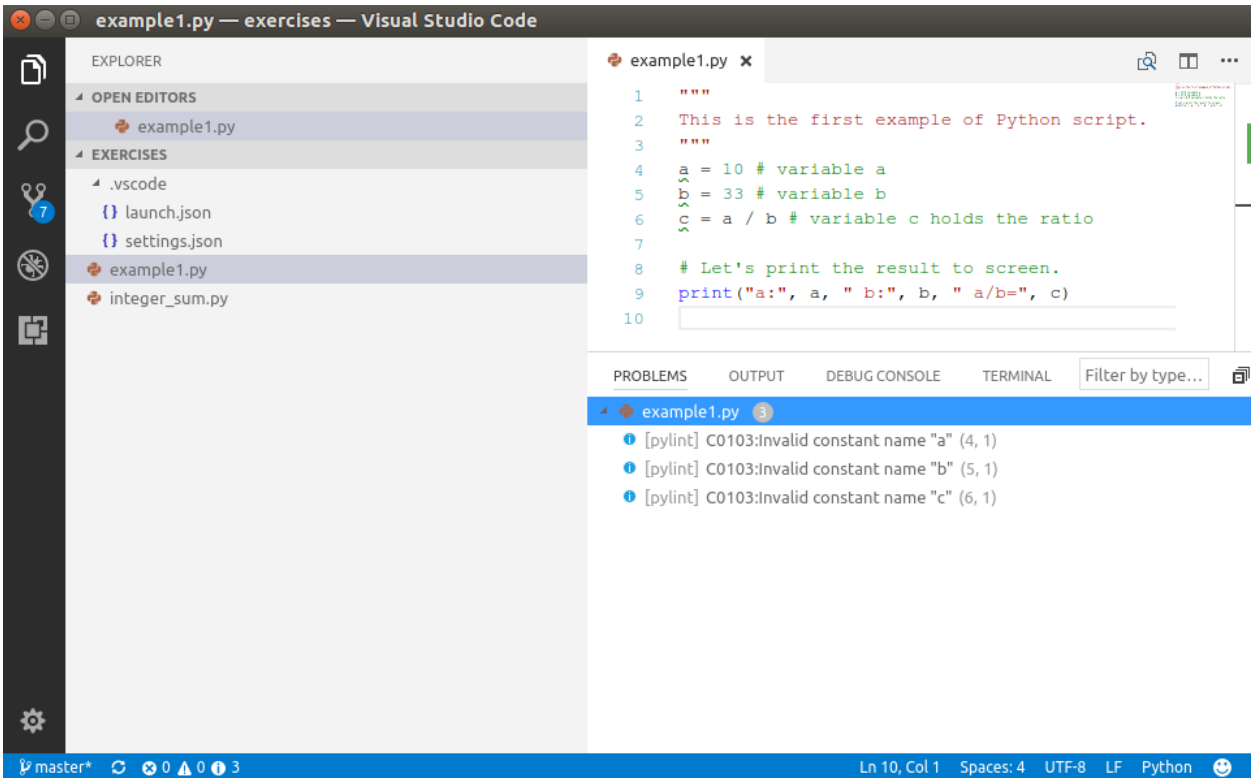
Please remember to comment your code, as it helps readability and will make your life easier when you have to modify or just understand the code you wrote some time in the past.

Please notice that Visual Studio Code will help you writing your Python scripts. For example, when you start writing the **print** line it will complete the code for you (**if the Pylint extension mentioned above is installed**), suggesting the functions that match the letters typed in. This useful feature is called **code completion** and, alongside suggesting possible matches, it also visualizes a description of the function and parameters it needs. Here is an example:



Save the file (Ctrl+S as shortcut). It is convenient to ask the IDE to highlight potential *syntactic* problems found in the code. You can toggle this function on/off by clicking on **View –> Problems**. The *Problems* panel should look like this

[26] https://www.python.org/dev/peps/pep-0008/



Visual Studio Code is warning us that the variable names *a,b,c* at lines 4,5,6 do not follow Python naming conventions for constants (do you understand why? Check here[27] to find the answer). This warning is because they have been defined at the top level (there is no structure to our script yet) and therefore are interpreted as constants. The naming convention for constants states that they should be in capital letters. To amend the code, you can just replace all the names with the corresponding capitalized name (i.e. A,B,C). If you do that, and you save the file again (Ctrl+S), you will see all these problems disappearing as well as the green underlining of the variable names. If your code does not have an empty line before the end, you might get another warning "*Final new line missing*".

Info

Note that these were just warnings and the interpreter **in this case** will happily and correctly execute the code anyway, but it is always good practice to understand what the warnings are telling us before deciding to ignore them!

Had we by mistake mispelled the **print** function name (something that should not happen with the code completion tool that suggests functions names!) writing *printt* (note the double t), upon saving the file, the IDE would have underlined in red the function name and flagged it up as a problem.

[27] https://www.python.org/dev/peps/pep-0008/#constants

```
1  """
2  This is the first example of Python script.
3  """
4  a = 10 # variable a
5  b = 33 # variable b
6  c = a / b # variable c holds the ratio
7
8  # Let's print the result to screen.
9  printt("a:", a, " b:", b, " a/b=", c)
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Filter by t

example1.py 4

[pylint] E0602:Undefined variable 'printt' (9, 1)

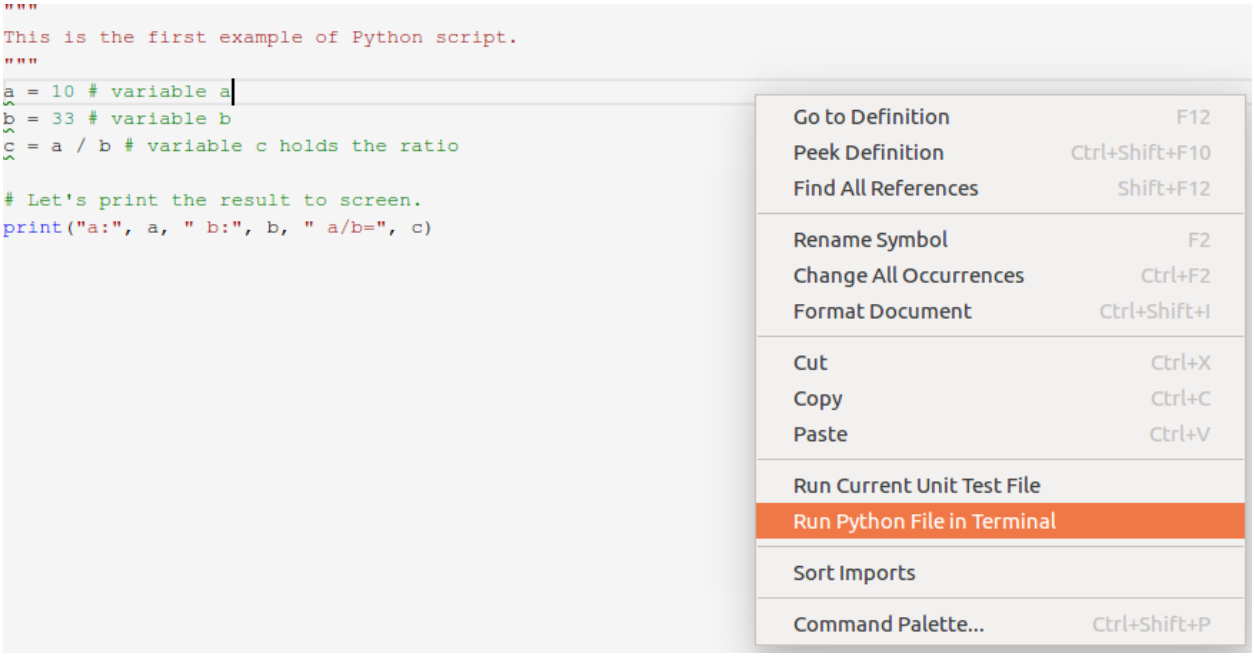
[pylint] C0103:Invalid constant name "a" (4, 1)

[pylint] C0103:Invalid constant name "b" (5, 1)

[pylint] C0103:Invalid constant name "c" (6, 1)

This is because the builtin function *printt* does not exist and the python interpreter does not know what to do when it reads it. Note that *printt* is actually underlined in red, meaning that there is an error which will cause the interpreter to stop the execution with a failure. **Please remember ALL ERRORS MUST BE FIXED before running any piece of code.**

Now it is time to execute the code. By **right-clicking** in the code panel and selecting **Run Python File in Terminal** (see picture below) you can execute the code you have just written.



Upon clicking on *Run Python File in Terminal* a terminal panel should pop up in the lower section of the coding panel and the result shown above should be reported.

Saving script files like the **example1.py** above is also handy because they can be invoked several times (later on we will

learn how to get inputs from the command line to make them more useful…). To do so, you just need to call the python intepreter passing the script file as parameter. From the folder containing the *example1.py* script:

```
python3 example1.py
```

will in fact return:

```
a: 10 b: 33 a/b= 0.30303030303030304
```

Info: syntactic vs semantic errors

Before ending this section, let me add another note on errors. The IDE will diligently point you out **syntactic** warnings and errors (i.e. errors/warnings concerning the structure of the written code like name of functions, number and type of parameters, etc.) but it will not detect **semantic** or **runtime** errors (i.e. connected to the meaning of your code or to the value of your variables). These sort of errors will most probably make your code crash or may result in unexpected results/behaviours. In the next section we will introduce the debugger, which is a useful tool to help detecting these errors.

Before getting into that, consider the following lines of code (do not focus on the *import* line, this is only to load the mathematics module and use its method *sqrt* to compute the square root of its parameter):

[7]:
```
"""
Runtime error example, compute square root of numbers
"""
import math

A = 16
B = math.sqrt(A)
C = 5*B
print("A:", A, " B:", B, " C:", C)

D = math.sqrt(A-C) # whoops, A-C is now -4!!!
print(D)
```

```
A: 16  B: 4.0  C: 20.0
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-5d4ed1b10924> in <module>
      9 print("A:", A, " B:", B, " C:", C)
     10
---> 11 D = math.sqrt(A-C) # whoops, A-C is now -4!!!
     12 print(D)

ValueError: math domain error
```

If you add that code to a python file (e.g. sqrt_example.py), you save it and you try to execute it, you should get an error message as reported above. You can see that the interpreter has happily printed off the vaule of A,B and C but then stumbled into an error at line 9 (math domain error) when trying to compute $\sqrt{A-C} = \sqrt{-4}$, because the sqrt method of the math module cannot be applied to negative values (i.e. it works in the domain of real numbers).

Please take some time to familiarize with Visual Studio Code (creating files, saving files etc.) as in the next practicals we will take this ability for granted.

2.8 The debugger

Another important feature of advanced Integrated Development Environments (IDEs) is their debugging capabilities. Visual Studio Code comes with a debugging tool that can help you trace the execution of your code and understand where possible errors hide.

Write the following code on a new file (let's call it *integer_sum.py*) and execute it to get the result.

[1]:
```
""" integer_sum.py is a script to
 compute the sum of the first 1200 integers. """

S = 0
for i in range(0, 1201):
    S = S + i

print("The sum of the first 1200 integers is: ", S)
```

```
The sum of the first 1200 integers is:  720600
```

Without getting into too many details, the code you just wrote starts initializing a variable S to zero, and then loops from 0 to 1200 assigning each time the value to a variable i, accumulating the sum of S + i in the variable S.

A final thing to notice is indentation.

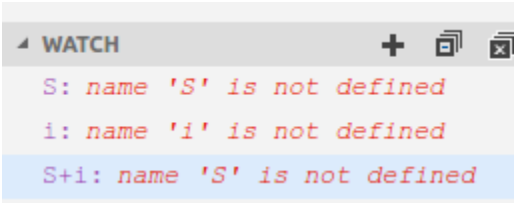
Info

In Python it is important to indent the code properly as this provides the right scope for variables (e.g. see that the line *S = S + 1* starts more to the right than the previous and following line – this is because it is inside the for loop). You do not have to worry about this for the time being, we will get to this in a later practical…

How does this code work? How does the value of S and i change as the code is executed? These are questions that can be answered by the debugger.

To start the debugger, click on **Debug –> Start Debugging** (shortcut F5). The following small panel should pop up:

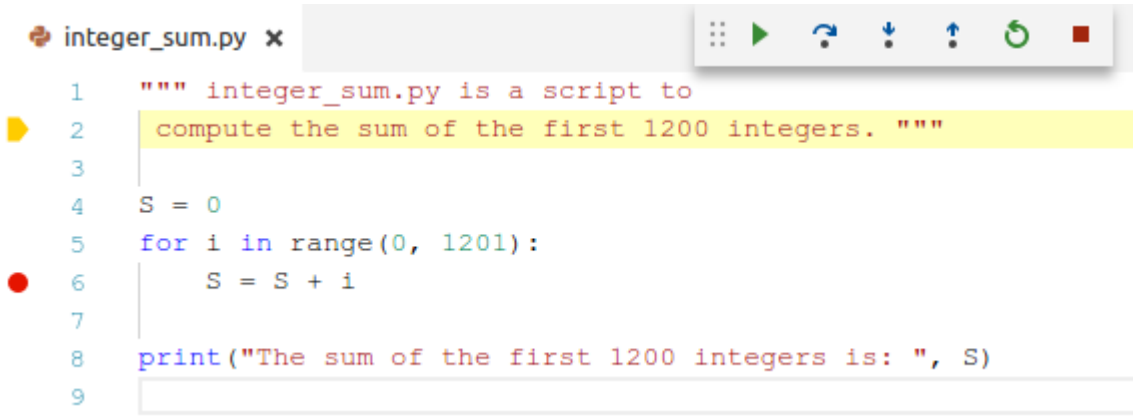
We will use it shortly, but before that, let's focus on what we want to track. On the left hand side of the main panel, a *Watch* panel appeared. This is where we need to add the things we want to monitor as the execution of the program goes. With respect to the code written above, we are interested in keeping an eye on the variables S, i and also of the expression S+i (that will give us the value of S of the next iteration). Add these three expressions in the watch panel (click on **+** to add new expressions). The watch panel should look like this:



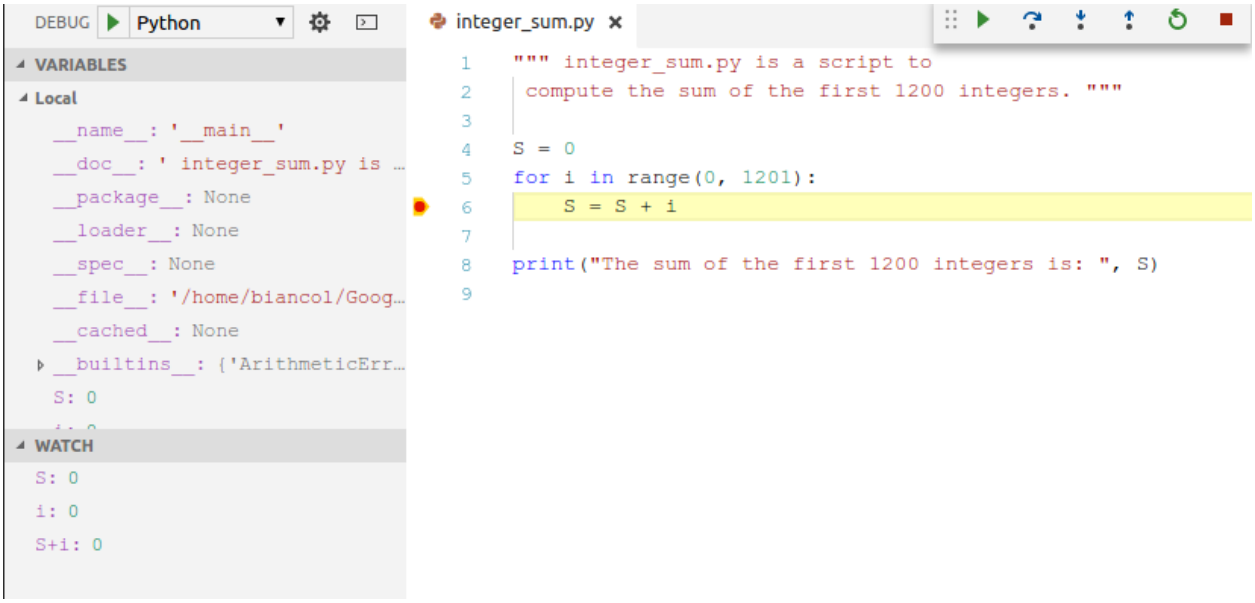
do not worry about the message "*name X is not defined*", this is normal as no execution has taken place yet and the interpreter still does not know the value of these expressions.

The final thing before starting to debug is to set some breakpoints, places where the execution will stop so that we can check the value of the watched expressions. This can be done by hovering with the mouse on the left of the line number.

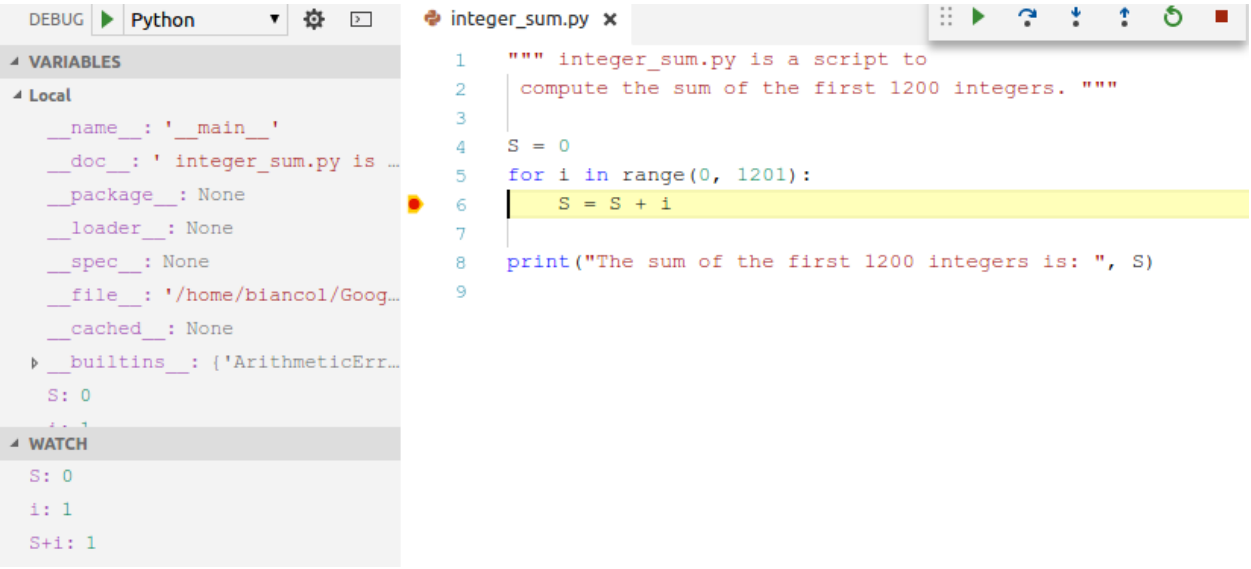
A small reddish dot should appear, place the mouse over the correct line (e.g. the line corresponding to $S = S + I$ and click to add the breakpoint (a red dot should appear once you click).



Now we are ready to start debugging the code. Click on the green triangle on the small debug panel and you will see that the yellow arrow moved to the breakpoint and that the watch panel updated the value of all our expressions.



The value of all expressions is zero because the debugger stopped **before** executing the code specified at the breakpoint line (recall that S is initialized to 0 and that i will range from 0 to 1200). If you click again on the green arrow, execution will continue until the next breakpoint (we are in a for loop, so this will be again the same line - trust me for the time being).



Now i has been increased to 1, S is still 0 (remember that the execution stopped **before** executing the code at the breakpoint) and therefore S + i is now 1. Click one more time on the green arrow and values should update accordingly (i.e. S to 1, i to 2 and S + i to 3), another round of execution should update S to 3, i to 3 and S + i to 6. Got how this works? Variable i is increased by one each time, while S increases by i. You can go on for a few more iterations and see if this makes any sense to you, once you are done with debugging you can stop the execution by pressing the red square on the small debug panel.

Note

The debugger is very useful to understand what your program does. Please spend some time to understand how this works as being able to run the debugger properly is a good help to identify and solve **semantic errors** of your code.

Other editors are available, if you already have your favourite one you can stick to it. Some examples are:

- Spyder[28]
- PyCharm Community Edition[29]
- Jupyter Notebook[30]. Note: we might use it later on in the course.

2.9 A quick Jupyter primer (just for your information, skip if not interested)

Jupyter allows to write notebooks organized in cells (these can be saved in files with .ipynb extension). Notebooks contain both the **code**, some **text describing the code** and the **output of the code execution**, they are quite useful to produce some quick reports on data analysis. where there is both code, output of running that code and text. The code by default is Python, but can also support other languages like R). The text is formatted using the **Markdown language**[31] - see cheatsheet[32] for its details. *Jupyter is becoming the de-facto standard for writing technical documentation.*

[28] https://pythonhosted.org/spyder/installation.html
[29] https://www.jetbrains.com/pycharm/
[30] http://jupyter.org/
[31] https://en.wikipedia.org/wiki/Markdown
[32] https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

2.9.1 Installation

To install it (if you have not installed python with Anaconda otherwise you should have it already):

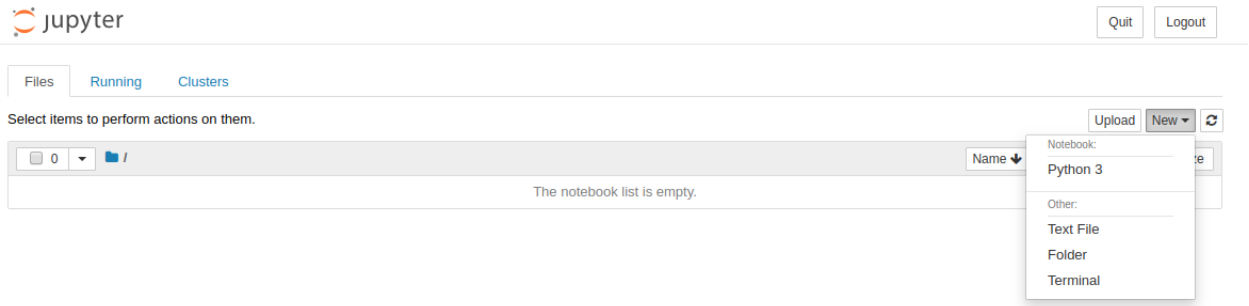
```
python3 -m pip install jupyter
```

you can find more information here[33]

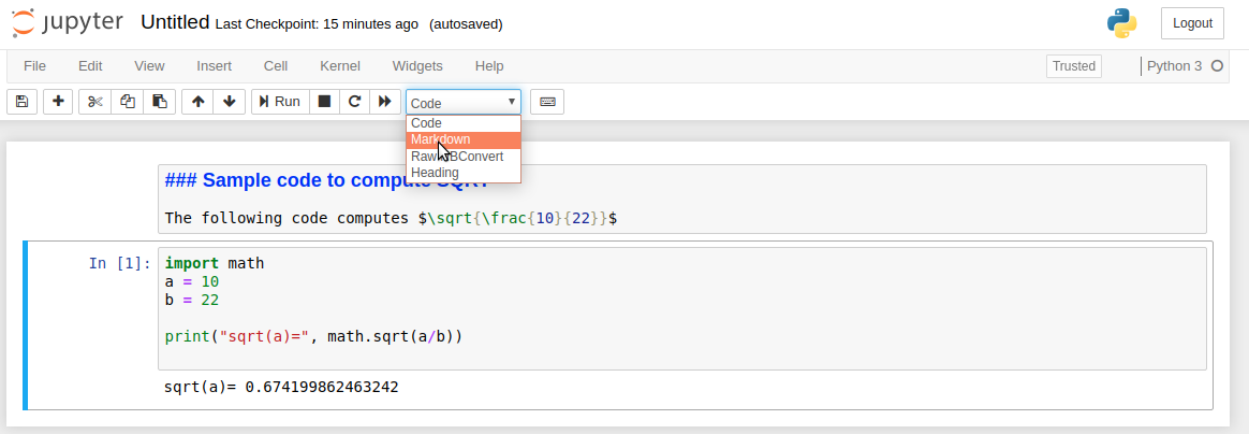
Upon successful installation, you can run it with:

```
jupyter-notebook
```

This should fire up a browser on a page where you can start creating your notebooks or modifying existing ones. To create a new notebook you simply click on **New**:

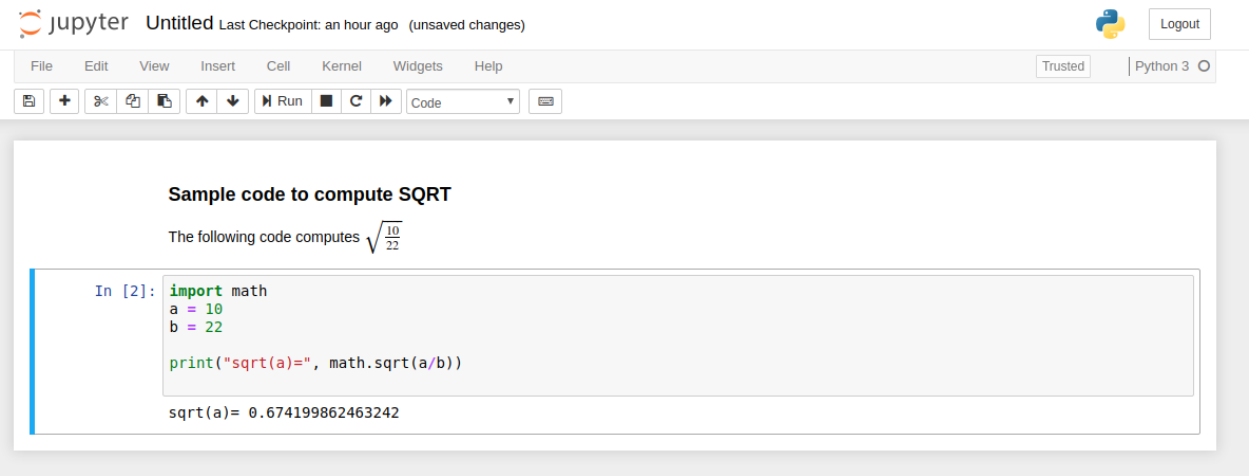


and then you can start adding cells (i.e. containers of code and text). The type of each cell is specified by selecting the cell and selecting the right type in the dropdown list:



Cells can be executed by clicking on the **Run** button. This will get the code to execute (and output to be written) and text to be processed to provide the final page layout. To go back to the edit mode, just double click on an executed cell.

[33] https://jupyter.org/install



Please take some more time to familiarize with Visual Studio Code (creating files, saving files, interacting with the debugger etc.) as in the next practicals we will take this ability for granted. Once you are done you can move on and do the following exercises.

2.10 Exercises

1. The size of the Sars-Cov-2 genome is 29,811 base pairs. 8,903 of these bases are adenines. Write some python code to compute the percentage of the genome that is an adenine and print it.

Show/Hide Solution

[2]:
```
gen_size = 29811
adenines = 8903
fraction = 100*(adenines/gen_size)
print("The Sars-Cov-2 genome has ", fraction, "% adenines")
```

```
The Sars-Cov-2 genome has  29.864815001174062 % adenines
```

2. Compute the area of a triangle having base 120 units (B) and height 33 (H). Assign the result to a variable named area and print it.

Show/Hide Solution

[3]:
```
B = 120
H = 33
Area = B*H/2
print("Triangle area is:", Area)
```

```
Triangle area is: 1980.0
```

3. Compute the area of a square having side (S) equal to 145 units. Assign the result to a variable named area and print it.

Show/Hide Solution

[4]:
```
S = 145
Area = S*S
print("Square area is:",Area)
```

```
Square area is: 21025
```

4. Modify the program at point 2. to acquire the side S from the user at runtime. Hint: use the input function (details here[34]) and remember to convert the acquired value into an int.

Show/Hide Solution

[5]:
```
S_str = input("Insert size: ")
print(type(S_str))
print(S_str)
S = int(S_str)
print(type(S))
print(S)
Area = S**2
print("Square area is:",Area)
```

```
Insert size: 27
<class 'str'>
27
<class 'int'>
27
Square area is: 729
```

5. If you have not done so already, put the two previous scripts in two separate files (e.g. triangle_area.py and square_area.py and execute them from the terminal).

6. Write a small script (trapezoid.py) that computes the area of a trapezoid having major base (MB) equal to 30 units, minor base (mb) equal to 12 and height (H) equal to 17. Print the resulting area. Try executing the script from inside Visual Studio Code and from the terminal.

Show/Hide Solution

[6]:
```
"""trapezoid.py"""
MB = 30
mb = 12
H = 17
Area = (MB + mb)*H/2
print("Trapezoid area is: ", Area)
```

```
Trapezoid area is:  357.0
```

7. Rewrite the example of the sum of the first 1200 integers by using the following equation: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Show/Hide Solution

[7]:
```
N = 1200

print("Sum of first 1200 integers: ", N*(N+1)/2)
```

```
Sum of first 1200 integers:  720600.0
```

8. Modify the program at point 6. to make it acquire the number of integers to sum N from the user at runtime.

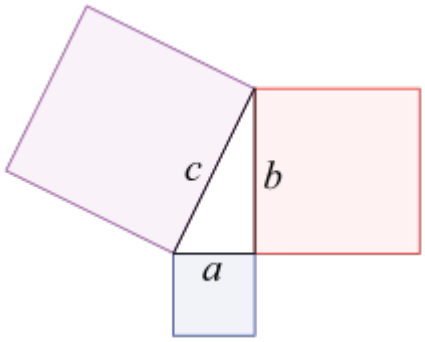
Show/Hide Solution

[8]:
```
print("Input number N:")
N = int(input())
print("Sum of first ", N, " integers: ", N*(N+1)/2)
```

[34] https://docs.python.org/3/library/functions.html#input

```
Input number N:
7
Sum of first  7  integers:  28.0
```

9. Write a small script to compute the length of the hypotenuse (c) of a right triangle having sides a=133 and b=72 units (see picture below). Hint: *remember the Pythagorean theorem and use math.sqrt*).



Show/Hide Solution

[9]:
```
import math

a = 133
b = 72

c = math.sqrt(a**2 + b**2)

print("Hypotenuse: ", c)
```

```
Hypotenuse:  151.23822268196622
```

10. Rewrite the trapezoid script making it compute the area of the trapezoid starting from the major base (MB), minor base (mb) and height (H) taken in input. (Hint: *use the input function and remember to convert the acquired value into an int*).

Show/Hide Solution

[10]:
```
"""trapezoidV2.py"""
MB = int(input("Input the major base (MB):"))
mb = int(input("Input the minor base (mb):"))
H = int(input("Input the height (H):"))
Area = (MB + mb)*H/2
print("Given MB:", str(MB) , " mb:", str(mb) , " and H:", H)
print("The trapezoid area is: ", Area)
```

```
Input the major base (MB):5
Input the minor base (mb):9
Input the height (H):12
Given MB: 5  mb: 9  and H: 12
The trapezoid area is:  84.0
```

11. Write a script that reads the side of an hexagon in input and computes its perimeter and area printing them to the screen. *Hint:* $Area = \frac{3*\sqrt{3}*side^2}{2}$

Show/Hide Solution

[11]:
```
import math

side = int(input("Please insert the side of the hexagon: "))

P = 6*side
A = (3*math.sqrt(3)*side**2)/2
print("Perimeter: ", P, " Area: ", A)
```

```
Please insert the side of the hexagon: 6
Perimeter:  36  Area:  93.53074360871938
```

CHAPTER

THREE

PRACTICAL 2

In this practical we will start interacting more with Python, practicing on how to handle data, functions and methods. We will see several built-in data types and then dive deeper into the data type **string**.

3.1 Slides

The slides of the introduction can be found here: Intro

3.2 Modules

Python modules are simply text files having the extension **.py** (e.g. `exercise.py`). When you were writing the code in the IDE in the previous practical, you were in fact implementing a **module**.

As said in the previous practical, once you implemented and saved the code of the module, you can execute it by typing

```
python3 exercise1.py
```

(which **in Windows** might be `python exercise1.py`, just make sure you are using python 3.x) or, in Visual Studio Code, by right clicking on the code panel and selecting **Run Python File in Terminal**.

A Module A can be loaded from another module B so that B can use the functions defined in A. Remember when we used the `sqrt` function? It is defined in the **module math**. To import it and use it we indeed wrote something like:

[1]:
```
import math

A = math.sqrt(4)
print(A)
```

```
2.0
```

Note

When importing modules we do not need to specify the extension ".py" of the file.

3.3 Objects

Python understands very well objects, and in fact everything is an object in Python.

Objects have **properties** (characteristic features) and **methods** (things they can do). For example, an object *car* could be defined to have the **properties** *model, make, color, number of doors, position* etc., and the **methods** *steer right, steer left, accelerate, break, stop, change gear, repaint,…* whose application might affect the state of the object.

According to Python's official documentation:

"Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects."

All you need to know for now is that in Python objects have an **identifier (ID)** (i.e. their name), a **type** (numbers, text, collections,…) and a **value** (the actual data represented by the objects). Once an object has been created the *identifier* and the *type* never change, while its *value* can either change (**mutable objects**) or stay constant (**immutable objects**).

Python provides the following built-in data types:

Type	Meaning	Domain	Mutable?
`bool`	Condition	`True`, `False`	No
`int`	Integer	$\mathbb{Z}$	No
`float`	Rational	$\mathbb{Q}$ (more or less)	No
`str`	Text	Text	No
`list`	Sequence	Collections of things	Yes
`tuple`	Sequence	Collections of things	No
`dict`	Map	Maps between things	Yes

We will stick with the simplest ones for now, but later on we will dive deeper into all of them.

3.4 Variables

Variables are just **references to objects**, in other words they are the **name** given to an object. Variables can be **assigned** to objects by using the assignment operator =.

The instruction

[2]:
```
sides = 4
```

might represent the number of sides of a square. What happens when we execute it in Python? An object is created, it is given an identifier, its **type** is set to "int" (an integer number), its **value** to 4 and a **name** *sides* is placed in the current namespace to point to that object, so that after that instruction we can access that object through its name. The type of an object can be accessed with the function **type()** and the identifier with the function **id()**:

[3]:
```
sides = 4
print( type(sides) )
print( id(sides) )
```

```
<class 'int'>
10914592
```

Consider now the following code:

[4]:
```
sides = 4 #a square
print ("value:", sides, " type:", type(sides), " id:", id(sides))
sides = 5 #a pentagon
print ("value:", sides, " type:", type(sides), " id:", id(sides))
```

```
value: 4  type: <class 'int'>  id: 10914592
value: 5  type: <class 'int'>  id: 10914624
```

The value of the variable sides has been changed from 4 to 5, but as stated in the table above, the type `int` is **immutable**. **Luckily, this did not prevent us to change the value of sides from 4 to 5.** What happened behind the scenes when we executed the instruction sides = 5 is that a new object has been created of type int (5 is still an integer) and it has been made accessible with the same name *sides*, but since it is a different object (i.e. the integer 5). As a poof of this, **check that the that the identifier printed above is actually different.**

Note: You do not have to really worry about what happens behind the scenes, as the Python interpreter will take care of these aspects for you, but it is nice to know what it does.

You can even change the type of a variable during execution but that is normally a **bad idea** as it makes understanding the code more complicated and leaves more room for errors.

Python allows you to do (**but, please, REFRAIN FROM DOING SO!**):

[5]:
```
sides = 4 #a square
print ("value:", sides, " type:", type(sides), " id:", id(sides))
sides = "four" #the sides in text format
print ("value:", sides, " type:", type(sides), " id:", id(sides))
```

```
value: 4  type: <class 'int'>  id: 10914592
value: four  type: <class 'str'>  id: 140640184741312
```

IMPORTANT NOTE: You can choose the name that you like for your variables (I advise to pick something reminding their meaning), but you need to adhere to some simple rules.

1. Names can only contain upper/lower case digits (`A-Z`, `a-z`), numbers (`0-9`) or underscores _;
2. Names cannot start with a number;
3. Names cannot be equal to reserved keywords:

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

3.5 Numeric types

We already mentioned that numbers are **immutable objects**. Python provides different numeric types: integers, booleans, reals (floats) and even complex numbers and fractions (but we will not get into those).

3.5.1 Integers

Their range of values is limited only by the memory available. As we have already seen, python provides also a set of standard operators to work with numbers:

[6]:
```
a = 7
b = 4

print(a + b) # 11
print(a - b) # 3
print(a // b) # integer division: 1
print(a * b) # 28
print(a ** b) # power: 2401
print(a / b) # division 1.75
print(type(a / b))
```

```
11
3
1
28
2401
1.75
<class 'float'>
```

Note that in the latter case the result is no more an integer, but a float (we will get to that later).

3.5.2 Booleans

These objects are used for the boolean algebra. Truth values are represented with the keywords `True` and `False` in Python. A boolean object can only have value `True` or `False`. We can convert booleans into integers with the builtin function `int`. Any integer can be converted into a boolean (and vice-versa) with:

[7]:
```
a = bool(1)
b = bool(0)
c = bool(72)
d = bool(-5)
t = int(True)
f = int(False)

print("a: ", a, " b: ", b, " c: ", c, " d: ", d , " t: ", t, " f: ", f)
```

```
a:  True  b:  False  c:  True  d:  True  t:  1  f:  0
```

any integer is evaluated to true, except 0. Note that, the truth values `True` and `False` respectively behave like the integers 1 and 0.

We can operate on boolean values with the boolean operators `and`, `or`, `not`. Recall boolean algebra for their use:

[8]:
```
T = True
F = False
```

(continues on next page)

(continued from previous page)

```
print ("T: ", T, " F:", F)

print ("T and F: ", T and F) #False
print ("T and T: ", T and T) #True
print ("F and F: ", F and F) #False
print ("not T: ", not T) # False
print ("not F: ", not F) # True
print ("T or F: ", T or F) # True
print ("T or T: ", T or T) # True
print ("F or F: ", F or F) # False
```

```
T:  True  F: False
T and F:  False
T and T:  True
F and F:  False
not T:  False
not F:  True
T or F:  True
T or T:  True
F or F:  False
```

Numeric comparators are operators that return a boolean value. Here are some examples:

`a == b`	True if and only if $a = b$
`a != b`	True if and only if $a \neq b$
`a < b`	True if and only if $a < b$
`a > b`	True if and only if $a > b$
`a <= b`	True if and only if $a \leq b$
`a >= b`	True if and only if $a \geq b$

Example: Given a variable a = 10 and a variable b = 77, let's swap their values (i.e. at the end a will be equal to 77 and b to 10). Let's also check the values at the beginning and at the end.

[9]:
```
a = 10
b =  77
print("a: ", a, " b:", b)
print("is a equal to 10?", a == 10)
print("is b equal to 77?", b == 77)

TMP = b  #we need to store the value of b safely
b = a    #ok, the old value of b is gone... is it?
a = TMP  #a gets the old value of b... :-)

print("a: ", a, " b:", b)
print("is a equal to 10?", a == 10)
print("is a equal to 77?", a == 77)
print("is b equal to 10?", b == 10)
print("is b equal to 77?", b == 77)
```

```
a:  10  b: 77
is a equal to 10? True
is b equal to 77? True
a:  77  b: 10
is a equal to 10? False
is a equal to 77? True
is b equal to 10? True
is b equal to 77? False
```

3.5.3 Real numbers

Python stores real numbers (floating point numbers) in 64 bits of information divided in sign, exponent and mantissa.

Example: Let's calculate the area of the center circle of a football pitch (radius = 9.15m) recalling that $area = \Pi * R^2$:

[10]:
```
R = 9.15
Pi = 3.141592653589793
Area = Pi*(R**2)
print (Area)
```

```
263.02199094017146
```

Note that the builtin math module of python contains the definition of Π, therefore we could rewrite the code above as:

[11]:
```
import math
R = 9.15
Pi = math.pi
Area = Pi*(R**2)
print (Area)
```

```
263.02199094017146
```

Note that the parenthesis around the `R**2` are not necessary as the operator `**` has the precedence, but I personally think it helps readability.

Here is a reminder of the precedence of operators:

`**`	Power (Highest precedence)
`+,-`	Unary plus and minus
`* / // %`	Multiply, divide, floor division, modulo
`+ -`	Addition and subtraction
`<= < > >=`	Comparison operators
`== !=`	Equality operators
`not or and`	Logical operators (Lowest precedence)

Example: Let's compute the GC content of a DNA sequence 33 base pairs long, having 12 As, 9 Ts, 5 Cs and 7Gs. The GC content can be expressed by the formula: $gc = \frac{G+C}{A+T+C+G}$ where A,T,C,G represent the number of nucleotides of each kind. What is the AT content? Is the GC content higher than the AT content?

[12]:
```
A = 12
T = 9
C = 5
```

(continues on next page)

(continued from previous page)

```
G = 7

gc = (G+C)/(A+T+C+G)
print("The GC content is: ", gc)

at = 1 - gc

print("The AT content is: ", at)

print (gc > at)
```

```
The GC content is:  0.36363636363636365
The AT content is:  0.6363636363636364
False
```

3.6 Strings

Strings are **immutable objects** (note the actual type is **str**) used by python to handle text data. Strings are sequences of *unicode code points* that can represent characters, but also formatting information (e.g. '\n' for new line). **Unlike other programming languages, python does not have the data type character, which is represented as a string of length 1**.

There are several ways to define a string:

[13]:
```
S = "my first string, in double quotes"

S1 = 'my second string, in single quotes'

S2 = '''my third string is
in triple quotes
therefore it can span several lines'''

S3 = """my fourth string, in triple double-quotes
can also span
several lines"""

print(S, '\n') #let's add a new line at the end of the string with \n
print(S1,'\n')
print(S2, '\n')
print(S3, '\n')
```

```
my first string, in double quotes

my second string, in single quotes

my third string is
in triple quotes
therefore it can span several lines

my fourth string, in triple double-quotes
can also span
several lines

```

To put special characters like '," and so on you need to "escape them" (i.e. write them following a back-slash).

\\	Backslash
\n	ASCII linefeed (also known as newline)
\t	ASCII tab character
\'	Single quote
\"	Double quote
\xxxx	Unicode character xxxx (hexadecimal)

Example: Let's print a string containing a quote and double quote (i.e. ' and ").

[14]:
```
myString = "This is how I \'quote\' and \"double quote\" things in strings"
print(myString)
```

```
This is how I 'quote' and "double quote" things in strings
```

Strings can be converted to and from numbers with the functions `str()`, `int()` or `float()`.

Example: Let's define a string *myString* with the value "47001" and convert it into an int. Try adding one and print the result.

[15]:
```
my_string = "47001"
print(my_string, " has type ", type(my_string))

my_int = int(my_string)

print(my_int, " has type ", type(my_int))

my_int = my_int + 1    #adds one

my_string = my_string + "1" #cannot add 1 (we need to use a string).
                            #This will append 1 at the end of the string

print(my_int)
print(my_string)
```

```
47001  has type  <class 'str'>
47001  has type  <class 'int'>
47002
470011
```

Be careful though that if the string cannot be converted into an integer, then you get an error

[16]:
```
my_wrong_number = "13a"

N = int(my_wrong_number)

print(N)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-bcfe98c1ea66> in <module>
      1 my_wrong_number = "13a"
      2
----> 3 N = int(my_wrong_number)
      4
      5 print(N)

ValueError: invalid literal for int() with base 10: '13a'
```

Python defines some operators to work with strings. Recall the slides shown during the lecture:

Result	Operator	Meaning
int	len(str)	Return the length of the string
str	str + str	Concatenate two strings
str	str * int	Replicate the string
bool	str in str	Check if a string is present in another string
str	str[int]	Read the character at specified index
str	str[int:int]	Extract a sub-string

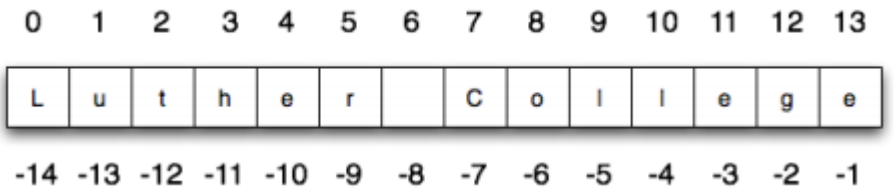
Example A tandem repeat is a short sequence of DNA that is repeated several times in a row. Let's create a string representing the tandem repeat of the motif "ATTCG" repeated 5 times. What is the length of the whole repetitive region? Is the motif "TCGAT" (m1) present in the region? The motif "TCCT" (m2)? Let's give an orientation to the tandem repeat by adding the string "5'-" (5' end) on the left and "-3'" (3' end) to the right.

[17]:
```
motif = "ATTCG"

tandem_repeat = motif * 5

print(motif)
print(tandem_repeat, " has length", len(tandem_repeat))
m1 = "TCGAT"
m2 = "TCCT"

print("Is ", m1, " in ", tandem_repeat, " ? ", m1 in tandem_repeat )
print("Is ", m2, " in ", tandem_repeat, " ? ", m2 in tandem_repeat )
oriented_tr = "5\'-" + tandem_repeat + "-3\'"
print(oriented_tr)
```

```
ATTCG
ATTCGATTCGATTCGATTCGATTCG  has length 25
Is  TCGAT  in  ATTCGATTCGATTCGATTCGATTCG  ?  True
Is  TCCT  in  ATTCGATTCGATTCGATTCGATTCG  ?  False
5'-ATTCGATTCGATTCGATTCGATTCG-3'
```

We can access strings at specific positions (indexing) or get a substring starting from a position S to a position E. The only thing to remember is that numbering starts from 0. The `i`-th character of a string can be accessed as `str[i-1]`. Substrings can be accessed as `str[S:E]`, optionally a third parameter can be specified to set the step (i.e. `str[S:E:STEP]`).

Important note. Remember that when you do str[S:E], **S is inclusive, while E is exclusive** (see S[0:6] below).



Let's see these aspects in action with an example:

[18]:
```
S = "Luther College"
```

(continues on next page)

(continued from previous page)

```
print(S) #print the whole string
print(S == S[:]) #a fancy way of making a copy of the original string
print(S[0]) #first character
print(S[3]) #fourth character
print(S[-1]) #last character
print(S[0:6]) #first six characters
print(S[-7:]) #final seven characters
print(S[0:len(S):2]) #every other character starting from the first
print(S[1:len(S):2]) #every other character starting from the second
```

```
Luther College
True
L
h
e
Luther
College
Lte olg
uhrClee
```

3.6.1 Methods for the str object

The object `str` has some methods that can be applied to it (remember methods are things you can do on objects). Recall from the lecture that the main methods are:

Result	Method	Meaning
`str`	`str.upper()`	Return the string in upper case
`str`	`str.lower()`	Return the string in lower case
`str`	`str.strip(str)`	Remove strings from the sides
`str`	`str.lstrip(str)`	Remove strings from the left
`str`	`str.rstrip(str)`	Remove strings from the right
`str`	`str.replace(str, str)`	Replace substrings
`bool`	`str.startswith(str)`	Check if the string starts with another
`bool`	`str.endswith(str)`	Check if the string ends with another
`int`	`str.find(str)`	Return the first position of a substring starting from the left
`int`	`str.rfind(str)`	Return the position of a substring starting from the right
`int`	`str.count(str)`	Count the number of occurrences of a substring

IMPORTANT NOTE: Since Strings are immutable, every operation that changes the string actually produces a new *str* object having the modified string as value.

Moreover, since **strings are immutable** we cannot directly change them with an assignment operator.

Example: Since the genetic code is degenerate, there are many codons encoding for the same aminoacid. Consider Proline, it can be encoded by the following codons: CCU, CCA,CCG, CCC. Let's create a string proline and assign it to its possible codons one after the other.

[19]:
```
"""
Wrong solution. We cannot directly replace the value of a string
"""

proline = "CCU"
print("Proline can be encoded by: ", proline)
proline[2]="A"
print(".. or by: ", proline)
```

```
Proline can be encoded by:  CCU
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-9750dcfa1cbd> in <module>
      5 proline = "CCU"
      6 print("Proline can be encoded by: ", proline)
----> 7 proline[2]="A"
      8 print(".. or by: ", proline)
      9

TypeError: 'str' object does not support item assignment
```

[20]:
```
"""
Correct solution. Using str.replace
"""
proline = "CCU"
print("Proline can be encoded by: ", proline)
proline = proline.replace("U","A")
print(".. or by: ", proline)
proline = proline.replace("A","G")
print(".. or by: ", proline)
proline = proline.replace("G","C")
print(".. or by: ", proline)
```

```
Proline can be encoded by:  CCU
.. or by:  CCA
.. or by:  CCG
.. or by:  CCC
```

[21]:
```
"""
Another correct solution. Using string slicing and catenation.
"""
proline = "CCU"
print("Proline can be encoded by: ", proline)
proline = proline[:-1]+"A" #equal to proline[0:-1] or proline[0:2]
print(".. or by: ", proline)
proline = proline[:-1]+"G"
print(".. or by: ", proline)
proline = proline[:-1]+"C"
print(".. or by: ", proline)
```

```
Proline can be encoded by:  CCU
.. or by:  CCA
.. or by:  CCG
.. or by:  CCC
```

Example: Given the DNA sequence S = ” aTATGCCCATatcgctAAATTGCTGCCATTACA “. Print its length (remov-

ing any blank spaces at either sides), the number of adenines, cytosines, guanines and thymines present. Is the sequence "ATCG" present in S? Print how many times the substring "TGCC" appears in S and all the corresponding indexes.

[22]:
```
S = "   aTATGCCCATatcgctAAATTGCTGCCATTACA    "

print(S)
S = S.strip(" ")
print(S)

print(len(S))
tmp_s = S.upper() #for simplicity to count only 4 different nucleotides
print("A count: ", tmp_s.count("A"))
print("C count: ", tmp_s.count("C"))
print("T count: ", tmp_s.count("T"))
print("G count: ", tmp_s.count("G"))
print("Is ATCG in ", tmp_s, "? ", tmp_s.find("ATCG") != -1) #or tmp_s.count("ATCG") >␣
↪0
print("TGCC is present ", tmp_s.count("TGCC"), " times in ", tmp_s)
print("TGCC is present at pos ", tmp_s.find("TGCC"))
print("TGCC is present at pos ", tmp_s.rfind("TGCC")) #or tmp_S.find("TGCC",4)
```

```
   aTATGCCCATatcgctAAATTGCTGCCATTACA
aTATGCCCATatcgctAAATTGCTGCCATTACA
33
A count:  10
C count:  9
T count:  10
G count:  4
Is ATCG in  ATATGCCCATATCGCTAAATTGCTGCCATTACA ?  True
TGCC is present  2  times in  ATATGCCCATATCGCTAAATTGCTGCCATTACA
TGCC is present at pos  3
TGCC is present at pos  23
```

3.7 Exercises

1. Given the following string on two lines:

```
text = """Nobody said it was easy
No one ever said it would be this hard"""
```

write some python code that a)prints the whole string; b) prints the first and last character; c) prints the first 10 characters; d) prints from the 19th character to the 31st; e) prints the string all in capital letters.

Show/Hide Solution

[34]:
```
text = """Nobody said it was easy
No one ever said it would be this hard"""
# a) prints the whole text
print(text)


#some empty space...
print("")

# b) 1st and last character
```

(continues on next page)

(continued from previous page)

```
print("1st char: ", text[0], " last char: ", text[-1])

#c) the 1st 10 characters:
print("1st 10 chars:", text[0:10])

#d) from the 19th to the 31st char
print("\nCharacters from 19 to 31:")
print(text[18:31])
print("\nAll upper case:")
upper_text = text.upper()
print(upper_text) #equivalent to: print(text.upper())
print("")
#NOTE THAT:
print(text)
print("")
#is different from
print(upper_text)
print("")
#as confirmed by python:
print("text and upper_text are equal: ", text == upper_text)

print("")
print("Newline? ", "\n" in text)
```

```
Nobody said it was easy
No one ever said it would be this hard

1st char:  N  last char:  d
1st 10 chars: Nobody sai

Characters from 19 to 31:
 easy
No one

All upper case:
NOBODY SAID IT WAS EASY
NO ONE EVER SAID IT WOULD BE THIS HARD

Nobody said it was easy
No one ever said it would be this hard

NOBODY SAID IT WAS EASY
NO ONE EVER SAID IT WOULD BE THIS HARD

text and upper_text are equal:  False

Newline?  True
```

2. An exon of a gene starts from position 12030 on a genome and ends at position 12174. Does an A/T SNP present at position 12111 affect this exon (i.e. is it inside the exon)? And what about a SNP present at position 12188? *Hint: create a suitable boolean expression to check if the positions are within the interval of the exon*.

Show/Hide Solution

[24]:
```
E_start = 12030
E_end = 12174
SNP1_pos = 12111
```

(continues on next page)

(continued from previous page)

```
SNP2_pos = 12188

Test1 = (SNP1_pos >= E_start and SNP1_pos <= E_end)
Test2 = (SNP2_pos >= E_start and SNP2_pos <= E_end)
print ("SNP1 (", SNP1_pos,") in [", E_start, ",", E_end, "]?", Test1)
print ("SNP2 (", SNP2_pos,") in [", E_start, ",", E_end, "]?", Test2)
```

```
SNP1 ( 12111 ) in [ 12030 , 12174 ]? True
SNP2 ( 12188 ) in [ 12030 , 12174 ]? False
```

3. SNP FB_AFFY_0000024 of the Apple 480K SNP chip has 5' flanking region (i.e. the forward probe) CATTATTTTCACTTGGGTCGAGGCCAGATTCCATC and 3' flanking region (i.e. the reverse probe) GGATTGCCCCGAAATCAGAGAAAAGTCG. The SNP is a G/A transversion. Answer the following questions:
    1. What is the length of the 5' flanking region? And that of the 3' flanking region?
    2. The IUPAC code of the G/A transversion is R. What is the sequence of the whole region using the "[G/A]" notation for the SNP (hint: concatenate it in a new string called *region*) and the iupac notation R (region_iupac)?
    3. Retrive and print only the SNP from *region* and *iupac_region*

Show/Hide Solution

[25]:
```
SNP_5prime =  "CATTATTTTCACTTGGGTCGAGGCCAGATTCCATC"
SNP_3prime = "GGATTGCCCGAAATCAGAGAAAAGTCG"

SNPseq = "G/A"
SNPiupac = "R"

print("Length of 5' end: ", len(SNP_5prime))
print("Length of 3' end: ", len(SNP_3prime))
region = SNP_5prime + "[" + SNPseq + "]" + SNP_3prime
region_iupac = SNP_5prime + SNPiupac + SNP_3prime
print(region)
print(region_iupac)

#string slicing and indexing!

snp_from_region = region[ len(SNP_5prime) + 1 : len(SNP_5prime) + 4 ]
snp_from_iupac = region_iupac[ len(SNP_5prime) ]

print("SNP from region: ", snp_from_region)
print("SNP from iupac region: ", snp_from_iupac)

# Another way:
#L_ind = region.find("[")
#R_ind = region.find("]")
#print(L_ind)
#print(R_ind)

#print(region[L_ind + 1 : R_ind])
```

```
Length of 5' end:  35
Length of 3' end:  27
CATTATTTTCACTTGGGTCGAGGCCAGATTCCATC[G/A]GGATTGCCCGAAATCAGAGAAAAGTCG
CATTATTTTCACTTGGGTCGAGGCCAGATTCCATCRGGATTGCCCGAAATCAGAGAAAAGTCG
```

(continues on next page)

(continued from previous page)

```
SNP from region:  G/A
SNP from iupac region:  R
```

4. Compute the melting temperature T_m of the primer with sequence "TTAGCACACGTGAGCCAATGGAGCAAACGGGTAATT". The melting temperature T_m (in degrees Celtius) can be computed as: $T_m = 64.9 + 41(GC - 16.4)/N$, where GC is the total number of G and C in the primer and N is its length.

Show/Hide Solution

[26]:
```
primer = "TTAGCACACGTGAGCCAATGGAGCAAACGGGTAATT"
N = len(primer)

gc = (primer.count("G") + primer.count("C"))

Tm = 64.9 + 41 * (gc - 16.4)/N

print("The melting T for primer ", primer, " is: ", Tm, "°C")
```

```
The melting T for primer  TTAGCACACGTGAGCCAATGGAGCAAACGGGTAATT  is:  65.
↪58333333333334 °C
```

5. The spike protein of the Sars-CoV-2 virus has the following aminoacidic sequence:

```
S = """
MFVFLVLLPLVSSQCVNLTTRTQLPPAYTNSFTRGVYYPDKVFRSSVLHSTQDLFLPFFS
NVTWFHAIHVSGTNGTKRFDNPVLPFNDGVYFASTEKSNIIRGWIFGTTLDSKTQSLLIV
NNATNVVIKVCEFQFCNDPFLGVYYHKNNKSWMESEFRVYSSANNCTFEYVSQPFLMDLE
GKQGNFKNLREFVFKNIDGYFKIYSKHTPINLVRDLPQGFSALEPLVDLPIGINITRFQT
LLALHRSYLTPGDSSSGWTAGAAAYYVGYLQPRTFLLKYNENGTITDAVDCALDPLSETK
CTLKSFTVEKGIYQTSNFRVQPTESIVRFPNITNLCPFGEVFNATRFASVYAWNRKRISN
CVADYSVLYNSASFSTFKCYGVSPTKLNDLCFTNVYADSFVIRGDEVRQIAPGQTGKIAD
YNYKLPDDFTGCVIAWNSNNLDSKVGGNYNYLYRLFRKSNLKPFERDISTEIYQAGSTPC
NGVEGFNCYFPLQSYGFQPTNGVGYQPYRVVVLSFELLHAPATVCGPKKSTNLVKNKCVN
FNFNGLTGTGVLTESNKKFLPFQQFGRDIADTTDAVRDPQTLEILDITPCSFGGVSVITP
GTNTSNQVAVLYQDVNCTEVPVAIHADQLTPTWRVYSTGSNVFQTRAGCLIGAEHVNNSY
ECDIPIGAGICASYQTQTNSPRRARSVASQSIIAYTMSLGAENSVAYSNNSIAIPTNFTI
SVTTEILPVSMTKTSVDCTMYICGDSTECSNLLLQYGSFCTQLNRALTGIAVEQDKNTQE
VFAQVKQIYKTPPIKDFGGFNFSQILPDPSKPSKRSFIEDLLFNKVTLADAGFIKQYGDC
LGDIAARDLICAQKFNGLTVLPPLLTDEMIAQYTSALLAGTITSGWTFGAGAALQIPFAM
QMAYRFNGIGVTQNVLYENQKLIANQFNSAIGKIQDSLSSTASALGKLQDVVNQNAQALN
TLVKQLSSNFGAISSVLNDILSRLDKVEAEVQIDRLITGRLQSLQTYVTQQLIRAAEIRA
SANLAATKMSECVLGQSKRVDFCGKGYHLMSFPQSAPHGVVFLHVTYVPAQEKNFTTAPA
ICHDGKAHFPREGVFVSNGTHWFVTQRNFYEPQIITTDNTFVSGNCDVVIGIVNNTVYDP
LQPELDSFKEELDKYFKNHTSPDVDLGDISGINASVVNIQKEIDRLNEVAKNLNESLIDL
QELGKYEQYIKWPWYIWLGFIAGLIAIVMVTIMLCCMTSCCSCLKGCCSCGSCCKFDEDD
SEPVLKGVKLHYT
"""
```

Write a little python script to answer the following questions: 1) What are the first 10 and the last 10 aminoacids? 2) How many aminoacids does it have (beware of new lines)? 3) How many Tyrosines (T) does it contain? 4) How many Triptophanes (W)? 5) How many Valines (V) followed by at least one Lysine (K)?

Show/Hide Solution

[27]:
```
S = """
MFVFLVLLPLVSSQCVNLTTRTQLPPAYTNSFTRGVYYPDKVFRSSVLHSTQDLFLPFFS
NVTWFHAIHVSGTNGTKRFDNPVLPFNDGVYFASTEKSNIIRGWIFGTTLDSKTQSLLIV
```

(continues on next page)

(continued from previous page)

```
NNATNVVIKVCEFQFCNDPFLGVYYHKNNKSWMESEFRVYSSANNCTFEYVSQPFLMDLE
GKQGNFKNLREFVFKNIDGYFKIYSKHTPINLVRDLPQGFSALEPLVDLPIGINITRFQT
LLALHRSYLTPGDSSSGWTAGAAAYYVGYLQPRTFLLKYNENGTITDAVDCALDPLSETK
CTLKSFTVEKGIYQTSNFRVQPTESIVRFPNITNLCPFGEVFNATRFASVYAWNRKRISN
CVADYSVLYNSASFSTFKCYGVSPTKLNDLCFTNVYADSFVIRGDEVRQIAPGQTGKIAD
YNYKLPDDFTGCVIAWNSNNLDSKVGGNYNYLYRLFRKSNLKPFERDISTEIYQAGSTPC
NGVEGFNCYFPLQSYGFQPTNGVGYQPYRVVVLSFELLHAPATVCGPKKSTNLVKNKCVN
FNFNGLTGTGVLTESNKKFLPFQQFGRDIADTTDAVRDPQTLEILDITPCSFGGVSVITP
GTNTSNQVAVLYQDVNCTEVPVAIHADQLTPTWRVYSTGSNVFQTRAGCLIGAEHVNNSY
ECDIPIGAGICASYQTQTNSPRRARSVASQSIIAYTMSLGAENSVAYSNNSIAIPTNFTI
SVTTEILPVSMTKTSVDCTMYICGDSTECSNLLLQYGSFCTQLNRALTGIAVEQDKNTQE
VFAQVKQIYKTPPIKDFGGFNFSQILPDPSKPSKRSFIEDLLFNKVTLADAGFIKQYGDC
LGDIAARDLICAQKFNGLTVLPPLLTDEMIAQYTSALLAGTITSGWTFGAGAALQIPFAM
QMAYRFNGIGVTQNVLYENQKLIANQFNSAIGKIQDSLSSTASALGKLQDVVNQNAQALN
TLVKQLSSNFGAISSVLNDILSRLDKVEAEVQIDRLITGRLQSLQTYVTQQLIRAAEIRA
SANLAATKMSECVLGQSKRVDFCGKGYHLMSFPQSAPHGVVFLHVTYVPAQEKNFTTAPA
ICHDGKAHFPREGVFVSNGTHWFVTQRNFYEPQIITTDNTFVSGNCDVVIGIVNNTVYDP
LQPELDSFKEELDKYFKNHTSPDVDLGDISGINASVVNIQKEIDRLNEVAKNLNESLIDL
QELGKYEQYIKWPWYIWLGFIAGLIAIVMVTIMLCCMTSCCSCLKGCCSCGSCCKFDEDD
SEPVLKGVKLHYT
"""

#Let's remove the newline character (\n)
S = S.replace('\n','')

#0. The first 5 and last 5 aminoacids:
print("The S protein: ", S[0:10] , "... " + S[-10:])
#1. How many aminoacids does the sequence have?
print("The S protein contains " + str(len(S)) + " aminoacids...")

#2. How many of these are T?
print("... " +  str(S.count("T")) + " of which are Tyrosines")

#3. How many of these are W?
print("... " +  str(S.count("W")) + " of which are Triptophanes")

#4. How many of these are VK?
print("... " +  str(S.count("VK")) + " VKs")
```

```
The S protein:  MFVFLVLLPL … VLKGVKLHYT
The S protein contains 1273 aminoacids…
… 97 of which are Tyrosines
… 12 of which are Triptophanes
… 4 VKs
```

6. Convert the following extract of the PalB2[35] gene into mRNA (i.e. replace thymine with uracile):

```
seq ="""CTGTCTCCCTCACTGTATGTAAATTGCATCTAGAATAGCA
TCTGGAGCACTAATTGACACATAGTGGGTATCAATTATTA
TTCCAGGTACTAGAGATACCTGGACCATTAACGGATAAAT
AGAAGATTCATTTGTTGAGTGACTGAGGATGGCAGTTCCT
GCTACCTTCAAGGATCTGGATGATGGGGAGAAACAGAGAA
CATAGTGTGAGAATACTGTGGTAAGGAAAGTACAGAGGAC
TGGTAGAGTGTCTAACCTAGATTTGGAGAAGGACCTAGAA
```

(continues on next page)

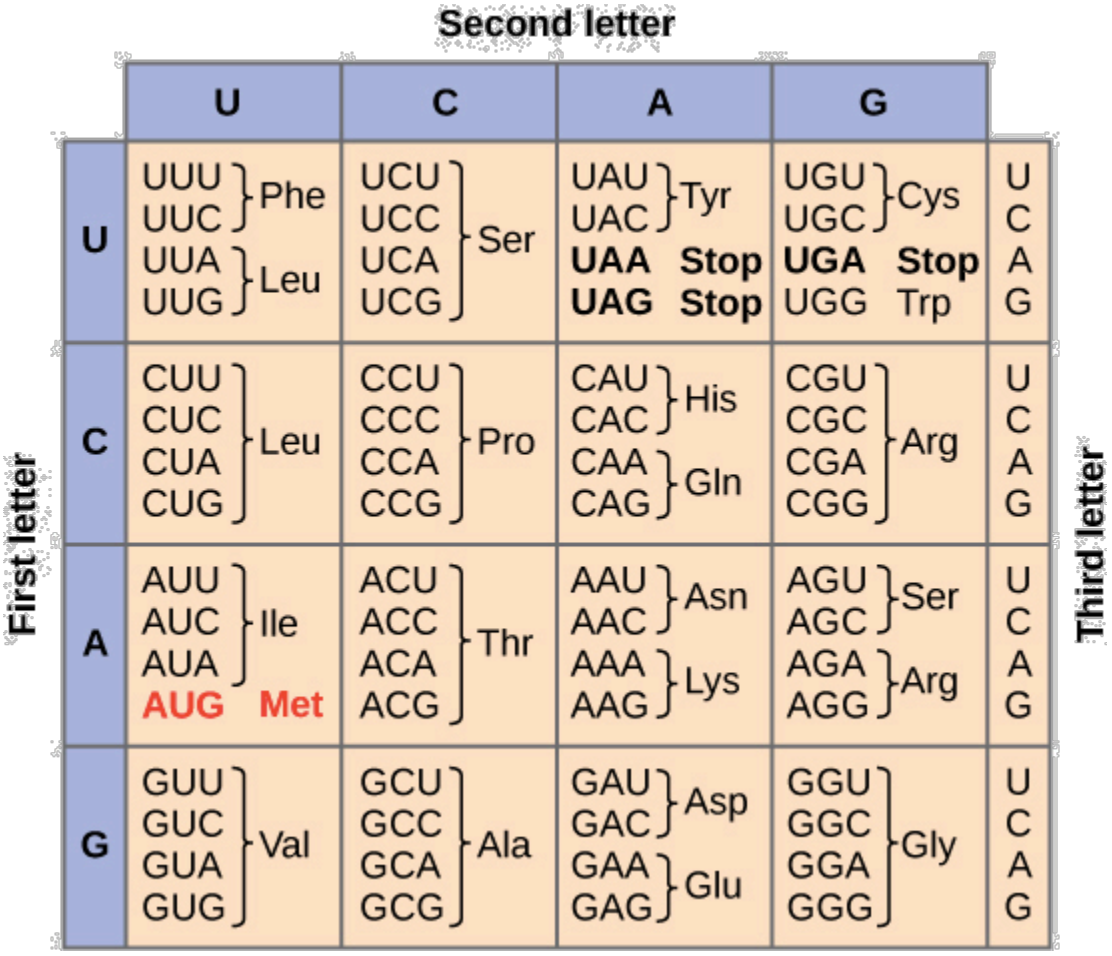
[35] http://www.ensembl.org/Homo_sapiens/Gene/Summary?g=ENSG00000083093;r=16:23603160-23641310

(continued from previous page)

```
GTCTATCCCAGGGAAATAAAAATCTAAGCTAAGGTTTGAG
GAATCAGTAGGAATTGGCAAAGGAAGGACATGTTCCAGAT
GATAGGAACAGGTTATGCAAAGATCCTGAAATGGTCAGAG
CTTGGTGCTTTTTGAGAACCAAAAGTAGATTGTTATGGAC
CAGTGCTACTCCCTGCCTCTTGCCAAGGGACCCCGCCAAG
CACTGCATCCCTTCCCTCTGACTCCACCTTTCCACTTGCC
CAGTATTGTTGGTGT"""
```

and print the number of uracils present and the total length of the sequence (**remember to remove newlines**).

Considering the genetic code and all the possible open reading frames, answer the following questions:



1. How many stop codons are present in the sequence?
2. How many Glycines (Gly)?
3. Is Tryptophane (Trp) present?
4. What is the position of the leftmost Trp? Print the codon to double check correctness (hint: slicing).
5. What is the position of the rightmost Trp? Print the codon to double check correctness (hint: slicing).

Show/Hide Solution

[28]:
```
seq ="""CTGTCTCCCTCACTGTATGTAAATTGCATCTAGAATAGCA
TCTGGAGCACTAATTGACACATAGTGGGTATCAATTATTA
```

(continues on next page)

(continued from previous page)

```
TTCCAGGTACTAGAGATACCTGGACCATTAACGGATAAAT
AGAAGATTCATTTGTTGAGTGACTGAGGATGGCAGTTCCT
GCTACCTTCAAGGATCTGGATGATGGGGAGAAACAGAGAA
CATAGTGTGAGAATACTGTGGTAAGGAAAGTACAGAGGAC
TGGTAGAGTGTCTAACCTAGATTTGGAGAAGGACCTAGAA
GTCTATCCCAGGGAAATAAAAATCTAAGCTAAGGTTTGAG
GAATCAGTAGGAATTGGCAAAGGAAGGACATGTTCCAGAT
GATAGGAACAGGTTATGCAAAGATCCTGAAATGGTCAGAG
CTTGGTGCTTTTTGAGAACCAAAAGTAGATTGTTATGGAC
CAGTGCTACTCCCTGCCTCTTGCCAAGGGACCCCGCCAAG
CACTGCATCCCTTCCCTCTGACTCCACCTTTCCACTTGCC
CAGTATTGTTGGTGT"""

seq = seq.replace("\n","")
mRNA = seq.replace("T","U")

print("Number of uracils: ", mRNA.count("U"))
print("Total length of the sequence: ", len(seq))
stopc = mRNA.count("UAA") + mRNA.count("UGA") + mRNA.count("UAG")
print("Number of stop codons: ", stopc)
gly = mRNA.count("GGU") + mRNA.count("GGC") + mRNA.count("GGA") + mRNA.count("GGG")
print("Number of glycines: ", gly)
print("Is Trp present? ", mRNA.find("UGG")> 0)
rmTrp = mRNA.find("UGG")
print("Leftmost Trp at pos:", rmTrp, " Codon: ", mRNA[rmTrp : rmTrp + 3])
lmTrp = mRNA.rfind("UGG")
print("Rightmost Trp at pos:", mRNA.rfind("UGG"), " Codon: ", mRNA[lmTrp:lmTrp+3])
```

```
Number of uracils:  140
Total length of the sequence:  535
Number of stop codons:  32
Number of glycines:  34
Is Trp present?  True
Leftmost Trp at pos: 42  Codon:  UGG
Rightmost Trp at pos: 529  Codon:  UGG
```

7. Consider the following Illumina HiSeq 4000 read:

```
read = """AATGATACGGCGACCACCGAGATCTACACGCCTCCCTCGCGC
CATCAGAGAGTCTGGGTCTCAGGTACCGCAGTTGTATCTTGCGCGACTATA
ATCCACGGCTCTTATTCTAGCGTGCGCGTACGGCGGTGGGCGTCGTTACGCTATATT"""
```

and try to answer the following questions:

```
1. How long is the read (beware of newlines)?
2. What is the GC content of the read (remember $gc = \frac{G+C}{A+T+C+G}$)?
3. A Nextera adapter is "AATGATACGGCGACCACCGAGATCTACACGCCTCCCTCGCGCCATCAG".
Is it present in the read? How long is it?
4. Remove the Nextera adapter from the read and recompute the GC content.
Has GC content increased after adapter trimmming?
```

Show/Hide Solution

[29]:
```
read = """AATGATACGGCGACCACCGAGATCTACACGCCTCCCTCGCGC
CATCAGAGAGTCTGGGTCTCAGGTACCGCAGTTGTATCTTGCGCGACTATA
ATCCACGGCTCTTATTCTAGCGTGCGCGTACGGCGGTGGGCGTCGTTACGCTATATT"""
```

(continues on next page)

(continued from previous page)

```
read = read.replace("\n","")
print("Read length is: ", len(read), " base pairs")
g = read.count("G")
c = read.count("C")
t = read.count("T")
a = read.count("A")

gc = (g + c) / (a + t + c + g)

print("GC content of read: ", gc)

adapter = "AATGATACGGCGACCACCGAGATCTACACGCCTCCCTCGCGCCATCAG"

print("Is the adapter present? ", adapter in read)
print("Adapter length: ", len(adapter))
print("The adapter starts at: ", read.find(adapter))

trimmed_read = read.replace(adapter,"")

tr_g = trimmed_read.count("G")
tr_c = trimmed_read.count("C")
tr_t = trimmed_read.count("T")
tr_a = trimmed_read.count("A")

tr_gc = (tr_g + tr_c) / (tr_a + tr_t + tr_c + tr_g)
print("GC content of trimmed read: ", tr_gc)
print("GC content has increased after trimming: ", tr_gc > gc)
```

```
Read length is:  150  base pairs
GC content of read:  0.56
Is the adapter present?  True
Adapter length:  48
The adapter starts at:  0
GC content of trimmed read:  0.5392156862745098
GC content has increased after trimming:  False
```

8. Given *geneA* starting at position 1000 and ending at position 3400, and *geneB* starting at position 3700 and ending at position 6000. Randomly select a position (*pos*) from 1 to 5202 and check the following: a. is pos in geneA? b. is pos in geneB? c. is pos in between the two genes? d. is pos within one of the two genes? e. is pos outside both genes? f. is pos within 100 bases before the start of geneA? To pick a random number you can import the random module and use the random.randint(start,end) function:

   ```
   import random

   pos = random.randint(1,6000)
   ```

Show/Hide Solution

[30]:
```
import random

geneA_start = 1000
geneA_end = 3400
geneB_start = 3700
geneB_end = 5201

pos = random.randint(1,6000)
print("Random position is: ", pos)
```

(continues on next page)

(continued from previous page)

```
answerA = (pos >= geneA_start and pos <= geneA_end)
answerB = (pos >= geneB_start and pos <= geneB_end)
answerC = (pos >geneA_end and pos <geneB_start)
answerD = (answerA or answerB)
answerE = (pos < geneA_start or (pos > geneA_end and pos < geneB_start) or (pos >␣
↪geneB_end))
answerF = (pos >= geneA_start - 100 ) and (pos < geneA_start)
print("Is ", pos, " in geneA [", geneA_start, ",", geneA_end, "]? ", answerA)
print("Is ", pos, " in geneB [", geneB_start, ",", geneB_end, "]? ", answerB)
print("Is ", pos, " between the two genes? ", answerC)
print("Is ", pos, " in one of the two genes? ", answerD)
print("Is ", pos, " outside of both genes? ", answerE)
print("Is ", pos, " within 100 bases from the start of geneA? ", answerF)
```

```
Random position is:  5701
Is  5701  in geneA [ 1000 , 3400 ]?  False
Is  5701  in geneB [ 3700 , 5201 ]?  False
Is  5701  between the two genes?  False
Is  5701  in one of the two genes?  False
Is  5701  outside of both genes?  True
Is  5701  within 100 bases from the start of geneA?  False
```

9. The DNA-binding domain of the Tumor Suppressor Protein TP53 can be represented by the string:

```
chain_a = """SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
EPHHELPPGSTKRALPNNT"""
```

Answer the following questions:

```
1. How many lines is the sequence written on?
2. How long is the sequence (remove newlines)?
3. Create a new sequence with all new lines removed
4. How many cysteines "C" and histidines "H" are there in the sequence?
5. Does the chain contain the sub-sequence "NLRVEYLDDRN"? Where?
6. Extract the first line of the sequence (Hint: use find and string slicing).
```

Show/Hide Solution

[31]:
```
chain_a = """SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
EPHHELPPGSTKRALPNNT"""

print(chain_a)
lines = chain_a.count('\n') + 1
print("The sequence is in ", lines, " lines")

sequence = chain_a.replace("\n","")
print("The sequence has ", len(sequence), " aminoacids")
print("The sequence counts ", sequence.count('C'), " cysteins")
```

(continues on next page)

(continued from previous page)

```
print("The sequence counts ", sequence.count('H'), " histidines")
subseq = "NLRVEYLDDRN"
print("Does the sequence contain ", subseq, "?", subseq in sequence )
pos = sequence.find(subseq)
getS = sequence[pos:pos+len(subseq)]
print(subseq, " is present at pos: ", pos , "[check:", getS , "]")

end_first_line = chain_a.find('\n')
print("The first line is: ", chain_a[0:end_first_line])
```

```
SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
EPHHELPPGSTKRALPNNT
The sequence is in  6  lines
The sequence has  219  aminoacids
The sequence counts  10  cysteins
The sequence counts  9  histidines
Does the sequence contain  NLRVEYLDDRN ? True
NLRVEYLDDRN  is present at pos:  106 [check: NLRVEYLDDRN ]
The first line is:  SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
```

10. Calculate the zeros of the equation $ax^2 - b = 0$ where a = 10 and b = 1. Hint: use math.sqrt or ** 0.5. Finally check that substituting the obtained value of x in the equation gives zero.

Show/Hide Solution

[32]:
```
import math

A = 10
B = 1

X = math.sqrt(B/A)

print("10X**2 - 1 = 0 for X:", X)
print(10*X**2 -1 == 0)
```

```
10X**2 - 1 = 0 for X: 0.31622776601683794
True
```

CHAPTER

FOUR

PRACTICAL 3

In this practical we will work with lists and tuples.

4.1 Slides

The slides of the introduction can be found here: Intro

4.2 Lists

Python lists are **ordered** collections of (homogeneous) objects, but they can hold also non-homogeneous data. List are **mutable objects**. Elements of the collection are specified within two square brackets **[]** and are comma separated.

We can use the function print to print the content of lists. Some examples of list definitions follow:

[1]:
```
my_first_list = [1,2,3]
print("first:" , my_first_list)

my_second_list = [1,2,3,1,3] #elements can appear several times
print("second: ", my_second_list)

fruits = ["apple", "pear", "peach", "strawberry", "cherry"] #elements can be strings
print("fruits:", fruits)

an_empty_list = []
print("empty:" , an_empty_list)

another_empty_list = list()
print("another empty:", another_empty_list)

a_list_containing_other_lists = [[1,2], [3,4,5,6]] #elements can be other lists
print("list of lists:", a_list_containing_other_lists)

my_final_example = [my_first_list, a_list_containing_other_lists]
print("a list of lists of lists:", my_final_example)
```

```
first: [1, 2, 3]
second:  [1, 2, 3, 1, 3]
fruits: ['apple', 'pear', 'peach', 'strawberry', 'cherry']
empty: []
```

(continues on next page)

(continued from previous page)

```
another empty: []
list of lists: [[1, 2], [3, 4, 5, 6]]
a list of lists of lists: [[1, 2, 3], [[1, 2], [3, 4, 5, 6]]]
```

4.2.1 Operators for lists

Python provides several operators to handle lists. The following operators behave like on strings (**remember that, as in strings, the first position is 0!**):

Result	Operator	Meaning
bool	=, !=	Check if two lists are equal or different
int	len(list)	Return the length of the list
list	list + list	Concatenate two lists (returns a new list)
list	list * int	Replicate the list (returns a new list)
list	list[int:int]	Extract a sub-list

While this **in** operator requires that the whole tested obj is present in the list

Result	Operator	Meaning
bool	obj in list	Check if an element is present in a list

and

Result	Operator	Meaning
obj	list[int]	Read/write an element at a specified index

can also change the corresponding value of the list (**lists are mutable objects**).

Let's see some examples.

[2]:
```
A = [1, 2, 3 ]
B = [1, 2, 3, 1, 2]

print("A is a ", type(A))

print(A, " has length: ", len(A))
print("A[0]: ", A[0], " A[1]:", A[1], " A[-1]:", A[-1])

print(B, " has length: ", len(B))
print("Is A equal to B?", A == B)

C = A + [1, 2]
print(C)
print("Is C equal to B?", B == C)   #same content
print("Is C the same object as B?", B is C) #different objects
D = [1, 2, 3]*8
```

(continues on next page)

(continued from previous page)

```
print(D)

E = D[12:18] #slicing
print(E)
print("Is A*2 equal to E?", A*2 == E)
```

```
A is a  <class 'list'>
[1, 2, 3]  has length:  3
A[0]:  1  A[1]: 2  A[-1]: 3
[1, 2, 3, 1, 2]  has length:  5
Is A equal to B? False
[1, 2, 3, 1, 2]
Is C equal to B? True
Is C the same object as B? False
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 1, 2, 3]
Is A*2 equal to E? True
```

[3]:
```
A = [1, 2, 3, 4, 5, 6]
B = [1, 3, 5]
print("A:", A)
print("B:", B)

print("Is B in A?", B in A)
print("A\'s ID:", id(A))
A[5] = [1,3,5] #we can add elements
print(A)
print("A\'s ID:", id(A)) #same as before! why?
print("A has length:", len(A))
print("Is now B in A?", B in A)
```

```
A: [1, 2, 3, 4, 5, 6]
B: [1, 3, 5]
Is B in A? False
A's ID: 139779855942920
[1, 2, 3, 4, 5, [1, 3, 5]]
A's ID: 139779855942920
A has length: 6
Is now B in A? True
```

Note: When **indexing**, do not exceed the list boundaries (or you will be prompted a `list index out of range` error).

Consider the following example:

[4]:
```
A = [1, 2, 3, 4, 5, 6]
print("A has length:", len(A))

print("First element:", A[0])
print("7th-element: ", A[6])
```

```
A has length: 6
First element: 1
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-699e5f04cae0> in <module>
      3
      4 print("First element:", A[0])
----> 5 print("7th-element: ", A[6])

IndexError: list index out of range
```

It is actually fine to exceed boundaries with slicing instead:

[5]:
```
A = [1, 2, 3, 4, 5, 6]
print("A has length:", len(A))

print("First element:", A[0])
print("last element: ", A[-1])

print("3rd to 10th: ", A[2:10])

print("8th to 11th:", A[7:11])
```

```
A has length: 6
First element: 1
last element:  6
3rd to 10th:  [3, 4, 5, 6]
8th to 11th: []
```

Example: Consider the matrix $M = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 1 \\ 1 & 1 & 3 \end{bmatrix}$ and the vector $v = [10, 5, 10]^T$. What is the matrix-vector product $M * v$?

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 1 \\ 1 & 1 & 3 \end{bmatrix} * [10, 5, 10]^T = [50, 30, 45]^T$$

[6]:
```
M = [[1, 2, 3], [1, 2, 1], [1, 1, 3]]
v = [10, 5, 10]
prod = [0, 0 ,0] #at the beginning the product is the null vector

prod[0]=M[0][0]*v[0] + M[0][1]*v[1] + M[0][2]*v[2]
prod[1]=M[1][0]*v[0] + M[1][1]*v[1] + M[1][2]*v[2]
prod[2]=M[2][0]*v[0] + M[2][1]*v[1] + M[2][2]*v[2]

print("M: ", M)
print("v: ", v)
print("M*v: ", prod)
```

```
M:  [[1, 2, 3], [1, 2, 1], [1, 1, 3]]
v:  [10, 5, 10]
M*v:  [50, 30, 45]
```

4.2.2 Methods of the class list

The class list has some methods that can be used to operate on it. Recall from the lecture the following methods:

Return	Method	Meaning
None	`list.append(obj)`	Add a new element at the end of the list
None	`list.extend(list)`	Add several new elements at the end of the list
None	`list.insert(int,obj)`	Add a new element at some given position
None	`list.remove(obj)`	Remove the first occurrence of an element
None	`list.reverse()`	Invert the order of the elements
None	`list.sort()`	Sort the elements
int	`list.count(obj)`	Count the occurrences of an element

Note: Lists are **mutable objects** and therefore virtually all the previous methods (except **count**) do not have an output value, but they **modify** the list.

Some usage examples follow:

[7]:
```
#A numeric list
A = [1, 2, 3]
print(A)
print("A has id:", id(A))
A.append(72) #appends one and only one object
print(A)
print("A has id:", id(A))
A.extend([1, 5, 124, 99]) #adds all these objects, one after the other.
print(A)
A.reverse() #NOTE: NO RETURN VALUE!!!
print(A)
A.sort()
print(A)
print("Min value: ", A[0]) # In this simple case, could have used min(A)
print("Max value: ", A[-1]) #In this simple case, could have used max(A)
print("Number 1 appears:", A.count(1), " times")
print("While number 837: ", A.count(837))

print("\nDone with numbers, let's go strings...\n")
#A string list
fruits = ["apple", "banana", "pineapple", "cherry","pear", "almond", "orange"]
#Let's get a reverse lexicographic order:
print(fruits)
fruits.sort()
```

(continues on next page)

(continued from previous page)

```
fruits.reverse() # equivalent to: fruits.sort(reverse=True)
print(fruits)
fruits.remove("banana")
print(fruits)
fruits.insert(5, "wild apple") #put wild apple after apple.
print(fruits)
print("\nSorted fruits:")
fruits.sort() # does not return anything. Modifies list!
print(fruits)
```

```
[1, 2, 3]
A has id: 139779846805128
[1, 2, 3, 72]
A has id: 139779846805128
[1, 2, 3, 72, 1, 5, 124, 99]
[99, 124, 5, 1, 72, 3, 2, 1]
[1, 1, 2, 3, 5, 72, 99, 124]
Min value:  1
Max value:  124
Number 1 appears: 2  times
While number 837:  0

Done with numbers, let's go strings…

['apple', 'banana', 'pineapple', 'cherry', 'pear', 'almond', 'orange']
['pineapple', 'pear', 'orange', 'cherry', 'banana', 'apple', 'almond']
['pineapple', 'pear', 'orange', 'cherry', 'apple', 'almond']
['pineapple', 'pear', 'orange', 'cherry', 'apple', 'wild apple', 'almond']

Sorted fruits:
['almond', 'apple', 'cherry', 'orange', 'pear', 'pineapple', 'wild apple']
```

An important thing to remember that we mentioned already a couple of times is that lists **are mutable objects** and therefore **virtually all the previous methods (except count) do not have an output value**:

[8]:
```
A = ["A", "B", "C"]

print("A:", A)

A_new = A.append("D")

print("A:", A)

print("A_new:", A_new)

#A_new is None. We cannot apply methods to it...
print(A_new is None)
print("A_new has " , A_new.count("D"), " Ds")
```

```
A: ['A', 'B', 'C']
A: ['A', 'B', 'C', 'D']
A_new: None
True
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-114913bce16b> in <module>
     11 #A_new is None. We cannot apply methods to it…
```

(continues on next page)

(continued from previous page)

```
     12 print(A_new is None)
---> 13 print("A_new has " , A_new.count("D"), " Ds")

AttributeError: 'NoneType' object has no attribute 'count'
```

Some things to remember

1. append and extend work quite differently:

[ ]:
```
A = [1, 2, 3]

A.extend([4, 5])
print(A)
B = [1, 2, 3]
B.append([4,5])
print(B)
```

2. To remove an object it must exist:

[ ]:
```
A = [1,2,3, [[4],[5,6]], 8]
print(A)
A.remove(2)
print(A)
A.remove([[4],[5,6]])
print(A)
A.remove(7)
```

3. To sort a list, its elements must be sortable (i.e. homogeneous)!

[9]:
```
A = [4,3, 1,7, 2]
print("A:", A)
A.sort()
print("A sorted:",  A)
A.append("banana")
print("A:", A)
A.sort()
print("A:", A)
```

```
A: [4, 3, 1, 7, 2]
A sorted: [1, 2, 3, 4, 7]
A: [1, 2, 3, 4, 7, 'banana']
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-b37960bcb6f2> in <module>
      5 A.append("banana")
      6 print("A:", A)
----> 7 A.sort()
      8 print("A:", A)

TypeError: '<' not supported between instances of 'str' and 'int'
```

Important to remember:

Lists are **mutable objects** and this has some consequences! Since lists are mutable objects, they hold references to objects rather than objects.

Take a look at the following examples:

[10]:
```
l1 = [1, 2]
l2 = [4, 3]
LL = [l1, l2]
print("l1:", l1)
print("l2:",l2)
print("LL:", LL)
l1.append(7)
print("\nAppending 7 to l1...")
print("l1:", l1)
print("LL now: ", LL)

LL[0][1] = -1
print("\nSetting LL[0][1]=-1...")
print("LL now:" , LL)
print("l1 now", l1)
#but the list can point also to a different object,
#without affecting the original list.
LL[0] = 100
print("\nSetting LL[0] = 100")
print("LL now:", LL)
print("l1 now", l1)
```

```
l1: [1, 2]
l2: [4, 3]
LL: [[1, 2], [4, 3]]

Appending 7 to l1…
l1: [1, 2, 7]
LL now:  [[1, 2, 7], [4, 3]]

Setting LL[0][1]=-1…
LL now: [[1, -1, 7], [4, 3]]
l1 now [1, -1, 7]

Setting LL[0] = 100
LL now: [100, [4, 3]]
l1 now [1, -1, 7]
```

Making copies

There are several ways to copy a list into another. Let's see the difference between = and [:]. Note what happens when lists get complicated.

[11]:
```
A = ["hi", "there"]
B = A
print("A:", A)
print("B:", B)
A.extend(["from", "python"])
print("A now: ", A)
print("B now: ", B)

print("\n---- copy example -------")
#Let's make a distinct copy of A.
C = A[:] #all the elements of A have been copied in C
print("C:", C)
A[3] = "java"
```

(continues on next page)

(continued from previous page)

```
print("A now:", A)
print("C now:", C)

print("\n---- be careful though -------")
#Watch out though that this is a shallow copy...
D = [A, A]
E = D[:]
print("D:", D)
print("E:", E)

D[0][0] = "hello"
print("\nD now:", D)
print("E now:", E)
print("A now:", A)
```

```
A: ['hi', 'there']
B: ['hi', 'there']
A now:  ['hi', 'there', 'from', 'python']
B now:  ['hi', 'there', 'from', 'python']

---- copy example -------
C: ['hi', 'there', 'from', 'python']
A now: ['hi', 'there', 'from', 'java']
C now: ['hi', 'there', 'from', 'python']

---- be careful though -------
D: [['hi', 'there', 'from', 'java'], ['hi', 'there', 'from', 'java']]
E: [['hi', 'there', 'from', 'java'], ['hi', 'there', 'from', 'java']]

D now: [['hello', 'there', 'from', 'java'], ['hello', 'there', 'from', 'java']]
E now: [['hello', 'there', 'from', 'java'], ['hello', 'there', 'from', 'java']]
A now: ['hello', 'there', 'from', 'java']
```

Equality and identity

Two different operators exist to check the **equality** of two lists (==) and the **identity** of two lists (is).

[12]:
```
A = [1, 2, 3]
B = A
C = [1, 2, 3]
print("Is A equal to B?", A == B)
print("Is A actually B?", A is B)
print("Is A equal to C?", A == C)
print("Is A actually C?", A is C)
#in fact:
print("\nA's id:", id(A))
print("B's id:", id(B))
print("C's id:", id(C))
#just to confirm that:
A.append(4)
B.append(5)
print("\nA now: ", A)
print("B now: ", B)
print("C now:", C)
```

```
Is A equal to B? True
Is A actually B? True
Is A equal to C? True
```

(continues on next page)

(continued from previous page)

```
Is A actually C? False

A's id: 139779847198600
B's id: 139779847198600
C's id: 139779847159368

A now:  [1, 2, 3, 4, 5]
B now:  [1, 2, 3, 4, 5]
C now: [1, 2, 3]
```

4.2.3 From strings to lists, the `split` method

Strings have a method *split* that can literally split the string at specific characters.

Example

Recall the protein seen in the previous practical:

```
chain_a = """SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
EPHHELPPGSTKRALPNNT"""
```

how can we split it into several lines?

[13]:
```
chain_a = """SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
EPHHELPPGSTKRALPNNT"""

lines = chain_a.split('\n')
print("Original sequence:")
print( chain_a, "\n") #some spacing to keep things clear
print("line by line:")
# write the following and you will appreciate loops! :-)
print("1st line:" ,lines[0])
print("2nd line:" ,lines[1])
print("3rd line:" ,lines[2])
print("4th line:" ,lines[3])
print("5th line:" ,lines[4])
print("6th line:" ,lines[5])

print("\nSplit the 1st line in correspondence of FRL:\n",lines[0].split("FRL"))
```

```
Original sequence:
SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
EPHHELPPGSTKRALPNNT
```

(continues on next page)

(continued from previous page)

```
line by line:
1st line: SSSVPSQKTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKM
2nd line: FCQLAKTCPVQLWVDSTPPPGTRVRAMAIYKQSQHMTEVV
3rd line: RRCPHHERCSDSDGLAPPQHLIRVEGNLRVEYLDDRNTFR
4th line: HSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRPILT
5th line: IITLEDSSGNLLGRNSFEVRVCACPGRDRRTEEENLRKKG
6th line: EPHHELPPGSTKRALPNNT

Split the 1st line in correspondence of FRL:
 ['SSSVPSQKTYQGSYG', 'GFLHSGTAKSVTCTYSPALNKM']
```

Note that in the last instruction, the substring FRL is disappeared (as happened to the newline).

4.2.4 And back to strings with the `join` method

Given a list, one can join the elements of the list together into a string by using the `join` method of the class string. The syntax is the following: *str.join(list)* which joins together all the elements in the list in a string separating them with the string *str*.

Example Given the list ['Sept', '30th', '2020', '11:30'], let's combine all its elements in a string joining the elements with a dash ("-") and print them. Let's finally join them with a tab ("\t") and print them.

[14]:
```
vals = ['Sept', '30th', '2020', '11:30']
print(vals)
myStr = "-".join(vals)
print("\n" + myStr)
myStr = "\t".join(vals)
print("\n" + myStr)
```

```
['Sept', '30th', '2020', '11:30']

Sept-30th-2020-11:30

Sept    30th    2020    11:30
```

4.3 Tuples

Tuples are the **immutable** version of lists (i.e. it is not possible to change their content without actually changing the object). They are sequential collections of objects, and elements of tuples are assumed to be in a particular order. They can hold heterogeneous information. They are defined with the brackets (). Some examples:

[15]:
```
first_tuple = (1,2,3)
print(first_tuple)

second_tuple = (1,) #this contains one element only, but we need the comma!
var = (1) #This is not a tuple!!!
print(second_tuple, " type:", type(second_tuple))
print(var, " type:", type(var))
empty_tuple = () #fairly useless
print(empty_tuple, "\n")
third_tuple = ("January", 1 ,2007) #heterogeneous info
print(third_tuple)
```

(continues on next page)

(continued from previous page)

```
days = (third_tuple,("February",2,1998), ("March",2,1978),("June",12,1978))
print(days, "\n")

#Remember tuples are immutable objects...
print("Days has id: ", id(days))
days = ("Mon","Tue","Wed","Thu","Fri","Sat","Sun")
#...hence reassignment creates a new object
print("Days now has id: ", id(days))
```

```
(1, 2, 3)
(1,)  type: <class 'tuple'>
1  type: <class 'int'>
()

('January', 1, 2007)
(('January', 1, 2007), ('February', 2, 1998), ('March', 2, 1978), ('June', 12, 1978))

Days has id:  139779908702520
Days now has id:  139779855774080
```

The following operators work on tuples and they behave exactly as on lists:

Result	Operator	Meaning
`bool`	`=, !=`	Check if two tuples are equal or different
`int`	`len(tuple)`	Return the length of the tuple
`tuple`	`tuple + tuple`	Concatenate two tuples (returns a new tuple)
`tuple`	`tuple * int`	Replicate the tuple (returns a tuple)
`tuple`	`tuple[int]`	Read an element of the tuple
`tuple`	`tuple[int:int]`	Extract a sub-tuple

[16]:
```
practical1 = ("Wednesday", "23/09/2020")
practical2 = ("Monday", "28/09/2020")
practical3 = ("Wednesday", "30/09/2020")

#A tuple containing 3 tuples
lectures = (practical1, practical2, practical3)
#One tuple only
mergedLectures = practical1 + practical2 + practical3

print("The first three lectures:\n", lectures, "\n")
print("mergedLectures:\n", mergedLectures)

#This returns the whole tuple
print("1st lecture was on: ", lectures[0], "\n")
#2 elements from the same tuple
print("1st lecture was on ", mergedLectures[0], ", ", mergedLectures[1], "\n")
# Return type is tuple!
```

(continues on next page)

(continued from previous page)

```
print("3rd lecture was on: ", lectures[2])
#2 elements from the same tuple returned in tuple
print("3rd lecture was on ", mergedLectures[4:], "\n")
```

```
The first three lectures:
 (('Wednesday', '23/09/2020'), ('Monday', '28/09/2020'), ('Wednesday', '30/09/2020'))

mergedLectures:
 ('Wednesday', '23/09/2020', 'Monday', '28/09/2020', 'Wednesday', '30/09/2020')
1st lecture was on:  ('Wednesday', '23/09/2020')

1st lecture was on  Wednesday ,  23/09/2020

3rd lecture was on:  ('Wednesday', '30/09/2020')
3rd lecture was on  ('Wednesday', '30/09/2020')
```

The following methods are available for tuples:

Return	Method	Meaning
int	`tuple.count(obj)`	Count the occurrences of an element
int	`tuple.index(obj)`	Return the index of the first occurrence of an object

[17]:
```
practical1 = ("Wednesday", "23/09/2020")
practical2 = ("Monday", "28/09/2020")
practical3 = ("Wednesday", "30/09/2020")


mergedLectures = practical1 + practical2 + practical3  #One tuple only
print(mergedLectures.count("Wednesday"), " lectures were on Wednesday")
print(mergedLectures.count("Monday"), " lecture was on Monday")
print(mergedLectures.count("Friday"), " lectures was on Friday")

print("Index:", practical2.index("Monday"))
#You cannot look for an element that does not exist
print("Index:", practical2.index("Wednesday"))
```

```
2  lectures were on Wednesday
1  lecture was on Monday
0  lectures was on Friday
Index: 0
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-f06c6edd5ecf> in <module>
     11 print("Index:", practical2.index("Monday"))
     12 #You cannot look for an element that does not exist
---> 13 print("Index:", practical2.index("Wednesday"))
```

(continues on next page)

(continued from previous page)

```
    14

ValueError: tuple.index(x): x not in tuple
```

4.4 Exercises

1. Given the following text string:

```
"""this is a text
string on
several lines that does not say anything."""
```

   a) print it; b) print how many lines, words and characters it contains. Finally, c)sort the words alphabetically and print the first and the last in lexicographic order.

Show/Hide Solution

[18]:
```
T = """this is a text
string on
several lines that does not say anything."""

#a) print it
print(T)
print("")
#b) print the lines, words and characters
lines = T.split('\n')

#NOTE: words are split by a space or a newline! They have already been
#split by newline.

words = lines[0].split(' ') + lines[1].split(' ') + lines[2].split(' ')

num_chars = len(T) #this includes spaces and newlines

print("Lines:", len(lines), "words:", len(words), "chars:", num_chars)
# alternative way for number of characters:
print("")
characters = list(T) #put all the characters of the string in a list
num_chars2 = len(characters)
print(characters)
print(num_chars2)

print("Unsorted words: ", words)
words.sort() #NOTE: it does not return ANYTHING!!!
print("\nSorted words: ", words)
print("")
print("First word: ", words[0])
print("Last word:", words[-1])
```

```
this is a text
string on
several lines that does not say anything.

Lines: 3 words: 13 chars: 66
```

(continues on next page)

(continued from previous page)

```
['t', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 't', 'e', 'x', 't', '\n', 's', 't',
↪ 'r', 'i', 'n', 'g', ' ', 'o', 'n', '\n', 's', 'e', 'v', 'e', 'r', 'a', 'l', ' ', 'l
↪', 'i', 'n', 'e', 's', ' ', 't', 'h', 'a', 't', ' ', 'd', 'o', 'e', 's', ' ', 'n',
↪'o', 't', ' ', 's', 'a', 'y', ' ', 'a', 'n', 'y', 't', 'h', 'i', 'n', 'g', '.']
66
Unsorted words:  ['this', 'is', 'a', 'text', 'string', 'on', 'several', 'lines', 'that
↪', 'does', 'not', 'say', 'anything.']

Sorted words:  ['this', 'that', 'text', 'string', 'several', 'say', 'on', 'not',
↪'lines', 'is', 'does', 'anything.', 'a']

First word:  a
Last word: this
```

2. The variant calling format (VCF[36]) is a format to represent structural variants of genomes (i.e. SNPs, insertions, deletions) . Each line of this format represents a variant, every piece of information within a line is separated by a tab (\t in python). The first 5 fields of this format report the chromosome (chr), the position (pos), the name of the variant (name), the reference allele (REF) and the alternative allele (ALT). Assuming to have a variable VCF defined containing the following three lines (representing three SNPs):

```
VCF = """MDC000001.124\\t7112\\tFB_AFFY_0000024\\tG\\tA
MDC000002.328\\t941\\tFB_AFFY_0000144\\tC\\tT
MDC000004.272\\t2015\\tFB_AFFY_0000222\\tG\\tA"""
```

```
1. Store these three variants as a list of lists, where each one of the fields is␣
↪kept separate
(e.g. the list should be similar to:
[[chr1,pos1,name1,ref1,alt1], [chr2, pos2, name2, ref2, alt2], ...] where all the␣
↪elements
are as specified in the string VCF (note that "..." means that the list is not␣
↪complete).

2. Print each variant changing its format in:  "name|chr|pos|REF/ALT".
```

Show/Hide Solution

[27]:
```
VCF = """MDC000001.124\t7112\tFB_AFFY_0000024\tG\tA
MDC000002.328\t941\tFB_AFFY_0000144\tC\tT
MDC000004.272\t2015\tFB_AFFY_0000222\tG\tA"""

variants = VCF.split('\n')
print(variants)

variants[0] = variants[0].split('\t')
variants[1] = variants[1].split('\t')
variants[2] = variants[2].split('\t')

print(variants)

print(variants[0])
print(variants[1])
print(variants[2], "\n")
```

(continues on next page)

[36] http://www.internationalgenome.org/wiki/Analysis/vcf4.0/

(continued from previous page)

```
info = variants[0]
print(info[2] + "|" + info[0] + "|" + info[1] + "|" + info[3] +"/" + info[4])
info = variants[1]
print(info[2] + "|" + info[0] + "|" + info[1] + "|" + info[3] +"/" + info[4])
info = variants[2]
print(info[2] + "|" + info[0] + "|" + info[1] + "|" + info[3] +"/" + info[4])
```

```
['MDC000001.124\t7112\tFB_AFFY_0000024\tG\tA', 'MDC000002.328\t941\tFB_AFFY_0000144\
↪tC\tT', 'MDC000004.272\t2015\tFB_AFFY_0000222\tG\tA']
[['MDC000001.124', '7112', 'FB_AFFY_0000024', 'G', 'A'], ['MDC000002.328', '941', 'FB_
↪AFFY_0000144', 'C', 'T'], ['MDC000004.272', '2015', 'FB_AFFY_0000222', 'G', 'A']]
['MDC000001.124', '7112', 'FB_AFFY_0000024', 'G', 'A']
['MDC000002.328', '941', 'FB_AFFY_0000144', 'C', 'T']
['MDC000004.272', '2015', 'FB_AFFY_0000222', 'G', 'A']

FB_AFFY_0000024|MDC000001.124|7112|G/A
FB_AFFY_0000144|MDC000002.328|941|C/T
FB_AFFY_0000222|MDC000004.272|2015|G/A
```

3. Given the list L = ["walnut", "eggplant", "lemon", "lime", "date", "onion", "nectarine", "endive"]:

```
1. Create another list (called newList) containing the first letter of each␣
↪element of L
(e.g newList =["w", "e", ...]).
2. Add a space to newList at position 4 and append an exclamation mark (!) at the␣
↪end.
3. Print the list.
4. Print the content of the list joining all the elements with an empty space
(i.e. use the method join: "".join(newList) )
```

Show/Hide Solution

[29]:
```
L = ["walnut", "eggplant", "lemon", "lime", "date", "onion", "nectarine", "endive" ]

newList = []
#the next few lines are to make you appreciate loops! :-)
newList.append(L[0][0])
newList.append(L[1][0])
newList.append(L[2][0])
newList.append(L[3][0])
newList.append(L[4][0])
newList.append(L[5][0])
newList.append(L[6][0])
newList.append(L[7][0])

newList.insert(4," ")
newList.append("!")

print(newList)
print("\n", "".join(newList))
```

```
['w', 'e', 'l', 'l', ' ', 'd', 'o', 'n', 'e', '!']

 well done!
```

4. Fastq is a standard format for storing sequences and quality information. More information on the format can be

found here[37]. This format can be used to store sequencing information coming from the sequencer. Each entry represents a read (i.e. a sequence) and carries four different pieces of information. A sample entry is the following:

```
entry = """@HWI-ST1296:75:C3F7CACXX:1:1101:19142:14904
CCAACAACTTTGACGCTAAGGATAGCTCCATGGCAGCATATCTGGCACAA
+
FHIIJIJJGIJJJJJIHHHFFFFFEE:;CIDDDDDDDDDDDDEDDDDDDB"""
```

where:

- (i) the first line is the read identifier starts with a "@" and carries several types of information regarding the instrument used for sequencing (the reported example is an illumina read - if you are interested, you can find more info on the format of the ID here[38].
- (ii). the second information is the sequence of the read (note that it can span several lines, not in our simple example though);
- (iii) a "+" sign to mark the end of the sequence information (optionally the read identifier can be repeated);
- (iv) the phred quality score of the read encoded as a text string. It must contain the same number of elements that are in the sequence. To decode each character into a the corresponding phred score, one needs to convert it into the unicode integer representation of it - 33. For example, the conversion of a character "I" in python can be done using the *ord* built in function in the following way: ord("I") - 33 = 40. Finally the phred score can be converted into probability of the base to be wrong with the following formula: $P = 10^{-Q/10}$, where Q is the phred quality score.

Given the entry above:

```
1. Check that the ID starts with a @
2. Store the sequence as a list where each element is one single base
(e.g. sequence =['T', 'A',...])
3. Store the quality as a list where each element is one single quality character
(e.g. qualChar = ['C', 'C', ...])
4. Check that the length of the sequence and quality are the same
5. Count how many times the sequence "TCCA" appears in the read
6. Retrieve the sub-list containing the quality values corresponding to the "TCCA"␣
↪string
7. Convert each value in the list at point 6 in the corresponding probability of the␣
↪base
being wrong
```

Show/Hide Solution

[31]:
```
entry = """@HWI-ST1296:75:C3F7CACXX:1:1101:19142:14904
CCAACAACTTTGACGCTAAGGATAGCTCCATGGCAGCATATCTGGCACAA
+
FHIIJIJJGIJJJJJIHHHFFFFFEE:;CIDDDDDDDDDDDDEDDDDDDB
"""

#1. Check that the ID starts with a @
print("Does header start with @? ", entry.startswith("@"))
entryList = entry.split('\n')

#2. and 3. Store the sequence/quality as a list where each element is one single base
sequence = list(entryList[1])
qualChar = list(entryList[3])
```

(continues on next page)

[37] https://en.wikipedia.org/wiki/FASTQ_format
[38] https://support.illumina.com/help/BaseSpace_OLH_009008/Content/Source/Informatics/BS/NamingConvention_FASTQ-files-swBS.htm

(continued from previous page)

```
#4. Check that the length of the sequence and quality are the same
print("Sequence and quality have the same length:", len(sequence) == len(qualChar))

#5. Count how many times the sequence "TCCA" appears in the read
print("Sequence \"TCAA\" is present ", entryList[1].count("TCCA"), " times" )

#6. Retrieve the sub-list containing the quality values corresponding to the "TCCA"␣
↪string
pos = entryList[1].find("TCCA")

qVals = qualChar[pos:pos+4]

print("Sequence: ", entryList[1][pos:pos+4])
print("Quality: ", qVals)

#7. Convert each value in the list at point 6 in the corresponding probability of the␣
↪base
# being wrong

phredS = []

phredS.append(ord(qVals[0]) - 33)
phredS.append(ord(qVals[1]) - 33)
phredS.append(ord(qVals[2]) - 33)
phredS.append(ord(qVals[3]) - 33)

print("Phred score:", phredS)

probVals = []
probVals.append(10**(0-phredS[0]/10))
probVals.append(10**(0-phredS[1]/10))
probVals.append(10**(0-phredS[2]/10))
probVals.append(10**(0-phredS[3]/10))

print("Error Probabilities:\n", probVals)
```

```
Does header start with @?  True
Sequence and quality have the same length: True
Sequence "TCAA" is present  1  times
26
['T', 'C', 'C', 'A']
Sequence:  TCCA
Quality:  [':', ';', 'C', 'I']
Phred score: [25, 26, 34, 40]
Error Probabilities:
 [0.0031622776601683794, 0.0025118864315095794, 0.00039810717055349735, 0.0001]
```

5. Given the list L = [10, 60, 72, 118, 11, 71, 56, 89, 120, 175] find the min, max and median value (hint: sort it and extract the right values). Create a list with only the elements at even indexes (i.e. [10, 72, 11, ..], note that the ".." means that the list is not complete) and re-compute min, max and median values. Finally, re-do the same for the elements located at odd indexes (i.e. [60, 118,..]).

Show/Hide Solution

[22]:
```
L = [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]
Lodd = L[1::2] #get only odd-indexed elements
Leven = L[0::2] #get only even-indexed elements

print("original:" , L)
print("Lodd:", Lodd)
print("Leven:", Leven)
L.sort()
Lodd.sort()
Leven.sort()


print("sorted:  " , L)
print("sorted Lodd:  " , Lodd)
print("sorted Leven:  " , Leven)

print("L: Min: ", L[0], " Max." , L[-1], " Median: ", L[len(L) // 2])
print("Lodd: Min: ", Lodd[0], " Max." , Lodd[-1], " Median: ", Lodd[len(Lodd) // 2])
print("Leven: Min: ", Leven[0], " Max." , Leven[-1], " Median: ", Leven[len(Leven) //␣
↪2])
```

```
original: [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]
Lodd: [60, 118, 71, 89, 175]
Leven: [10, 72, 11, 56, 120]
sorted:   [10, 11, 56, 60, 71, 72, 89, 118, 120, 175]
sorted Lodd:   [60, 71, 89, 118, 175]
sorted Leven:   [10, 11, 56, 72, 120]
L: Min:  10  Max. 175  Median:  72
Lodd: Min:  60  Max. 175  Median:  89
Leven: Min:  10  Max. 120  Median:  56
```

6. Given the string pets = “siamese cat,dog,songbird,guinea pig,rabbit,hampster” convert it into a list. Create then a tuple of tuples where each tuple has two information: the name of the pet and the length of the name. E.g. ((“dog”,3), (“hampster”,8)). Print the tuple.

Show/Hide Solution

[23]:
```
pets = "cat,dog,bird,guinea pig,rabbit,hampster"

pet_list = pets.split(',')

print(pet_list)

pet_tuples = ((pet_list[0], len(pet_list[0])),
              (pet_list[1], len(pet_list[1])),
              (pet_list[2], len(pet_list[2])),
              (pet_list[3], len(pet_list[3])),
              (pet_list[4], len(pet_list[4])),
              (pet_list[5], len(pet_list[5])))

print(pet_tuples)
```

```
['cat', 'dog', 'bird', 'guinea pig', 'rabbit', 'hampster']
(('cat', 3), ('dog', 3), ('bird', 4), ('guinea pig', 10), ('rabbit', 6), ('hampster',␣
↪8))
```

7. Given the string S=”apple|pear|apple|cherry|pear|apple|pear|pear|cherry|pear|strawberry”. Store the elements separated by the “|” in a list.

    1. How many elements does the list have?

2. Knowing that the list created at the previous point has only four distinct elements (i.e. “apple”,”pear”,”cherry” and “strawberry”), create another list where each element is a tuple containing the name of the fruit and its multiplicity (that is how many times it appears in the original list). Ex. list_of_tuples = [(“apple”, 3), (“pear”, “5”),…]
3. Print the content of each tuple in a separate line (ex. first line: apple is present 3 times)

Show/Hide Solution

[24]:
```
S="apple|pear|apple|cherry|pear|apple|pear|pear|cherry|pear|strawberry"

Slist = S.split("|")
print(Slist)

appleT = ("apple", Slist.count("apple"))
pearT = ("pear", Slist.count("pear"))
cherryT = ("cherry", Slist.count("cherry"))
strawberryT = ("strawberry", Slist.count("strawberry"))
list_of_tuples =[appleT, pearT, cherryT, strawberryT]

print(list_of_tuples, "\n") #adding newline to separate elements

print(appleT[0], " is present ", appleT[1], " times")
print(pearT[0], " is present ", pearT[1], " times")
print(cherryT[0], " is present ", cherryT[1], " times")
print(strawberryT[0], " is present ", strawberryT[1], " times")
```

```
['apple', 'pear', 'apple', 'cherry', 'pear', 'apple', 'pear', 'pear', 'cherry', 'pear
↪', 'strawberry']
[('apple', 3), ('pear', 5), ('cherry', 2), ('strawberry', 1)]

apple  is present  3  times
pear  is present  5  times
cherry  is present  2  times
strawberry  is present  1  times
```

8. Define three tuples representing points in the 3D space: A = (10,20,30), B = (1,72, 100) and C = (4, 9, 20).
   1. Compute the Euclidean distance between A and B (let’s call it AB), A and C (AC), B and C (BC) and print them. Remember that the distance d between two points $X_1 = (x_1, y_1, z_1)$ and $X_2 = (x_2, y_2, z_2)$ is $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$. Hint: remember to import math to use the *sqrt* method.
   2. Create a point D multiplying every element of C by 10. (Hint: do not use C*10, as this will repeat C 10 times). And compute the distance AD, BD, CD.
   3. Answer to the following questions (writing a suitable boolean expression and printing the result):
      1. Is A closer to B than to C?
      2. Is A closer to C than to B or to D?
      3. Is B closer to A than to C to D?

Show/Hide Solution

[25]:
```
import math

A = (10,20,30)
B = (1,72, 100)
C = (4,9,20)
```

(continues on next page)

(continued from previous page)

```
D = (C[0]*10, C[1]*10, C[2]*10)

AB = math.sqrt( (B[0]-A[0])**2 + (B[1] - A[1])**2 + (B[2] - A[2])**2)
AC = math.sqrt( (C[0]-A[0])**2 + (C[1] - A[1])**2 + (C[2] - A[2])**2)
BC = math.sqrt( (C[0]-B[0])**2 + (C[1] - B[1])**2 + (C[2] - B[2])**2)
AD = math.sqrt( (D[0]-A[0])**2 + (D[1] - A[1])**2 + (D[2] - A[2])**2)
BD = math.sqrt( (D[0]-B[0])**2 + (D[1] - B[1])**2 + (D[2] - B[2])**2)
CD = math.sqrt( (D[0]-C[0])**2 + (D[1] - C[1])**2 + (D[2] - C[2])**2)

print("Distance AB: ", AB)
print("Distance AC: ", AC)
print("Distance AD: ", AD)
print("Distance BC: ", BC)
print("Distance BD: ", BD)
print("Distance CD: ", CD)

print("A is closer to B than to C: ", AB < AC)
print("A is closer to C than to B or D:", AC < AB and AC < AD)
print("B is closer to A than C to D:", AB < CD)
```

```
Distance AB:  87.66413177577246
Distance AC:  16.0312195418814
Distance AD:  186.27936010197158
Distance BC:  101.87246929372037
Distance BD:  108.83473710171766
Distance CD:  200.64147128647159
A is closer to B than to C:  False
A is closer to C than to B or D: True
B is closer to A than C to D: True
```

9. Given the matrix $M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ compute $M^2 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{bmatrix}$

Show/Hide Solution

[26]:
```
M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Msq = [[0, 0, 0], [0, 0,0], [0, 0, 0]]# the output is initialized to the zero matrix
#Msq[i,j] is represented as Msq[i][j]
Msq[0][0] = M[0][0]*M[0][0] + M[0][1]*M[1][0] + M[0][2] * M[2][0]
Msq[0][1] = M[0][0]*M[0][1] + M[0][1]*M[1][1] + M[0][2] * M[2][1]
Msq[0][2] = M[0][0]*M[0][2] + M[0][1]*M[1][2] + M[0][2] * M[2][2]
Msq[1][0] = M[1][0]*M[0][0] + M[1][1]*M[1][0] + M[1][2] * M[2][0]
Msq[1][1] = M[1][0]*M[0][1] + M[1][1]*M[1][1] + M[1][2] * M[2][1]
Msq[1][2] = M[1][0]*M[0][2] + M[1][1]*M[1][2] + M[1][2] * M[2][2]
Msq[2][0] = M[2][0]*M[0][0] + M[2][1]*M[1][0] + M[2][2] * M[2][0]
Msq[2][1] = M[2][0]*M[0][1] + M[2][1]*M[1][1] + M[2][2] * M[2][1]
Msq[2][2] = M[2][0]*M[0][2] + M[2][1]*M[1][2] + M[2][2] * M[2][2]

print(Msq)
```

```
[[30, 36, 42], [66, 81, 96], [102, 126, 150]]
```

CHAPTER

FIVE

PRACTICAL 4

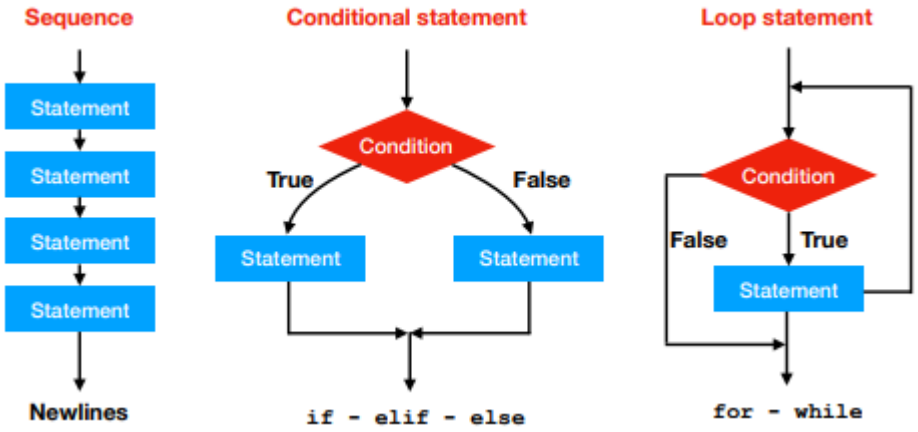
In this practical we will work with conditionals (branching) and loops.

5.1 Slides

The slides of the introduction can be found here: Intro

5.2 Execution flow

Recall from the lecture that there are at least three types of execution flows. Our statements can be simple and structured **sequentially**, when one instruction is executed right after the previous one, but some more complex flows involve **conditional** branching (when the portion of the code to be executed depends on the value of some condition), or **loops** when a portion of the code is executed multiple times until a certain condition becomes False.



These portions of code are generally called **blocks** and Python, unlike most of the programming languages, **uses indentation (and some keywords like else, ':', 'next', etc.) to define blocks**.

5.3 Conditionals

We can use conditionals any time a decision needs to be made depending on the value of some condition. A block of code will be executed if the condition is evaluated to the boolean **True** and another one if the condition is evaluated to **False**.

5.3.1 The basic *if - else* statement

The basic syntax of conditionals is an if statement like:

```
if condition :

    #This is the True branch
    #do something

else:

    #This is the False branch (or else branch)
    #do something else
```

where condition is a boolean expression that tells the interpreter which of the two blocks should be executed. **If and only if** the condition is **True** the first branch is executed, otherwise execution goes to the second branch (i.e. the else branch). Note that the **condition is followed by a ":"** character and that **the two branches are indented**. This is the way Python uses to identify the block of instructions that belong to the same branch. The **else keyword is followed by ":"** and **is not indented** (i.e. it is at the same level of the *if* statement. There is no keyword at the end of the "else branch", but **indentation tells when the block of code is finished**.

Example: Let's get an integer from the user and test if it is even or odd, printing the result to the screen.

[1]:
```
num = int(input("Dear user give me an integer:"))
res = ""
if num % 2 == 0:
    #The number is even
    res = "even"
else:
    #The number is odd
    res = "odd"

print("Number ", num, " is ", res)
```

```
Dear user give me an integer:27
Number  27  is  odd
```

Note that the execution is sequential until the *if* keyword, then it branches until the indentation goes back to the same level of the if (i.e. the two branches rejoin at the *print* statement in the final line). **Remember that the else branch is optional.**

5.3.2 The *if - elif - else* statement

If statements can be chained in such a way that there are more than two possible branches to be followed. Chaining them with the **if - elif - else** statement will make execution follow only one of the possible paths.

The syntax is the following:

```
if condition :

    #This is branch 1
    #do something

elif condition1 :

    #This is branch 2
    #do something

elif condition2 :

    #This is branch 3
    #do something

else:

    #else branch. Executed if all other conditions are false
    #do something else
```

Note that **branch 1** is executed if condition is **True**, **branch 2** if and only if **condition is False and condition1 is True**, **branch 3** if condition is **False, condition 1 is False and condition2 is True**. **If all conditions are False the else branch is executed**.

Example: The tax rate of a salary depends on the income. If the income is < 10000 euros, no tax is due, if the income is between 10000 euros and 20000 the tax rate is 25%, if between 20000 and 45000 it is 35% otherwise it is 40%. What is the tax due by a person earning 35000 euros per year?

[2]:
```
income = 35000
rate = 0.0

if income < 10000:
    rate = 0
elif income < 20000:
    rate = 0.25
elif income < 45000:
    rate = 0.35
else:
    rate = 0.4

tax = income*rate

print("The tax due is ", tax, " euros (i.e ", rate*100, "%)")
```

```
The tax due is  12250.0  euros (i.e  35.0 %)
```

Note the difference in the two following cases:

[3]:
```
#Example 1

val = 10
```

(continues on next page)

(continued from previous page)

```
if val > 5:
    print("Value >5")
elif val > 5:
    print("I said value is >5!")
else:
    print("Value is <= 5")

#Example 2

val = 10

if(val > 5):
    print("\n\nValue is >5")

if(val > 5):
    print("I said Value is >5!!!")
```

```
Value >5


Value is >5
I said Value is >5!!!
```

5.4 Loops

Looping is the ability of repeating a specific block of code several times (i.e. until a specific condition is True or there are no more elements to process).

5.4.1 For loop

The *for* loop is used to loop over a collection of objects (e.g. a string, list, tuple, …). The basic syntax of the for loop is the following:

```
for elem in collection :
    #OK, do something with elem
    # instruction 1
    # instruction 2
```

the variable *elem* will get the value of each one of the elements present in *collection* one after the other. The end of the block of code to be executed for each element in the collection is again defined by indentation.

Depending on the type of the collection elem will get different values. Recall from the lecture that:

`str`	`for` iterates over the characters
`list`	`for` iterates over the elements
`tuple`	`for` iterates over the elements
`dict`	`for` iterates over the keys

Let's see this in action:

[4]:
```
S = "Hi there from python"
Slist = S.split(" ")
Stuple = ("Hi","there","from","python")
print("String:", S)
print("List:", Slist)
print("Tuple:", Stuple)

#for loop on string
print("On strings:")
for c in S:
    print(c)

print("\nOn lists:")
#for loop on list
for item in Slist:
    print(item)

print("\nOn tuples:")
#for loop on list
for item in Stuple:
    print(item)
```

```
String: Hi there from python
List: ['Hi', 'there', 'from', 'python']
Tuple: ('Hi', 'there', 'from', 'python')
On strings:
H
i

t
h
e
r
e

f
r
o
m

p
y
t
h
o
n

On lists:
Hi
there
from
python

On tuples:
Hi
there
from
python
```

5.4.2 Looping over a range

It is possible to loop over a range of values with the pythom built-in function *range*. The *range* function accepts either two or three parameters (all of them are **integers**). Similarly to the slicing operator, it needs the **starting point**, **end point** and **an optional step**. Three distinct syntaxes are available:

```
range(E)         # ranges from 0 to E-1
range(S,E)       # ranges from S to E-1
range(S,E,step) # ranges from S to E-1 with +step jumps
```

Remember that *S* is **included** while *E* is **excluded**. Let's see some examples.

Example: Given a list of integers, return a list with all the even numbers.

[5]:
```
myList = [1, 7, 9, 121, 77, 82]
onlyEven = []

for i in range(0, len(myList)):  #this is equivalent to range(len(myList)):
    if myList[i] % 2 == 0 :
        onlyEven.append(myList[i])

print("original list:", myList)
print("only even numbers:", onlyEven)
```

```
original list: [1, 7, 9, 121, 77, 82]
only even numbers: [82]
```

Example: Store in a list the multiples of 19 between 1 and 100.

[6]:
```
multiples = []

for i in range(19,101,19):    # equal to: list(range(19,101,19))
    multiples.append(i)       #

print("multiples of 19: ", multiples)

#alternative way:
multiples = []
for i in range(1, (100//19) + 1):
    multiples.append(i*19)
print("multiples of 19:", multiples)
```

```
multiples of 19:  [19, 38, 57, 76, 95]
multiples of 19: [19, 38, 57, 76, 95]
```

Note: range works differently in Python 2.x and 3.x

In Python 3 the *range* function returns an iterator rather storing the entire list.

[7]:
```
#Check out the difference:
print(range(0,10))

print(list(range(0,10)))
```

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Example: Let's consider the two DNA strings s1 = "ATACATATAGGGCCAATTATTATAAGTCAC" and s2 = "CGCCACTTAAGCGCCCTGTATTAAAGTCGC" that have the same length. Let's create a third string out such that $out[i]$ is "|" if $s1[i] == s2[i]$, " " otherwise.

[8]:
```
s1 = "ATACATATAGGGCCAATTATTATAAGTCAC"
s2 = "CGCCACTTAAGCGCCCTGTATTAAAGTCGC"

outSTR = ""
for i in range(len(s1)):
    if s1[i] == s2[i]:
        outSTR = outSTR + "|"
    else:
        outSTR = outSTR + " "

print(s1)
print(outSTR)
print(s2)
```

```
ATACATATAGGGCCAATTATTATAAGTCAC
   ||  || |  |  |   |  ||||| |
CGCCACTTAAGCGCCCTGTATTAAAGTCGC
```

5.4.3 Nested for loops

In some occasions it is useful to nest one (or more) for loops into another one. The basic syntax is:

```
for i in collection:
    for j in another_collection:
        #do some stuff with i and j
```

Example:

Given the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ stored as a list of lists (i.e. matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]].

Print it out as:
$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

[9]:
```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for i in range(len(matrix)):
    line = ""
    for j in range(len(matrix[i])):
        line = line + str(matrix[i][j]) + " " #note int --> str conversion!
    print(line)


# Without nested for (but this is not exactly the same).
# NOTE: cannot do print(" ".join(row)) because we have integers

#for row in matrix:
#    print(" ".join(str(row)))
```

(continues on next page)

(continued from previous page)

```
#
#Outputs:
#[ 1 ,   2 ,   3 ]
#[ 4 ,   5 ,   6 ]
#[ 7 ,   8 ,   9 ]
```

```
1 2 3
4 5 6
7 8 9
```

5.4.4 While loops

The **for** loop is great **when we have to iterate over a finite sequence of elements**. But when one needs **to loop until a specific condition holds true**, another construct must be used: the **while** statement. The loop will end when the condition becomes false.

The basic syntax is the following:

```
while condition:

    #do something
    #update the value of condition
```

An example follows:

[10]:
```
i = 0
while i < 5:
    print("i now is:", i)
    i = i + 1 #THIS IS VERY IMPORTANT!
```

```
i now is: 0
i now is: 1
i now is: 2
i now is: 3
i now is: 4
```

Note that if *condition* is false at the beginning the block of code is never executed.

Note: The loop will continue until *condition* holds true and the only code executed is the block defined through the indentation. This block of code must update the value of condition otherwise the interpreter will get stuck in the loop and will never exit.

We can combine *for* loops and *while* loops one into the code block of the other:

[11]:
```
for i in range(1,10):
    j = 1
    output = ""
    while j<= i:
        output = str(j) + " " + output
        j = j + 1
    print(output)
```

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
```

Note the way the numbers are concatenated together to form the output string. Check the difference of the previous code to the following:

[12]:
```
for i in range(1,10):
    j = 1
    while j<=i:
        print(j, end = " ")
        j = j + 1
    print("")
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

5.5 Exercises

1. Given the integer 134479170, print if it is divisible for the numbers from 2 to 16. Hint: use for and if.

Show/Hide Solution

[13]:
```
val = 134479170

flag = False #boolean variable to monitor
             #if we found at least 1 divisor

for divisor in range(2,17): #2 included, 17 excluded
    if val % divisor == 0:
        #Print the divisor
        print(val, " can be divided by ", divisor)
        #Give proof that it is a divisor
        print("\t", divisor, "*", val//divisor, "=", val )
        flag = True

if flag == False: #this is false if no divisors have been found
    print("Sorry, could not find divisors")
```

```
134479170  can be divided by  2
         2 * 67239585 = 134479170
134479170  can be divided by  3
         3 * 44826390 = 134479170
```

(continues on next page)

(continued from previous page)

```
134479170  can be divided by  5
         5 * 26895834 = 134479170
134479170  can be divided by  6
         6 * 22413195 = 134479170
134479170  can be divided by  7
         7 * 19211310 = 134479170
134479170  can be divided by  9
         9 * 14942130 = 134479170
134479170  can be divided by  10
         10 * 13447917 = 134479170
134479170  can be divided by  14
         14 * 9605655 = 134479170
134479170  can be divided by  15
         15 * 8965278 = 134479170
```

2. Given the DNA string

```
DNA="GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC"
```

write a Python script that reverse-complements it. To reverse-complement a string of DNA, one needs to replace any A with T, T with A, C with G and G with C, while any other character is complemented in N. Finally, the sequence has to be reversed (e.g. the first base becomes the last). For example, ATCG becomes CGAT.

Show/Hide Solution

[14]:
```
DNA = "GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC"

revComp = ""
for base in DNA:
    if base == "T":
        revComp = "A"+ revComp
    elif base == "A":
        revComp = "T"+ revComp
    elif base == "C":
        revComp = "G"+ revComp
    elif base == "G":
        revComp = "C" + revComp
    else:
        revComp = "N" + revComp

print("5'-", DNA, "-3'")
print("3'-", revComp, "-5'")
```

```
5'- GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC -3'
3'- GGGGTTTTTAATAGTAAGCCCGCGCGTATATATCTGTACTGATATATGTAATC -5'
```

[15]:
```
""" Another solution"""

DNA = "GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC"


dna_list = list(DNA)
print(dna_list)
compl = []
for el in dna_list:
    if el == 'A':
        compl.append('T')
```

(continues on next page)

(continued from previous page)

```
    elif el == 'T':
        compl.append('A')
    elif el == 'C':
        compl.append('G')
    elif el == 'G':
        compl.append('C')
    else:
        compl.append('N')
print(compl)
compl.reverse()
print(compl)
revComp = "".join(compl)
print("\n5'-", DNA, "-3'")
print("3'-", revComp, "-5'")
```

```
['G', 'A', 'T', 'T', 'A', 'C', 'A', 'T', 'A', 'T', 'A', 'T', 'C', 'A', 'G', 'T', 'A',
↪'C', 'A', 'G', 'A', 'T', 'A', 'T', 'A', 'T', 'A', 'C', 'G', 'C', 'G', 'C', 'G', 'G',
↪ 'G', 'C', 'T', 'T', 'A', 'C', 'T', 'A', 'T', 'T', 'A', 'A', 'A', 'A', 'A', 'C', 'C
↪', 'C', 'C']
['C', 'T', 'A', 'A', 'T', 'G', 'T', 'A', 'T', 'A', 'T', 'A', 'G', 'T', 'C', 'A', 'T',
↪'G', 'T', 'C', 'T', 'A', 'T', 'A', 'T', 'A', 'T', 'G', 'C', 'G', 'C', 'G', 'C', 'C',
↪ 'C', 'G', 'A', 'A', 'T', 'G', 'A', 'T', 'A', 'A', 'T', 'T', 'T', 'T', 'T', 'G', 'G
↪', 'G', 'G']
['G', 'G', 'G', 'G', 'T', 'T', 'T', 'T', 'T', 'A', 'A', 'T', 'A', 'G', 'T', 'A', 'A',
↪'G', 'C', 'C', 'C', 'G', 'C', 'G', 'C', 'G', 'T', 'A', 'T', 'A', 'T', 'A', 'T', 'C',
↪ 'T', 'G', 'T', 'A', 'C', 'T', 'G', 'A', 'T', 'A', 'T', 'A', 'T', 'G', 'T', 'A', 'A
↪', 'T', 'C']

5'- GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC -3'
3'- GGGGTTTTTAATAGTAAGCCCGCGCGTATATATCTGTACTGATATATGTAATC -5'
```

3. Count how many of the first 100 integers are divisible by 2, 3, 5, 7 but not by 10 and print these counts. Be aware that a number can be divisible by more than one of these numbers (e.g. 6) and therefore it must be counted as divisible by all of them (e.g. 6 must be counted as divisible by 2 and 3).

Show/Hide Solution

[1]:
```
cnts = [0,0,0,0] #cnts[0] counts for 2, cnts[1] counts for 3...
vals = [2,3,5,7]
for i in range(1,101):
    if  i % 2 == 0 and i % 10 != 0:
        cnts[0] = cnts[0] + 1

    if i % 3 == 0 and i % 10 != 0:
        cnts[1] = cnts[1] + 1

    if i % 5 == 0 and i % 10 != 0:
        cnts[2] = cnts[2] + 1

    if i % 7 == 0 and i % 10 != 0:
        cnts[3] = cnts[3] + 1

for i in range(len(cnts)):
    print(cnts[i], " numbers are divisible by ", vals[i], "(but not 10) in the first␣
↪100")
```

(continues on next page)

(continued from previous page)

```
"""Another version, more compact.
   Makes use of two for loops one into the other
"""
print("\n----------------------------------------\n")
cnts = [0,0,0,0] #cnts[0] counts for 2, cnts[1] counts for 3...
vals = [2,3,5,7]
for i in range(1,101):
    for j in range(len(vals)):
        if i % vals[j] == 0 and i % 10 != 0:
            cnts[j] += 1
for i in range(0, len(cnts)):
    print(cnts[i], " numbers are divisible by ", vals[i], "(but not 10) in the first␣
↪100")
```

```
40  numbers are divisible by  2 (but not 10) in the first 100
30  numbers are divisible by  3 (but not 10) in the first 100
10  numbers are divisible by  5 (but not 10) in the first 100
13  numbers are divisible by  7 (but not 10) in the first 100

----------------------------------------

40  numbers are divisible by  2 (but not 10) in the first 100
30  numbers are divisible by  3 (but not 10) in the first 100
10  numbers are divisible by  5 (but not 10) in the first 100
13  numbers are divisible by  7 (but not 10) in the first 100
```

4. Write a python script that creates the following pattern:

```
+
++
+++
++++
+++++
++++++
+++++++ <-- 7
++++++
+++++
++++
+++
++
+
```

Show/Hide Solution

[17]:
```
outStr = ""
for i in range(0,7):
    outStr = ""
    for j in range(0,i+1):
        outStr = outStr + "+"
    if i == 6:
        outStr = outStr + " <-- 7"

    print(outStr) # print the ascending part

for i in range(1,7):
    outStr = ""
```

(continues on next page)

(continued from previous page)

```
    for j in range(0, 7-i):
        outStr = outStr + "+"
    print(outStr) #print the descending part
```

```
+
++
+++
++++
+++++
++++++
+++++++ <-- 7
++++++
+++++
++++
+++
++
+
```

5. Given the following fastq entry:

```
@HWI-ST1296:75:C3F7CACXX:1:1101:19142:14904
CCAACAACTTTGACGCTAAGGATAGCTCCATGGCAGCATATCTGGCACAA
+
FHIIJIJJGIJJJJJ1HHHFFFFFEE:;CIDDDDDDDDDDDDEDDD-./0
```

Store the sequence and the quality in two strings. Create a list with all the quality phred scores (given a quality character "X" the phred score is: ord("X") -33. Finally print all the bases that have quality lower than 25, reporting the base, its position, quality character and phred score.

Output example: base: C index: 15 qual:1 phred: 16.

Show/Hide Solution

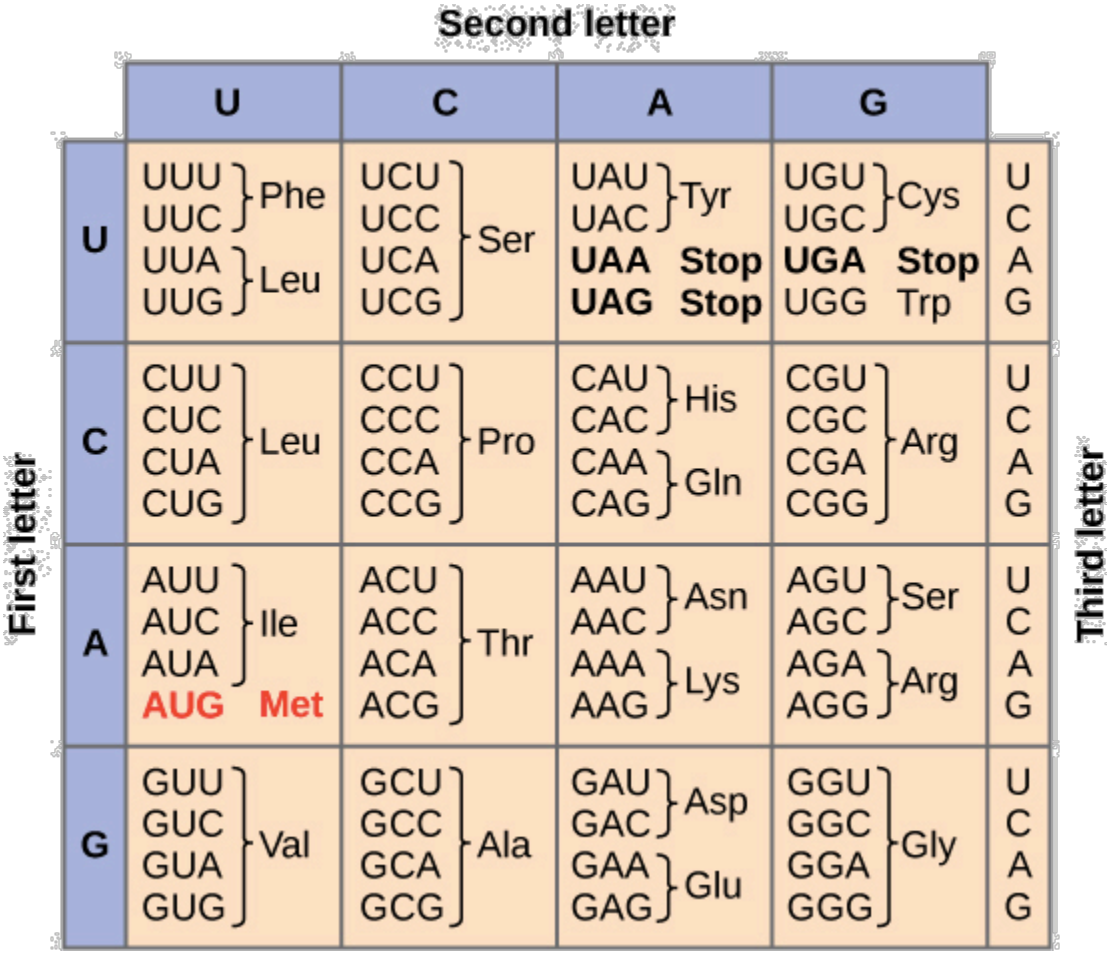
[18]:
```
entry = """@HWI-ST1296:75:C3F7CACXX:1:1101:19142:14904
CCAACAACTTTGACGCTAAGGATAGCTCCATGGCAGCATATCTGGCACAA
+
FHIIJIJJGIJJJJJ1HHHFFFFFEE:;CIDDDDDDDDDDDDEDDD-./0"""

lines = entry.split("\n")
seq = lines[1]
qual = lines[3]

phredScores = []
for i in range(len(qual)):
    phredScores.append(ord(qual[i]) - 33)


for i in range(len(seq)):
    if phredScores[i] <25:
        print("base:",seq[i],"index:",i,"qual:",qual[i],"phredScore:",phredScores[i])
```

```
base: C index: 15 qual: 1 phredScore: 16
base: A index: 46 qual: - phredScore: 12
base: C index: 47 qual: . phredScore: 13
base: A index: 48 qual: / phredScore: 14
base: A index: 49 qual: 0 phredScore: 15
```

6. Given the following sequence:

```
seq=AUGCUGUCUCCCUCACUGUAUGUAAAUUGCAUCUAGAAUAGCA
UCUGGAGCACUAAUUGACACAUAGUGGGUAUCAAUUAUUA
UUCCAGGUACUAGAGAUACCUGGACCAUUAACGGAUAAAU
AGAAGAUUCAUUUGUUGAGUGACUGAGGAUGGCAGUUCCU
GCUACCUUCAAGGAUCUGGAUGAUGGGGAGAAACAGAGAA
CAUAGUGUGAGAAUACUGUGGUAAGGAAAGUACAGAGGAC
UGGUAGAGUGUCUAACCUAGAUUUGGAGAAGGACCUAGAA
GUCUAUCCCAGGGAAAUAAAAAUCUAAGCUAAGGUUUGAG
GAAUCAGUAGGAAUUGGCAAAGGAAGGACAUGUUCCAGAU
GAUAGGAACAGGUUAUGCAAAGAUCCUGAAAUGGUCAGAG
CUUGGUGCUUUUUGAGAACCAAAAGUAGAUUGUUAUGGAC
CAGUGCUACUCCCUGCCUCUUGCCAAGGGACCCCGCCAAG
CACUGCAUCCCUUCCCUCUGACUCCACCUUUCCACUUGCC
CAGUAUUGUUGGUGU
```

and considering the genetic code and the first forward open reading frame (i.e. the string as it is **remembering to remove newlines**).



1. How many start codons are present in the whole sequence (i.e. AUG)?
2. How many stop codons (i.e. UAA,UAG, UGA)
3. Create another string in which any codon with except the start and stop codons are substituted with "—" and print the resulting string.

Show/Hide Solution

[19]:
```
seq ="""AUGCUGUCUCCCUCACUGUAUGUAAAUUGCAUCUAGAAUAGCA
UCUGGAGCACUAAUUGACACAUAGUGGGUAUCAAUUAUUA
UUCCAGGUACUAGAGAUACCUGGACCAUUAACGGAUAAAU
AGAAGAUUCAUUUGUUGAGUGACUGAGGAUGGCAGUUCCU
GCUACCUUCAAGGAUCUGGAUGAUGGGGAGAAACAGAGAA
CAUAGUGUGAGAAUACUGUGGUAAGGAAAGUACAGAGGAC
UGGUAGAGUGUCUAACCUAGAUUUGGAGAAGGACCUAGAA
GUCUAUCCCAGGGAAAUAAAAAUCUAAGCUAAGGUUUGAG
GAAUCAGUAGGAAUUGGCAAAGGAAGGACAUGUUCCAGAU
GAUAGGAACAGGUUAUGCAAAGAUCCUGAAAUGGUCAGAG
CUUGGUGCUUUUUGAGAACCAAAAGUAGAUUGUUAUGGAC
CAGUGCUACUCCCUGCCUCUUGCCAAGGGACCCCGCCAAG
CACUGCAUCCCUUCCCUCUGACUCCACCUUUCCACUUGCC
CAGUAUUGUUGGUG"""

seq = seq.replace("\n","")


countStart = 0
countEnd = 0

newSeq = ""
i = 0
while i < len(seq):
    codon = seq[i:i+3]
    if codon == "AUG":
            countStart = countStart + 1

    elif codon == "UAA" or codon == "UAG" or codon == "UGA":
        countEnd = countEnd + 1
    else:
        codon = "---"

    newSeq = newSeq + codon
    i = i + 3


print("\nNumber of start codons:", countStart)
print("Number of end codons:", countEnd)
print("\n")
print(seq)
print("\n")
print(newSeq)
```

```
Number of start codons: 2
Number of end codons: 12


AUGCUGUCUCCCUCACUGUAUGUAAAUUGCAUCUAGAAUAGCAUCUGGAGCACUAAUUGACACAUAGUGGGUAUCAAUUAUUAUUCCAGGUACUAGAGAUA


AUG------------------------------UAG--------------------UGA--------------------------
↪-------UAG---------------UAA-----------------------UGA-----------------------------
↪-------------UGA-------------------------UGA---------------------------------UAG-----
↪-UAA-------------------------------------------------------UAA---------------UAG-----
↪-----------------------------------------------------AUG--------------------------
↪-------------------------------------------------------------------------------UGA--
↪-------------------------------
```

(continues on next page)

(continued from previous page)

7. Playing time! Write a python scripts that:
    1. Picks a random number from 1 to 10, with: import random myInt = random.randint(1,10)
    2. Asks the user to guess a number and checks if the user has guessed the right one
    3. If the guess is right the program will stop with a congratulation message
    4. If the guess is wrong the program will continue asking a number, reporting the numbers already guessed (hint: store them in a list and print it).
    5. Modify the program to notify the user if he/she inputs the same number more than once.

Show/Hide Solution

[2]:
```
import random
myInt = random.randint(1,10)
guessedNumbers = []

found = False
while found == False:
    userInt = int(input("Guess a number from 1 to 10: "))
    if userInt == myInt:
        print("Congratulations. The number I guessed was ", myInt)
        found = True
    else:
        if userInt in guessedNumbers:
            print("You already guessed ", userInt)
        else:
            guessedNumbers.append(userInt)
        guessedNumbers.sort()
        print("Nope, try again. So far you guessed: ", guessedNumbers)
```

```
Guess a number from 1 to 10: 7
Nope, try again. So far you guessed:  [7]
Guess a number from 1 to 10: 3
Nope, try again. So far you guessed:  [3, 7]
Guess a number from 1 to 10: 1
Nope, try again. So far you guessed:  [1, 3, 7]
Guess a number from 1 to 10: 2
Nope, try again. So far you guessed:  [1, 2, 3, 7]
Guess a number from 1 to 10: 5
Nope, try again. So far you guessed:  [1, 2, 3, 5, 7]
Guess a number from 1 to 10: 4
Congratulations. The number I guessed was  4
```

CHAPTER

SIX

PRACTICAL 5

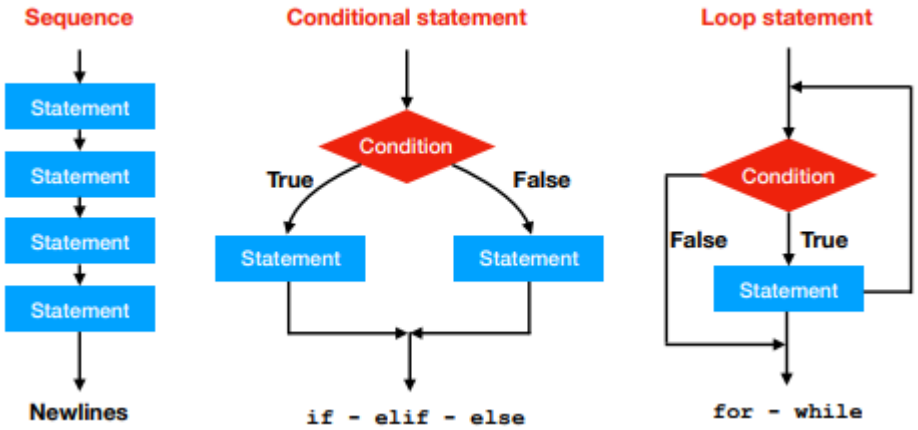
In this practical we will add some information on loops and introduce dictionaries.

6.1 Slides

The slides of the introduction can be found here: Intro

6.2 More on loops

As seen in the previous practical and in the lecture, there are three different ways of execution flow:



We have already seen the *if*, *for* and *while* loops and their variants. Please remember that the code block of each of these statements is defined by the *indentation* (i.e. spacing).

6.2.1 Ternary operator

In some cases it is handy to be able to initialize a variable depending on the value of another one.

Example: The discount rate applied to a purchase depends on the amount of the sale. Create a variable *discount* setting its value to 0 if the variable *amount* is lower than 100 euros, to 10% if it is higher.

[1]:
```
amount = 110
discount = 0

if amount >100:
    discount = 0.1
else:
    discount = 0 # not necessary

print("Total amount:", amount, "discount:", discount)
```

```
Total amount: 110 discount: 0.1
```

The previous code can be written more coincisely as:

[2]:
```
amount = 110
discount = 0.1 if amount > 100 else 0
print("Total amount:", amount, "discount:", discount)
```

```
Total amount: 110 discount: 0.1
```

The basic syntax of the ternary operator is:

```
variable = value if condition else other_value
```

meaning that the *variable* is initialized to *value* if the *condition* holds, otherwise to *other_value*.

Python also allows in line operations separated by a ";". This is a compact way of specifying several instructions on a single line, but it makes the code more difficult to read.

[3]:
```
a = 10; b = a + 1; c = b +2
print(a,b,c)
```

```
10 11 13
```

Note: Although the ternary operator and in line operations are sometimes useful and less verbose than the explicit definition, they are considered "non-pythonic" and advised against.

6.2.2 Break and continue

Sometimes it is useful to skip an entire iteration of a loop or end the loop before its supposed end. This can be achieved with two different statements: **continue** and **break**.

Continue statement

Within a **for** or **while** loop, **continue** makes the interpreter skip that iteration and move on to the next.

Example: Print all the odd numbers from 1 to 20.

[4]:
```
#Two equivalent ways
#1. Testing remainder == 1
for i in range(21):
    if i % 2 == 1:
        print(i, end = " ")

print("")

#2. Skipping if remainder == 0 in for
for i in range(21):
    if i % 2 == 0:
        continue
    print(i, end = " ")
```

```
1 3 5 7 9 11 13 15 17 19
1 3 5 7 9 11 13 15 17 19
```

Continue can be used also within while loops but we need to be careful to update the value of the variable before reaching the continue statement or we will get stuck in never-ending loops. **Example:** Print all the odd numbers from 1 to 20.

[ ]:
```
#Wrong code:
i = 0
while i < 21:
    if i % 2 == 0:
        continue
    print(i, end = " ")
    i = i + 1 # NEVER EXECUTED IF i % 2 == 0!!!!
```

a possible correct solution using while:

[5]:
```
i = -1
while i< 20:        #i is incremented in the loop, so 20!!!
    i = i + 1         #the variable is updated no matter what
    if i % 2 == 0:
        continue
    print(i, end = " ")
```

```
1 3 5 7 9 11 13 15 17 19
```

Break statement

Within a **for** or **while** loop, **break** makes the interpreter exit the loop and continue with the sequential execution. Sometimes it is useful to get out of the loop if to complete our task we do not need to get to the end of it.

Example: Given the following list of integers [1,5,6,4,7,1,2,3,7] print them until a number already printed is found.

[6]:
```
L = [1,5,6,4,7,1,2,3,7]
found = []
for i in L:
    if i in found:
        break
```

(continues on next page)

(continued from previous page)

```
    found.append(i)
    print(i, end = " ")
```

```
1 5 6 4 7
```

Example: Pick a random number from 1 and 50 and count how many times it takes to randomly choose number 27. Limit the number of random picks to 40 (i.e. if more than 40 picks have been done and 27 has not been found exit anyway with a message).

[7]:
```
import random

iterations = 1
picks = []
while iterations <= 40:
    pick = random.randint(1,50)
    picks.append(pick)

    if pick == 27:
        break
    iterations += 1   #equal to: iterations = iterations + 1

if iterations == 41:
    print("Sorry number 27 was never found!")
else:
    print("27 found in ", iterations, "iterations")

print(picks)
```

```
27 found in  9 iterations
[9, 16, 8, 37, 42, 7, 14, 28, 27]
```

An alternative way without using the break statement makes use of a *flag* variable (that when changes value will make the loop end):

[8]:
```
import random
found = False # This is called flag
iterations = 1
picks = []
while iterations <= 40 and found == False: #the flag is used to exit
    pick = random.randint(1,50)
    picks.append(pick)
    if pick == 27:
        found = True     #update the flag, will exit at next iteration
    iterations += 1

if iterations == 41 and not found:
    print("Sorry number 27 was never found!")
else:
    print("27 found in ", iterations -1, "iterations")

print(picks)
```

```
Sorry number 27 was never found!
[14, 32, 28, 25, 49, 28, 31, 6, 11, 17, 21, 13, 35, 15, 3, 30, 34, 19, 38, 20, 47, 38,
↪ 14, 42, 32, 19, 23, 49, 40, 21, 17, 35, 47, 1, 39, 41, 31, 33, 21, 35]
```

6.2.3 List comprehension

List comprehension is a quick way of creating a list. The resulting list is normally obtained by applying a function or a method to the elements of another list that **remains unchanged**.

The basic syntax is:

```
new_list = [ some_function (x) for x in start_list]
```

or

```
new_list = [ x.some_method() for x in start_list]
```

List comprehension can also be used to filter elements of a list and produce another list as sublist of the first one (**remember that the original list is not changed**).

In this case the syntax is:

```
new_list = [ some_function (x) for x in start_list if condition]
```

or

```
new_list = [ x.some_method() for x in start_list if condition]
```

where the element x in start_list becomes part of new_list if and only if the condition holds True.

Let's see some examples:

Example: Given a list of strings ["hi", "there", "from", "python"] create a list with the length of the corresponding element (i.e. the one with the same index).

[9]:
```
elems = ["hi", "there", "from", "python"]

newList = [len(x) for x in elems]

for i in range(0,len(elems)):
    print(elems[i], " has length ", newList[i])
```

```
hi  has length  2
there  has length  5
from  has length  4
python  has length  6
```

Example: Given a list of strings ["dog", "cat", "rabbit", "guinea pig", "hamster", "canary", "goldfish"] create a list with the elements starting with a "c" or "g".

[10]:
```
pets = ["dog", "cat", "rabbit", "guinea pig", "hamster", "canary", "goldfish"]

cg_pets = [x for x in pets if x.startswith("c") or x.startswith("g")]

print("Original:")
print(pets)
print("Filtered:")
print(cg_pets)
```

```
Original:
['dog', 'cat', 'rabbit', 'guinea pig', 'hamster', 'canary', 'goldfish']
Filtered:
['cat', 'guinea pig', 'canary', 'goldfish']
```

Example: Create a list with all the numbers divisible by 17 from 1 to 200.

[11]:
```
values = [ x for x in range(1,200) if x % 17 == 0]
print(values)
```

```
[17, 34, 51, 68, 85, 102, 119, 136, 153, 170, 187]
```

Example: Transpose the matrix $\begin{bmatrix} 1 & 10 \\ 2 & 20 \\ 3 & 30 \\ 4 & 40 \end{bmatrix}$ stored as a list of lists (i.e. matrix = [[1, 10], [2,20], [3,30], [4,40]]). The output matrix should be: $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 10 & 20 & 30 & 40 \end{bmatrix}$, represented as [[1, 2, 3, 4], [10, 20, 30, 40]]

[12]:
```
matrix = [[1, 10], [2,20], [3,30], [4,40]]
print(matrix)
transpose = [[row[i] for row in matrix] for i in range(2)]
print (transpose)
```

```
[[1, 10], [2, 20], [3, 30], [4, 40]]
[[1, 2, 3, 4], [10, 20, 30, 40]]
```

Example: Given the list: [“Hotel”, “Icon”,” Bus”,”Train”, “Hotel”, “Eye”, “Rain”, “Elephant”] create a list with all the first letters.

[13]:
```
myList = ["Hotel", "Icon"," Bus","Train", "Hotel", "Eye", "Rain", "Elephant"]
initials = [x[0] for x in myList]

print(myList)
print(initials)
print("".join(initials))
```

```
['Hotel', 'Icon', ' Bus', 'Train', 'Hotel', 'Eye', 'Rain', 'Elephant']
['H', 'I', ' ', 'T', 'H', 'E', 'R', 'E']
HI THERE
```

With list comprehension we can copy a list into another one, but this is a shallow copy:

[10]:
```
a = [1,2,3]
b = [a, [[a]]]
print("B:", b)
c = [x for x in b]
print("C:", c)
a.append(4)
print("B now:" , b)
print("C now:", c)
```

```
B: [[1, 2, 3], [[[1, 2, 3]]]]
C: [[1, 2, 3], [[[1, 2, 3]]]]
B now: [[1, 2, 3, 4], [[[1, 2, 3, 4]]]]
C now: [[1, 2, 3, 4], [[[1, 2, 3, 4]]]]
```

6.3 Dictionaries

A **dictionary** is a map between one object, the **key**, and another object, the **value**. Dictionaries are **mutable objects** and contain sequences of mappings *key –> object* but there is not specific ordering among them. Dictionaries are defined using the curly braces **{key1 : value1, key2 : value2}** and **:** to separate keys from values.

Some examples on how to define dictionaries follow:

[14]:
```
first_dict = {"one" : 1, "two": 2, "three" : 3, "four" : 4}
print("First:", first_dict)

empty_dict = dict()
print("Empty:",empty_dict)

second_dict = {1 : "one", 2 : "two", "three" :3 } #BAD IDEA BUT POSSIBLE!!!
print(second_dict)

third_dict = dict(zip(["one","two","three","four"],[1,2,3,4]))
print(third_dict)
print(first_dict == third_dict)
```

```
First: {'one': 1, 'two': 2, 'three': 3, 'four': 4}
Empty: {}
{1: 'one', 2: 'two', 'three': 3}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
True
```

Note that there is no ordering of the keys, and that the order in which they have been inserted is not preserved. Moreover, keys and values can be dishomogeneous (e.g. keys can be strings and values integers). An interesting case is *third_dict* where the function *zip* followed by *dict* is used to map the keys of the first list into the values present in the second.

Note that keys can be **dishomogeneous**, even though this is a bad idea normally. The only requirement for the **keys** is that they **must be immutable objects**. Trying to use a mutable object as a key will make the interpreter crash with the error: **unhashable type**. Finally, keys must be unique. We cannot associate more than one value to the same key.

[15]:
```
a = (1,2,3) #a,b are tuples: hence immutable
b = (1,3,5)

my_dict = {a : 6, b : 9 }
print(my_dict)

c = [1,2,3] #c,d are lists: hence mutable
d = [1,3,5]

dict2 = {c : 6, d : 9}
print(dict2)
```

```
{(1, 2, 3): 6, (1, 3, 5): 9}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-0fe98c7f5acd> in <module>
      8 d = [1,3,5]
      9
---> 10 dict2 = {c : 6, d : 9}
     11 print(dict2)

TypeError: unhashable type: 'list'
```

6.3.1 Functions working on dictionaries

As for the other data types, python provides several operators that can be applied to dictionaries. The following operators are available and they basically work as in lists. The only exception being that the operator **in** checks whether the specified object is present among the **keys**.

Result	Operator	Meaning
`bool`	`obj in dict`	Return `True` if a key is present in the dictionary
`int`	`len(dict)`	Return the number of elements in the dictionary
`obj`	`dict[obj]`	Read the value associate with a key
	`dict[obj] = obj`	Add or modify the value associated with a key

Some usage examples follow:

[16]:
```
myDict = {"one" : 1, "two" : 2, "twentyfive" : 25}

print(myDict)
myDict["ten"] = 10
myDict["twenty"] = 20
print(myDict)
myDict["ten"] = "10-again"
print(myDict)
print("The dictionary has ", len(myDict), " elements")
print("The value of \"ten\" is:", myDict["ten"])
print("The value of \"two\" is:", myDict["two"])

print("Is \"twentyfive\" in dictionary?", "twentyfive" in myDict)
print("Is \"seven\" in dictionary?", "seven" in myDict)
```

```
{'one': 1, 'two': 2, 'twentyfive': 25}
{'one': 1, 'two': 2, 'twentyfive': 25, 'ten': 10, 'twenty': 20}
{'one': 1, 'two': 2, 'twentyfive': 25, 'ten': '10-again', 'twenty': 20}
The dictionary has  5  elements
The value of "ten" is: 10-again
The value of "two" is: 2
Is "twentyfive" in dictionary? True
Is "seven" in dictionary? False
```

6.3.2 Dictionary methods

Recall what seen in the lecture, the following methods are available for dictionaries:

Return	Method	Meaning
`list`	`dict.keys()`	Returns the list of the keys that are present in the dictionary
`list`	`dict.values()`	Returns the list of the values that are present in the dictionary
`list of tuples`	`dict.items()`	Returns the list of pairs (key, value) that are present in the dictionary

These methods are new to dictionaries and can be used to loop through the elements in them.

ERRATUM: `dict.keys()` returns a `dict_keys` object not a list. To cast it to list, we need to call `list(dict.keys())`. The same applies to `dict.values()` that returns a `dict_values` object that needs conversion to a list with `list(dict.values())`.

[17]:
```
D = {"k1" : 1, "k2" : 2 , "k3" : 3}

print("keys:" , D.keys(), "values:", D.values())
print("")
print("keys:", list(D.keys()), "values:", list(D.values()))
```

```
keys: dict_keys(['k1', 'k2', 'k3']) values: dict_values([1, 2, 3])

keys: ['k1', 'k2', 'k3'] values: [1, 2, 3]
```

Example Given the protein sequence below, store in a dictionary all the aminoacids present and count how many times they appear. Finally print out the stats (e.g. how many amino-acids are present, the most frequent, the least frequent and the frequency of all of them **in alphabetical order**).

```
>sp|P00517|KAPCA_BOVIN cAMP-dependent protein kinase catalytic subunit alpha
MGNAAAAKKGSEQESVKEFLAKAKEDFLKKWENPAQNTAHLDQFERIKTLGTGSFGRVML
VKHMETGNHYAMKILDKQKVVKLKQIEHTLNEKRILQAVNFPFLVKLEFSFKDNSNLYMV
MEYVPGGEMFSHLRRIGRFSEPHARFYAAQIVLTFEYLHSLDLIYRDLKPENLLIDQQGY
IQVTDFGFAKRVKGRTWTLCGTPEYLAPEIILSKGYNKAVDWWALGVLIYEMAAGYPPFF
ADQPIQIYEKIVSGKVRFPSHFSSDLKDLLRNLLQVDLTKRFGNLKNGVNDIKNHKWFAT
TDWIAIYQRKVEAPFIPKFKGPGDTSNFDDYEEEEIRVSINEKCGKEFSEF
```

[1]:
```
protein = """MGNAAAAKKGSEQESVKEFLAKAKEDFLKKWENPAQNTAHLDQFERIKTLGTGSFGRVML
VKHMETGNHYAMKILDKQKVVKLKQIEHTLNEKRILQAVNFPFLVKLEFSFKDNSNLYMV
MEYVPGGEMFSHLRRIGRFSEPHARFYAAQIVLTFEYLHSLDLIYRDLKPENLLIDQQGY
IQVTDFGFAKRVKGRTWTLCGTPEYLAPEIILSKGYNKAVDWWALGVLIYEMAAGYPPFF
ADQPIQIYEKIVSGKVRFPSHFSSDLKDLLRNLLQVDLTKRFGNLKNGVNDIKNHKWFAT
TDWIAIYQRKVEAPFIPKFKGPGDTSNFDDYEEEEIRVSINEKCGKEFSEF"""

protein = protein.replace("\n","")

print(protein)
```

(continues on next page)

(continued from previous page)

```
amino_acids = dict()

for a in protein:
    if a in amino_acids:
        amino_acids[a] = amino_acids[a] + 1 # amino_acids[a] += 1
    else:
        amino_acids[a] = 1


num_aminos = len(amino_acids)

print("The number of amino-acids present is ", num_aminos)
#let's get all aminoacids
#and sort them alphabetically
a_keys = list(amino_acids.keys())

a_keys.sort()

# Another example of dictionaries
mostF = {"frequency" : -1, "aminoacid" : "-"}
leastF = {"frequency" : len(protein), "aminoacid" : "-"}

for a in a_keys:
    freq = amino_acids[a]
    if(mostF["frequency"] < freq):
        mostF["frequency"] = freq
        mostF["aminoacid"] = a

    if(leastF["frequency"] > freq):
        leastF["frequency"] = freq
        leastF["aminoacid"] = a
    print(a, " is present", freq, "times")

print("Amino", leastF["aminoacid"], "has the lowest freq. (",leastF["frequency"],")")
print("Amino", mostF["aminoacid"], "has the highest freq. (",mostF["frequency"],")")
```

```
MGNAAAAKKGSEQESVKEFLAKAKEDFLKKWENPAQNTAHLDQFERIKTLGTGSFGRVMLVKHMETGNHYAMKILDKQKVVKLKQIEHTLNEKRILQAVNF
The number of amino-acids present is  20
A  is present 23 times
C  is present 2 times
D  is present 18 times
E  is present 27 times
F  is present 25 times
G  is present 22 times
H  is present 9 times
I  is present 21 times
K  is present 34 times
L  is present 32 times
M  is present 8 times
N  is present 17 times
P  is present 14 times
Q  is present 14 times
R  is present 15 times
S  is present 16 times
T  is present 14 times
V  is present 20 times
W  is present 6 times
```

(continues on next page)

(continued from previous page)

```
Y  is present 14 times
Amino C has the lowest freq. ( 2 )
Amino K has the highest freq. ( 34 )
```

Important NOTE. Accessing a value through the key of a dictionary requires that the pair key-value one searches for is present in the dictionary. If the searched key is not present the interpreter crashes out throwing a KeyError as follows:

[19]:
```
myDict = {"one" : 1, "two" : 2, "three" : 3}

print(myDict["one"])
print(myDict["seven"])
```

```
1
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-19-a05b31e54a02> in <module>
      2
      3 print(myDict["one"])
----> 4 print(myDict["seven"])

KeyError: 'seven'
```

This could be avoided by checking that the key is present in the dictionary beforehand:

[20]:
```
myDict = {"one" : 1, "two" : 2, "three" : 3}

search_keys = ["one", "seven"]

for s in search_keys:
    if s in myDict:
        print("key:", s, "value:", myDict[s])
    else:
        print("key", s, "not found in dictionary")
```

```
key: one value: 1
key seven not found in dictionary
```

or by using the dictionary method **get** that has two inputs: the key and a default value to return in case key is not present in the dictionary:

[21]:
```
myDict = {"one" : 1, "two" : 2, "three" : 3}

search_keys = ["one", "seven"]

for s in search_keys:
    print("key:", s, "value:", myDict.get(s, "not found"))
```

```
key: one value: 1
key: seven value: not found
```

6.4 Exercises

1. Given the following two lists of integers: [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100] and [44,32,7, 100, 81, 13, 1, 21, 71]:
    1. Sort the two lists
    2. Create a third list as intersection of the two lists (i.e. an element is in the intersection if it is present in both lists).
    3. Print the three lists.

Show/Hide Solution

[23]:
```
"""First solution, prints multiple times repeated elements in L1"""

L1 = [1, 13, 22, 7, 43, 81, 77, 12, 15,21, 84,100]
L2 = [44,32,7, 100, 81, 13, 1, 21, 71]

L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]

print("L1:    ", L1)
print("L2:    ", L2)
print("inters:", intersection)

L1 = [1, 9, 1, 7, 44, 9, 9, 9, 81, 77, 12, 15,21, 84,100]
L2 = [44, 32, 21, 7, 100, 81, 13, 9, 1, 21, 71]

print("\n--------------------------")
L1.sort()
L2.sort()
intersection = [x for x in L1 if x in L2]
print("L1:    ", L1)
print("L2:    ", L2)
print("inters:", intersection)

print("\n\n ------ Second solution --------")
"""Second solution, does not print multiple times repeated elements in L1"""
print("L1:    ", L1)
print("L2:    ", L2)

intersection2 = [L1[x] for x in range(len(L1)) if L1[x] in L2 and L1[x] not in␣
↪L1[x+1:]]
print("inters:", intersection2)
```

```
L1:     [1, 7, 12, 13, 15, 21, 22, 43, 77, 81, 84, 100]
L2:     [1, 7, 13, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 13, 21, 81, 100]

--------------------------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 1, 7, 9, 9, 9, 9, 21, 44, 81, 100]
```

(continues on next page)

(continued from previous page)

```
 ------ Second solution --------
L1:     [1, 1, 7, 9, 9, 9, 9, 12, 15, 21, 44, 77, 81, 84, 100]
L2:     [1, 7, 9, 13, 21, 21, 32, 44, 71, 81, 100]
inters: [1, 7, 9, 21, 44, 81, 100]
```

2. The sequence below is the Sars-Cov2 ORF1a polyprotein. 1. Count and print how many aminoacids it is composed of and 2. put in a dictionary all the indexes of the occurrences of the following four aminoacids: `TTTL`, `GFAV`, `KMLL` (i.e. the key of the dictionary is the sequence and the value is the list of all positions at which the four-mers appear).

```
ORF1a = """MESLVPGFNEKTHVQLSLPVLQVRDVLVRGFGDSVEEVLSEARQHLKDGTCGLVEVEKGVLPQLEQPYVF
IKRSDARTAPHGHVMVELVAELEGIQYGRSGETLGVLVPHVGEIPVAYRKVLLRKNGNKGAGGHSYGADL
KSFDLGDELGTDPYEDFQENWNTKHSSGVTRELMRELNGGAYTRYVDNNFCGPDGYPLECIKDLLARAGK
ASCTLSEQLDFIDTKRGVYCCREHEHEIAWYTERSEKSYELQTPFEIKLAKKFDTFNGECPNFVFPLNSI
IKTIQPRVEKKKLDGFMGRIRSVYPVASPNECNQMCLSTLMKCDHCGETSWQTGDFVKATCEFCGTENLT
KEGATTCGYLPQNAVVKIYCPACHNSEVGPEHSLAEYHNESGLKTILRKGGRTIAFGGCVFSYVGCHNKC
AYWVPRASANIGCNHTGVVGEGSEGLNDNLLEILQKEKVNINIVGDFKLNEEIAIILASFSASTSAFVET
VKGLDYKAFKQIVESCGNFKVTKGKAKKGAWNIGEQKSILSPLYAFASEAARVVRSIFSRTLETAQNSVR
VLQKAAITILDGISQYSLRLIDAMMFTSDLATNNLVVMAYITGGVVQLTSQWLTNIFGTVYEKLKPVLDW
LEEKFKEGVEFLRDGWEIVKFISTCACEIVGGQIVTCAKEIKESVQTFFKLVNKFLALCADSIIIGGAKL
KALNLGETFVTHSKGLYRKCVKSREETGLLMPLKAPKEIIFLEGETLPTEVLTEEVVLKTGDLQPLEQPT
SEAVEAPLVGTPVCINGLMLLEIKDTEKYCALAPNMMVTNNTFTLKGGAPTKVTFGDDTVIEVQGYKSVN
ITFELDERIDKVLNEKCSAYTVELGTEVNEFACVVADAVIKTLQPVSELLTPLGIDLDEWSMATYYLFDE
SGEFKLASHMYCSFYPPDEDEEEGDCEEEEFEPSTQYEYGTEDDYQGKPLEFGATSAALQPEEEQEEDWL
DDDSQQTVGQQDGSEDNQTTTIQTIVEVQPQLEMELTPVVQTIEVNSFSGYLKLTDNVYIKNADIVEEAK
KVKPTVVVNAANVYLKHGGGVAGALNKATNNAMQVESDDYIATNGPLKVGGSCVLSGHNLAKHCLHVVGP
NVNKGEDIQLLKSAYENFNQHEVLLAPLLSAGIFGADPIHSLRVCVDTVRTNVYLAVFDKNLYDKLVSSF
LEMKSEKQVEQKIAEIPKEEVKPFITESKPSVEQRKQDDKKIKACVEEVTTTLEETKFLTENLLLYIDIN
GNLHPDSATLVSDIDITFLKKDAPYIVGDVVQEGVLTAVVIPTKKAGGTTEMLAKALRKVPTDNYITTYP
GQGLNGYTVEEAKTVLKKCKSAFYILPSIISNEKQEILGTVSWNLREMLAHAEETRKLMPVCVETKAIVS
TIQRKYKGIKIQEGVVDYGARFYFYTSKTTVASLINTLNDLNETLVTMPLGYVTHGLNLEEAARYMRSLK
VPATVSVSSPDAVTAYNGYLTSSSKTPEEHFIETISLAGSYKDWSYSGQSTQLGIEFLKRGDKSVYYTSN
PTTFHLDGEVITFDNLKTLLSLREVRTIKVFTTVDNINLHTQVVDMSMTYGQQFGPTYLDGADVTKIKPH
NSHEGKTFYVLPNDDTLRVEAFEYYHTTDPSFLGRYMSALNHTKKWKYPQVNGLTSIKWADNNCYLATAL
LTLQQIELKFNPPALQDAYYRARAGEAANFCALILAYCNKTVGELGDVRETMSYLFQHANLDSCKRVLNV
VCKTCGQQQTTLKGVEAVMYMGTLSYEQFKKGVQIPCTCGKQATKYLVQQESPFVMMSAPPAQYELKHGT
FTCASEYTGNYQCGHYKHITSKETLYCIDGALLTKSSEYKGPITDVFYKENSYTTTIKPVTYKLDGVVCT
EIDPKLDNYYKKDNSYFTEQPIDLVPNQPYPNASFDNFKFVCDNIKFADDLNQLTGYKKPASRELKVTFF
PDLNGDVVAIDYKHYTPSFKKGAKLLHKPIVWHVNNATNKATYKPNTWCIRCLWSTKPVETSNSFDVLKS
EDAQGMDNLACEDLKPVSEEVVENPTIQKDVLECNVKTTEVVGDIILKPANNSLKIIEEVGHTDLMAAYV
DNSSLTIKKPNELSRVLGLKTLATHGLAAVNSVPWDTIANYAKPFLNKVVSTTTNIVTRCLNRVCTNYMP
YFFTLLLQLCTFTRSTNSRIKASMPTTIAKNTVKSVGKFCLEASFNYLKSPNFSKLINIIIWFLLLSVCL
GSLIYSTAALGVLMSNLGMPSYCTGYREGYLNSTNVTIATYCTGSIPCSVCLSGLDSLDTYPSLETIQIT
ISSFKWDLTAFGLVAEWFLAYILFTRFFYVLGLAAIMQLFFSYFAVHFISNSWLMWLIINLVQMAPISAM
VRMYIFFASFYYVWKSYVHVVDGCNSSTCMMCYKRNRATRVECTTIVNGVRRSFYVYANGGKGFCKLHNW
NCVNCDTFCAGSTFISDEVARDLSLQFKRPINPTDQSSYIVDSVTVKNGSIHLYFDKAGQKTYERHSLSH
FVNLDNLRANNTKGSLPINVIVFDGKSKCEESSAKSASVYYSQLMCQPILLLDQALVSDVGDSAEVAVKM
FDAYVNTFSSTFNVPMEKLKTLVATAEAELAKNVSLDNVLSTFISAARQGFVDSDVETKDVVECLKLSHQ
SDIEVTGDSCNNYMLTYNKVENMTPRDLGACIDCSARHINAQVAKSHNIALIWNVKDFMSLSEQLRKQIR
SAAKKNNLPFKLTCATTRQVVNVVTTKIALKGGKIVNNWLKQLIKVTLVFLFVAAIFYLITPVHVMSKHT
DFSSEIIGYKAIDGGVTRDIASTDTCFANKHADFDTWFSQRGGSYTNDKACPLIAAVITREVGFVVPGLP
GTILRTTNGDFLHFLPRVFSAVGNICYTPSKLIEYTDFATSACVLAAECTIFKDASGKPVPYCYDTNVLE
GSVAYESLRPDTRYVLMDGSIIQFPNTYLEGSVRVVTTFDSEYCRHGTCERSEAGVCVSTSGRWVLNNDY
YRSLPGVFCGVDAVNLLTNMFTPLIQPIGALDISASIVAGGIVAIVVTCLAYYFMRFRRAFGEYSHVVAF
NTLLFLMSFTVLCLTPVYSFLPGVYSVIYLYLTFYLTNDVSFLAHIQWMVMFTPLVPFWITIAYIICIST
KHFYWFFSNYLKRRVVFNGVSFSTFEEAALCTFLLNKEMYLKLRSDVLLPLTQYNRYLALYNKYKYFSGA
```

(continues on next page)

(continued from previous page)

```
MDTTSYREAACCHLAKALNDFSNSGSDVLYQPPQTSITSAVLQSGFRKMAFPSGKVEGCMVQVTCGTTTL
NGLWLDDVVYCPRHVICTSEDMLNPNYEDLLIRKSNHNFLVQAGNVQLRVIGHSMQNCVLKLKVDTANPK
TPKYKFVRIQPGQTFSVLACYNGSPSGVYQCAMRPNFTIKGSFLNGSCGSVGFNIDYDCVSFCYMHHMEL
PTGVHAGTDLEGNFYGPFVDRQTAQAAGTDTTITVNVLAWLYAAVINGDRWFLNRFTTTLNDFNLVAMKY
NYEPLTQDHVDILGPLSAQTGIAVLDMCASLKELLQNGMNGRTILGSALLEDEFTPFDVVRQCSGVTFQS
AVKRTIKGTHHWLLLTILTSLLVLVQSTQWSLFFFLYENAFLPFAMGIIAMSAFAMMFVKHKHAFLCLFL
LPSLATVAYFNMVYMPASWVMRIMTWLDMVDTSLSGFKLKDCVMYASAVVLLILMTARTVYDDGARRVWT
LMNVLTLVYKVYYGNALDQAISMWALIISVTSNYSGVVTTVMFLARGIVFMCVEYCPIFFITGNTLQCIM
LVYCFLGYFCTCYFGLFCLLNRYFRLTLGVYDYLVSTQEFRYMNSQGLLPPKNSIDAFKLNIKLLGVGGK
PCIKVATVQSKMSDVKCTSVVLLSVLQQLRVESSSKLWAQCVQLHNDILLAKDTTEAFEKMVSLLSVLLS
MQGAVDINKLCEEMLDNRATLQAIASEFSSLPSYAAFATAQEAYEQAVANGDSEVVLKKLKKSLNVAKSE
FDRDAAMQRKLEKMADQAMTQMYKQARSEDKRAKVTSAMQTMLFTMLRKLDNDALNNIINNARDGCVPLN
IIPLTTAAKLMVVIPDYNTYKNTCDGTTFTYASALWEIQQVVDADSKIVQLSEISMDNSPNLAWPLIVTA
LRANSAVKLQNNELSPVALRQMSCAAGTTQTACTDDNALAYYNTTKGGRFVLALLSDLQDLKWARFPKSD
GTGTIYTELEPPCRFVTDTPKGPKVKYLYFIKGLNNLNRGMVLGSLAATVRLQAGNATEVPANSTVLSFC
AFAVDAAKAYKDYLASGGQPITNCVKMLCTHTGTGQAITVTPEANMDQESFGGASCCLYCRCHIDHPNPK
GFCDLKGKYVQIPTTCANDPVGFTLKNTVCTVCGMWKGYGCSCDQLREPMLQSADAQSFLNGFAV"""
```

Show/Hide Solution

[15]:
```
ORF1a = """MESLVPGFNEKTHVQLSLPVLQVRDVLVRGFGDSVEEVLSEARQHLKDGTCGLVEVEKGVLPQLEQPYVF
IKRSDARTAPHGHVMVELVAELEGIQYGRSGETLGVLVPHVGEIPVAYRKVLLRKNGNKGAGGHSYGADL
KSFDLGDELGTDPYEDFQENWNTKHSSGVTRELMRELNGGAYTRYVDNNFCGPDGYPLECIKDLLARAGK
ASCTLSEQLDFIDTKRGVYCCREHEHEIAWYTERSEKSYELQTPFEIKLAKKFDTFNGECPNFVFPLNSI
IKTIQPRVEKKKLDGFMGRIRSVYPVASPNECNQMCLSTLMKCDHCGETSWQTGDFVKATCEFCGTENLT
KEGATTCGYLPQNAVVKIYCPACHNSEVGPEHSLAEYHNESGLKTILRKGGRTIAFGGCVFSYVGCHNKC
AYWVPRASANIGCNHTGVVGEGSEGLNDNLLEILQKEKVNINIVGDFKLNEEIAIILASFSASTSAFVET
VKGLDYKAFKQIVESCGNFKVTKGKAKKGAWNIGEQKSILSPLYAFASEAARVVRSIFSRTLETAQNSVR
VLQKAAITILDGISQYSLRLIDAMMFTSDLATNNLVVMAYITGGVVQLTSQWLTNIFGTVYEKLKPVLDW
LEEKFKEGVEFLRDGWEIVKFISTCACEIVGGQIVTCAKEIKESVQTFFKLVNKFLALCADSIIIGGAKL
KALNLGETFVTHSKGLYRKCVKSREETGLLMPLKAPKEIIFLEGETLPTEVLTEEVVLKTGDLQPLEQPT
SEAVEAPLVGTPVCINGLMLLEIKDTEKYCALAPNMMVTNNTFTLKGGAPTKVTFGDDTVIEVQGYKSVN
ITFELDERIDKVLNEKCSAYTVELGTEVNEFACVVADAVIKTLQPVSELLTPLGIDLDEWSMATYYLFDE
SGEFKLASHMYCSFYPPDEDEEEGDCEEEEFEPSTQYEYGTEDDYQGKPLEFGATSAALQPEEEQEEDWL
DDDSQQTVGQQDGSEDNQTTTIQTIVEVQPQLEMELTPVVQTIEVNSFSGYLKLTDNVYIKNADIVEEAK
KVKPTVVVNAANVYLKHGGGVAGALNKATNNAMQVESDDYIATNGPLKVGGSCVLSGHNLAKHCLHVVGP
NVNKGEDIQLLKSAYENFNQHEVLLAPLLSAGIFGADPIHSLRVCVDTVRTNVYLAVFDKNLYDKLVSSF
LEMKSEKQVEQKIAEIPKEEVKPFITESKPSVEQRKQDDKKIKACVEEVTTTLEETKFLTENLLLYIDIN
GNLHPDSATLVSDIDITFLKKDAPYIVGDVVQEGVLTAVVIPTKKAGGTTEMLAKALRKVPTDNYITTYP
GQGLNGYTVEEAKTVLKKCKSAFYILPSIISNEKQEILGTVSWNLREMLAHAEETRKLMPVCVETKAIVS
TIQRKYKGIKIQEGVVDYGARFYFYTSKTTVASLINTLNDLNETLVTMPLGYVTHGLNLEEAARYMRSLK
VPATVSVSSPDAVTAYNGYLTSSSKTPEEHFIETISLAGSYKDWSYSGQSTQLGIEFLKRGDKSVYYTSN
PTTFHLDGEVITFDNLKTLLSLREVRTIKVFTTVDNINLHTQVVDMSMTYGQQFGPTYLDGADVTKIKPH
NSHEGKTFYVLPNDDTLRVEAFEYYHTTDPSFLGRYMSALNHTKKWKYPQVNGLTSIKWADNNCYLATAL
LTLQQIELKFNPPALQDAYYRARAGEAANFCALILAYCNKTVGELGDVRETMSYLFQHANLDSCKRVLNV
VCKTCGQQQTTLKGVEAVMYMGTLSYEQFKKGVQIPCTCGKQATKYLVQQESPFVMMSAPPAQYELKHGT
FTCASEYTGNYQCGHYKHITSKETLYCIDGALLTKSSEYKGPITDVFYKENSYTTTIKPVTYKLDGVVCT
EIDPKLDNYYKKDNSYFTEQPIDLVPNQPYPNASFDNFKFVCDNIKFADDLNQLTGYKKPASRELKVTFF
PDLNGDVVAIDYKHYTPSFKKGAKLLHKPIVWHVNNATNKATYKPNTWCIRCLWSTKPVETSNSFDVLKS
EDAQGMDNLACEDLKPVSEEVVENPTIQKDVLECNVKTTEVVGDIILKPANNSLKIIEEVGHTDLMAAYV
DNSSLTIKKPNELSRVLGLKTLATHGLAAVNSVPWDTIANYAKPFLNKVVSTTTNIVTRCLNRVCTNYMP
YFFTLLLQLCTFTRSTNSRIKASMPTTIAKNTVKSVGKFCLEASFNYLKSPNFSKLINIIIWFLLLSVCL
GSLIYSTAALGVLMSNLGMPSYCTGYREGYLNSTNVTIATYCTGSIPCSVCLSGLDSLDTYPSLETIQIT
ISSFKWDLTAFGLVAEWFLAYILFTRFFYVLGLAAIMQLFFSYFAVHFISNSWLMWLIINLVQMAPISAM
VRMYIFFASFYYVWKSYVHVVDGCNSSTCMMCYKRNRATRVECTTIVNGVRRSFYVYANGGKGFCKLHNW
NCVNCDTFCAGSTFISDEVARDLSLQFKRPINPTDQSSYIVDSVTVKNGSIHLYFDKAGQKTYERHSLSH
FVNLDNLRANNTKGSLPINVIVFDGKSKCEESSAKSASVYYSQLMCQPILLLDQALVSDVGDSAEVAVKM
```

(continues on next page)

(continued from previous page)

```
FDAYVNTFSSTFNVPMEKLKTLVATAEAELAKNVSLDNVLSTFISAARQGFVDSDVETKDVVECLKLSHQ
SDIEVTGDSCNNYMLTYNKVENMTPRDLGACIDCSARHINAQVAKSHNIALIWNVKDFMSLSEQLRKQIR
SAAKKNNLPFKLTCATTRQVVNVVTTKIALKGGKIVNNWLKQLIKVTLVFLFVAAIFYLITPVHVMSKHT
DFSSEIIGYKAIDGGVTRDIASTDTCFANKHADFDTWFSQRGGSYTNDKACPLIAAVITREVGFVVPGLP
GTILRTTNGDFLHFLPRVFSAVGNICYTPSKLIEYTDFATSACVLAAECTIFKDASGKPVPYCYDTNVLE
GSVAYESLRPDTRYVLMDGSIIQFPNTYLEGSVRVVTTFDSEYCRHGTCERSEAGVCVSTSGRWVLNNDY
YRSLPGVFCGVDAVNLLTNMFTPLIQPIGALDISASIVAGGIVAIVVTCLAYYFMRFRRAFGEYSHVVAF
NTLLFLMSFTVLCLTPVYSFLPGVYSVIYLYLTFYLTNDVSFLAHIQWMVMFTPLVPFWITIAYIICIST
KHFYWFFSNYLKRRVVFNGVSFSTFEEAALCTFLLNKEMYLKLRSDVLLPLTQYNRYLALYNKYKYFSGA
MDTTSYREAACCHLAKALNDFSNSGSDVLYQPPQTSITSAVLQSGFRKMAFPSGKVEGCMVQVTCGTTTL
NGLWLDDVVYCPRHVICTSEDMLNPNYEDLLIRKSNHNFLVQAGNVQLRVIGHSMQNCVLKLKVDTANPK
TPKYKFVRIQPGQTFSVLACYNGSPSGVYQCAMRPNFTIKGSFLNGSCGSVGFNIDYDCVSFCYMHHMEL
PTGVHAGTDLEGNFYGPFVDRQTAQAAGTDTTITVNVLAWLYAAVINGDRWFLNRFTTTLNDFNLVAMKY
NYEPLTQDHVDILGPLSAQTGIAVLDMCASLKELLQNGMNGRTILGSALLEDEFTPFDVVRQCSGVTFQS
AVKRTIKGTHHWLLLTILTSLLVLVQSTQWSLFFFLYENAFLPFAMGIIAMSAFAMMFVKHKHAFLCLFL
LPSLATVAYFNMVYMPASWVMRIMTWLDMVDTSLSGFKLKDCVMYASAVVLLILMTARTVYDDGARRVWT
LMNVLTLVYKVYYGNALDQAISMWALIISVTSNYSGVVTTVMFLARGIVFMCVEYCPIFFITGNTLQCIM
LVYCFLGYFCTCYFGLFCLLNRYFRLTLGVYDYLVSTQEFRYMNSQGLLPPKNSIDAFKLNIKLLGVGGK
PCIKVATVQSKMSDVKCTSVVLLSVLQQLRVESSSKLWAQCVQLHNDILLAKDTTEAFEKMVSLLSVLLS
MQGAVDINKLCEEMLDNRATLQAIASEFSSLPSYAAFATAQEAYEQAVANGDSEVVLKKLKKSLNVAKSE
FDRDAAMQRKLEKMADQAMTQMYKQARSEDKRAKVTSAMQTMLFTMLRKLDNDALNNIINNARDGCVPLN
IIPLTTAAKLMVVIPDYNTYKNTCDGTTFTYASALWEIQQVVDADSKIVQLSEISMDNSPNLAWPLIVTA
LRANSAVKLQNNELSPVALRQMSCAAGTTQTACTDDNALAYYNTTKGGRFVLALLSDLQDLKWARFPKSD
GTGTIYTELEPPCRFVTDTPKGPKVKYLYFIKGLNNLNRGMVLGSLAATVRLQAGNATEVPANSTVLSFC
AFAVDAAKAYKDYLASGGQPITNCVKMLCTHTGTGQAITVTPEANMDQESFGGASCCLYCRCHIDHPNPK
GFCDLKGKYVQIPTTCANDPVGFTLKNTVCTVCGMWKGYGCSCDQLREPMLQSADAQSFLNGFAV"""


ORF1a = ORF1a.replace("\n","")
print("ORF1a protein has ", len(ORF1a), " aminoacids")

seqs = {"TTTL" : [], "GFAV": [], "KMLL": []}

for i in range(len(ORF1a)-3):
    fourmer = ORF1a[i:i+4]
    if fourmer in seqs:  #remember it tests the presence in the keys
        seqs[fourmer].append(i)

for s in seqs:
    print(s, " is present at positions: ", seqs[s])
```

```
ORF1a protein has  4405  aminoacids
TTTL  is present at positions:  [1239, 3286, 3486]
GFAV  is present at positions:  [4401]
KMLL  is present at positions:  []
```

3. Given the string "nOBody Said iT was eAsy, No oNe Ever saId it WoulD be tHis hArd…"
    1. Create a list with all the letters that are capitalized (use str.isupper)
    2. Print the list
    3. Use the string method **join** to concatenate all the letters in a string, using "*" as separator. The syntax of join is str.join(list) and it outputs a string with all the elements in list joined with the character in str (es. "+".join([1,2,3]) returns "1+2+3").

The expected output:

```
['O', 'B', 'S', 'T', 'A', 'N', 'N', 'E', 'I', 'W', 'D', 'H', 'A']
O*B*S*T*A*N*N*E*I*W*D*H*A
```

Show/Hide Solution

[24]:
```
text = "nOBody Said iT was eAsy, No oNe Ever saId it WoulD be tHis hArd…"
initials = [x for x in text if x.isupper()]

print(initials)
print("*".join(initials))
```

```
['O', 'B', 'S', 'T', 'A', 'N', 'N', 'E', 'I', 'W', 'D', 'H', 'A']
O*B*S*T*A*N*N*E*I*W*D*H*A
```

4. Given the following sequence:

```
AUGCUGUCUCCCUCACUGUAUGUAAAUUGCAUCUAGAAUAGCA
UCUGGAGCACUAAUUGACACAUAGUGGGUAUCAAUUAUUA
UUCCAGGUACUAGAGAUACCUGGACCAUUAACGGAUAAAU
AGAAGAUUCAUUUGUUGAGUGACUGAGGAUGGCAGUUCCU
GCUACCUUCAAGGAUCUGGAUGAUGGGGAGAAACAGAGAA
CAUAGUGUGAGAAUACUGUGGUAAGGAAAGUACAGAGGAC
UGGUAGAGUGUCUAACCUAGAUUUGGAGAAGGACCUAGAA
GUCUAUCCCAGGGAAAUAAAAAUCUAAGCUAAGGUUUGAG
GAAUCAGUAGGAAUUGGCAAAGGAAGGACAUGUUCCAGAU
GAUAGGAACAGGUUAUGCAAAGAUCCUGAAAUGGUCAGAG
CUUGGUGCUUUUUGAGAACCAAAAGUAGAUUGUUAUGGAC
CAGUGCUACUCCCUGCCUCUUGCCAAGGGACCCCGCCAAG
CACUGCAUCCCUUCCCUCUGACUCCACCUUUCCACUUGCC
CAGUAUUGUUGGUG
```

considering only first forward open reading frame (i.e. the string as it is **remembering to remove newlines**): 1. create a list of all the codons and print it (considering only first forward open reading frame we will have the codons: AUG, CUG, UCU,…).

2. create then a dictionary in which codons are keys and values are how many times that codon is present in the first forward open reading frame and print it; 3. retrieve from the dictionary and print the number of times codons: "AUG", "AGA", "GGG", "AUA" are present. If a codon is not present the script should print "NOT present" rather than crash.

Show/Hide Solution

[25]:
```
seq= """AUGCUGUCUCCCUCACUGUAUGUAAAUUGCAUCUAGAAUAGCA
UCUGGAGCACUAAUUGACACAUAGUGGGUAUCAAUUAUUA
UUCCAGGUACUAGAGAUACCUGGACCAUUAACGGAUAAAU
AGAAGAUUCAUUUGUUGAGUGACUGAGGAUGGCAGUUCCU
GCUACCUUCAAGGAUCUGGAUGAUGGGGAGAAACAGAGAA
CAUAGUGUGAGAAUACUGUGGUAAGGAAAGUACAGAGGAC
UGGUAGAGUGUCUAACCUAGAUUUGGAGAAGGACCUAGAA
GUCUAUCCCAGGGAAAUAAAAAUCUAAGCUAAGGUUUGAG
GAAUCAGUAGGAAUUGGCAAAGGAAGGACAUGUUCCAGAU
GAUAGGAACAGGUUAUGCAAAGAUCCUGAAAUGGUCAGAG
CUUGGUGCUUUUUGAGAACCAAAAGUAGAUUGUUAUGGAC
CAGUGCUACUCCCUGCCUCUUGCCAAGGGACCCCGCCAAG
CACUGCAUCCCUUCCCUCUGACUCCACCUUUCCACUUGCC
CAGUAUUGUUGGUG"""
```

(continues on next page)

(continued from previous page)

```
seq = seq.replace("\n","")

codons = [seq[i:i+3] for i in range(0,len(seq),3)]

count_dict = dict()

for el in codons:
    if el not in count_dict:
        count_dict[el] = codons.count(el)

print("1st fwd ORF codons:")
print(codons)
print("Codon counts:")
print(count_dict)
print("\n")
to_check = ["AUG", "AGA", "GGG", "AUA"]
for c in to_check:
    if c in count_dict:
        print(c , "is present", count_dict[c], "times")
    else:
        print(c, "is NOT present")
```

```
1st fwd ORF codons:
['AUG', 'CUG', 'UCU', 'CCC', 'UCA', 'CUG', 'UAU', 'GUA', 'AAU', 'UGC', 'AUC', 'UAG',
↪'AAU', 'AGC', 'AUC', 'UGG', 'AGC', 'ACU', 'AAU', 'UGA', 'CAC', 'AUA', 'GUG', 'GGU',
↪'AUC', 'AAU', 'UAU', 'UAU', 'UCC', 'AGG', 'UAC', 'UAG', 'AGA', 'UAC', 'CUG', 'GAC',
↪'CAU', 'UAA', 'CGG', 'AUA', 'AAU', 'AGA', 'AGA', 'UUC', 'AUU', 'UGU', 'UGA', 'GUG',
↪'ACU', 'GAG', 'GAU', 'GGC', 'AGU', 'UCC', 'UGC', 'UAC', 'CUU', 'CAA', 'GGA', 'UCU',
↪'GGA', 'UGA', 'UGG', 'GGA', 'GAA', 'ACA', 'GAG', 'AAC', 'AUA', 'GUG', 'UGA', 'GAA',
↪'UAC', 'UGU', 'GGU', 'AAG', 'GAA', 'AGU', 'ACA', 'GAG', 'GAC', 'UGG', 'UAG', 'AGU',
↪'GUC', 'UAA', 'CCU', 'AGA', 'UUU', 'GGA', 'GAA', 'GGA', 'CCU', 'AGA', 'AGU', 'CUA',
↪'UCC', 'CAG', 'GGA', 'AAU', 'AAA', 'AAU', 'CUA', 'AGC', 'UAA', 'GGU', 'UUG', 'AGG',
↪'AAU', 'CAG', 'UAG', 'GAA', 'UUG', 'GCA', 'AAG', 'GAA', 'GGA', 'CAU', 'GUU', 'CCA',
↪'GAU', 'GAU', 'AGG', 'AAC', 'AGG', 'UUA', 'UGC', 'AAA', 'GAU', 'CCU', 'GAA', 'AUG',
↪'GUC', 'AGA', 'GCU', 'UGG', 'UGC', 'UUU', 'UUG', 'AGA', 'ACC', 'AAA', 'AGU', 'AGA',
↪'UUG', 'UUA', 'UGG', 'ACC', 'AGU', 'GCU', 'ACU', 'CCC', 'UGC', 'CUC', 'UUG', 'CCA',
↪'AGG', 'GAC', 'CCC', 'GCC', 'AAG', 'CAC', 'UGC', 'AUC', 'CCU', 'UCC', 'CUC', 'UGA',
↪'CUC', 'CAC', 'CUU', 'UCC', 'ACU', 'UGC', 'CCA', 'GUA', 'UUG', 'UUG', 'GUG']
Codon counts:
{'AUG': 2, 'CUG': 3, 'UCU': 2, 'CCC': 3, 'UCA': 1, 'UAU': 3, 'GUA': 2, 'AAU': 8, 'UGC
↪': 7, 'AUC': 4, 'UAG': 4, 'AGC': 3, 'UGG': 5, 'ACU': 4, 'UGA': 5, 'CAC': 3, 'AUA':␣
↪3, 'GUG': 4, 'GGU': 3, 'UCC': 5, 'AGG': 5, 'UAC': 4, 'AGA': 8, 'GAC': 3, 'CAU': 2,
↪'UAA': 3, 'CGG': 1, 'UUC': 1, 'AUU': 1, 'UGU': 2, 'GAG': 3, 'GAU': 4, 'GGC': 1, 'AGU
↪': 6, 'CUU': 2, 'CAA': 1, 'GGA': 7, 'GAA': 7, 'ACA': 2, 'AAC': 2, 'AAG': 3, 'GUC':␣
↪2, 'CCU': 4, 'UUU': 2, 'CUA': 2, 'CAG': 2, 'AAA': 3, 'UUG': 7, 'GCA': 1, 'GUU': 1,
↪'CCA': 3, 'UUA': 2, 'GCU': 2, 'ACC': 2, 'CUC': 3, 'GCC': 1}


AUG is present 2 times
AGA is present 8 times
GGG is NOT present
AUA is present 3 times
```

5. Given the following list of gene correlations:

```
geneCorr = [["G1C2W9", "G1C2Q7", 0.2], ["G1C2W9", "G1C2Q4", 0.9],
["Q6NMS1", "G1C2W9", 0.8],["G1C2W9", "Q6NMS1",0.4], ["G1C2Q7", "G1C2Q4",0.76]]
```

where each sublist ["gene1", "gene2", corr] represents a correlation between *gene1* and *gene2* with correlation *corr*, create another list containing only the elements having an high correlation (i.e. > 0.75). Print this list.

Expected result:

```
[['G1C2W9', 'G1C2Q4', 0.9], ['Q6NMS1', 'G1C2W9', 0.8], ['G1C2Q7', 'G1C2Q4', 0.76]]
```

Show/Hide Solution

[26]:
```
geneCorr = [["G1C2W9", "G1C2Q7", 0.2], ["G1C2W9", "G1C2Q4", 0.9], ["Q6NMS1", "G1C2W9",
↪ 0.8],
            ["G1C2W9", "Q6NMS1",0.4], ["G1C2Q7", "G1C2Q4",0.76]]

highlyCorr = [x for x in geneCorr if x[2] > 0.75]

print(geneCorr, "\n")
print(highlyCorr)
```

```
[['G1C2W9', 'G1C2Q7', 0.2], ['G1C2W9', 'G1C2Q4', 0.9], ['Q6NMS1', 'G1C2W9', 0.8], [
↪'G1C2W9', 'Q6NMS1', 0.4], ['G1C2Q7', 'G1C2Q4', 0.76]]

[['G1C2W9', 'G1C2Q4', 0.9], ['Q6NMS1', 'G1C2W9', 0.8], ['G1C2Q7', 'G1C2Q4', 0.76]]
```

6. Given the following sequence of DNA:

DNA = "GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC"

```
1. Create a dictionary reporting the frequency of each base (i.e. key is the
base and value is the frequency).
2. Create a dictionary representing an index of all possible dimers (i.e. 2
bases, 16 dimers in total): AA, AT, AC, AG, TA, TT, TC, TG, ... . In this case,
keys of the dictionary are dimers and values are lists with all possible starting
positions of the dimer.
3. Print the DNA string.
4. Print for each base its frequency
4. Print all positions of the dimer "AT"
```

The expected result is:

```
sequence: GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC
G has frequency: 0.1509433962264151
C has frequency: 0.22641509433962265
A has frequency: 0.3584905660377358
T has frequency: 0.2641509433962264
{'GG': [32, 33], 'TC': [11], 'GT': [14], 'CA': [5, 12, 17], 'TT': [2, 36, 42],
'CG': [27, 29, 31], 'TA': [3, 7, 9, 15, 21, 23, 25, 37, 40, 43], 'AG': [13, 18],
'GA': [0, 19], 'CT': [35, 39], 'GC': [28, 30, 34], 'AT': [1, 6, 8, 10, 20, 22, 24,␣
↪41],
'CC': [49, 50, 51], 'AA': [44, 45, 46, 47], 'AC': [4, 16, 26, 38, 48]}

Dimer AT is found at: [1, 6, 8, 10, 20, 22, 24, 41]
```

Show/Hide Solution

[27]:
```
DNA = "GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC"

n = len(DNA)

baseFreq = {"A" : DNA.count("A")/n, "T" : DNA.count("T")/n,
            "C": DNA.count("C")/n, "G" : DNA.count("G")/n }

dimersDict ={}

print("sequence:", DNA)

for base in baseFreq:
    print(base, "has frequency:", baseFreq[base])

for ind in range(len(DNA) -1 ): #need -1 because at each iteration I get the dimer␣
↪[ind:ind+1]
    dimer = DNA[ind:ind+2]
    if dimer in dimersDict:
        dimersDict[dimer].append(ind)
    else:
        dimersDict[dimer] = [ind]
print(dimersDict, "\n")
print("Dimer AT is found at:", dimersDict["AT"])
```

```
sequence: GATTACATATATCAGTACAGATATATACGCGCGGGCTTACTATTAAAAACCCC
A has frequency: 0.3584905660377358
T has frequency: 0.2641509433962264
C has frequency: 0.22641509433962265
G has frequency: 0.1509433962264151
{'GA': [0, 19], 'AT': [1, 6, 8, 10, 20, 22, 24, 41], 'TT': [2, 36, 42], 'TA': [3, 7,␣
↪9, 15, 21, 23, 25, 37, 40, 43], 'AC': [4, 16, 26, 38, 48], 'CA': [5, 12, 17], 'TC':␣
↪[11], 'AG': [13, 18], 'GT': [14], 'CG': [27, 29, 31], 'GC': [28, 30, 34], 'GG': [32,
↪ 33], 'CT': [35, 39], 'AA': [44, 45, 46, 47], 'CC': [49, 50, 51]}

Dimer AT is found at: [1, 6, 8, 10, 20, 22, 24, 41]
```

7. Given the following table, reporting molecular weights for each amino acid, store them in a dictionary where the key is the one letter code and the value is the molecular weight (e.g. {“A” : 89, “R”:174”}).

Amino Acid	Three-Letter Abbreviation	One-Letter Symbol	Molecular Weight
Alanine	Ala	A	89Da
Arginine	Arg	R	174Da
Asparagine	Asn	N	132Da
Aspartic acid	Asp	D	133Da
Asparagine or aspartic acid	Asx	B	133Da
Cysteine	Cys	C	121Da
Glutamine	Gln	Q	146Da
Glutamic acid	Glu	E	147Da
Glutamine or glutamic acid	Glx	Z	147Da
Glycine	Gly	G	75Da
Histidine	His	H	155Da
Isoleucine	Ile	I	131Da
Leucine	Leu	L	131Da
Lysine	Lys	K	146Da
Methionine	Met	M	149Da
Phenylalanine	Phe	F	165Da
Proline	Pro	P	115Da
Serine	Ser	S	105Da
Threonine	Thr	T	119Da
Tryptophan	Trp	W	204Da
Tyrosine	Tyr	Y	181Da
Valine	Val	V	117Da

Write a python script to answer the following questions:

```
1. What is the average molecular weight of an amino acid?
2. What is the total molecular weight and number of aminoacids
of the P53 peptide GSRAHSSHLKSKKGQSTSRHK?
3. What is the total molecular weight and number of aminoacids
of the peptide YTSLIHSLIEESQNQQEKNEQELLELDKWASLWNWF?
```

Show/Hide Solution

[28]:
```
mws = {"A" : 89, "R" : 174, "N" : 132, "D" : 133, "B": 133, "C": 121, "Q": 146,
       "E": 147, "Z": 147, "G": 75, "H": 155, "I": 131, "L" : 131,
      "K" : 146, "M" : 149, "F" : 165, "P" : 115, "S" : 105, "T" : 119,
       "W" : 204, "Y": 181, "V" : 117
      }
avgW = 0
for amino in mws:
    avgW = avgW + mws[amino]

avgW = avgW/len(mws)
print("The average molecular weight of amino acids is:", avgW)

P53amino = "GSRAHSSHLKSKKGQSTSRHK"
```

(continues on next page)

(continued from previous page)

```
totW = 0
for amino in P53amino:
    totW += mws[amino]
print("Peptide ", P53amino, "has",len(P53amino), "amino acids and a total mw of",totW,
↪ "Da")

totW = 0 #reusing same variable name
peptide = "YTSLIHSLIEESQNQQEKNEQELLELDKWASLWNWF"
for amino in peptide:
    totW += mws[amino]

print("Peptide ", peptide, "has",len(peptide), "amino acids and a total mw of",totW,
↪"Da")
```

```
The average molecular weight of amino acids is: 137.04545454545453
Peptide  GSRAHSSHLKSKKGQSTSRHK has 21 amino acids and a total mw of 2662 Da
Peptide  YTSLIHSLIEESQNQQEKNEQELLELDKWASLWNWF has 36 amino acids and a total mw of␣
↪5076 Da
```

8. The following string is an extract of a blast[39] alignment with compacted textual output. This is a comma (",") separated text file where the columns report the following info: the first column is the query_id, the second the subject_id (i.e. the reference on which we aligned the query), the third is the percentage of identity and then we have the alignment length, number of mismatches, gap opens, start point of the alignment on the query, end point of the alignment on the query, start point of the alignment on the subject, end point of the alignment on the subject and evalue of the alignment.

```
#Fields:q.id,s.id,% ident,align len,mismatches,gap opens,q.start,q.end,s.start,s.end,
↪evalue
ab1_400a,scaffold16155,98.698,384,4,1,12,394,6700,7083,0.0
ab1_400b,scaffold14620,98.698,384,4,1,12,394,1240,857,0.0
92A2_SP6_344a,scaffold14394,95.575,113,5,0,97,209,250760,250648,2.92e-44
92A2_SP6_344b,scaffold10682,97.849,93,2,0,18,110,898,990,3.81e-38
92A2_T7_558a,scaffold277,88.746,311,31,3,21,330,26630,26937,5.81e-103
92A2_T7_558b,scaffold277,89.545,220,21,2,27,246,27167,26950,6.06e-73
92A2_T7_558c,scaffold1125,88.125,320,31,5,30,346,231532,231847,7.51e-102
ab1_675a,scaffold4896,100.000,661,0,0,15,675,79051,78391,0.0
ab1_676b,scaffold4896,99.552,670,0,3,7,673,78421,79090,0.0
```

1. For each alignment, store the subject id, the percentage of identity and evalue, subject start and end in a dictionary using the query id as key. All this information can be stored in a dictionary having subject id as key and a dictionary with all the information as value:

   ```
   alignments["ab1_400"] = {"subjectid" : scaffold16155, "perc_id" : 98.698, "evalue
   ↪" : 0.0}
   ```

2. Print the whole dictionary
3. Print only the alignments having percentage of identity > 90%

Note: skip the first comment line (i.e. skip line if starts with "#"). Note1: when storing the percentage of identity remember to convert the string into a float.

The expected output is:

[39] https://www.ncbi.nlm.nih.gov/pubmed/2231712

```
{'92A2_T7_558a': {'perc_id': 88.746, 'subjectid': 'scaffold277', 'evalue': '5.81e-103
↪'},
'92A2_T7_558b': {'perc_id': 89.545, 'subjectid': 'scaffold277', 'evalue': '6.06e-73'},
'92A2_SP6_344b': {'perc_id': 97.849, 'subjectid': 'scaffold10682', 'evalue': '3.81e-38
↪'},
'92A2_T7_558c': {'perc_id': 88.125, 'subjectid': 'scaffold1125', 'evalue': '7.51e-102
↪'},
'ab1_400a': {'perc_id': 98.698, 'subjectid': 'scaffold16155', 'evalue': '0.0'},
'ab1_675a': {'perc_id': 100.0, 'subjectid': 'scaffold4896', 'evalue': '0.0'},
'ab1_400b': {'perc_id': 98.698, 'subjectid': 'scaffold14620', 'evalue': '0.0'},
'92A2_SP6_344a': {'perc_id': 95.575, 'subjectid': 'scaffold14394', 'evalue': '2.92e-44
↪'},
'ab1_676b': {'perc_id': 99.552, 'subjectid': 'scaffold4896', 'evalue': '0.0'}}


Alignments with identity > 90%:

Query id    Subject id  % ident evalue
92A2_SP6_344b    scaffold10682   97.849      3.81e-38
ab1_400a     scaffold16155   98.698      0.0
ab1_675a     scaffold4896    100.0   0.0
ab1_400b     scaffold14620   98.698      0.0
92A2_SP6_344a    scaffold14394   95.575      2.92e-44
ab1_676b     scaffold4896    99.552      0.0
```

Show/Hide Solution

[29]:
```
blast_out = """#Fields:q.id,s.id,% ident,align len,mismatches,gap opens,q.start,q.end,
↪s.start,s.end,evalue
ab1_400a,scaffold16155,98.698,384,4,1,12,394,6700,7083,0.0
ab1_400b,scaffold14620,98.698,384,4,1,12,394,1240,857,0.0
92A2_SP6_344a,scaffold14394,95.575,113,5,0,97,209,250760,250648,2.92e-44
92A2_SP6_344b,scaffold10682,97.849,93,2,0,18,110,898,990,3.81e-38
92A2_T7_558a,scaffold277,88.746,311,31,3,21,330,26630,26937,5.81e-103
92A2_T7_558b,scaffold277,89.545,220,21,2,27,246,27167,26950,6.06e-73
92A2_T7_558c,scaffold1125,88.125,320,31,5,30,346,231532,231847,7.51e-102
ab1_675a,scaffold4896,100.000,661,0,0,15,675,79051,78391,0.0
ab1_676b,scaffold4896,99.552,670,0,3,7,673,78421,79090,0.0"""



alignments = dict()

for align in blast_out.split("\n"):
    if align.startswith("#") == False:
        align_info = align.split(",")
        alignments[align_info[0]] = {"subjectid" : align_info[1],
                                     "perc_id" : float(align_info[2]),
                                    "evalue" : align_info[-1]
                                    }

print(alignments, "\n\n")
print("Alignments with identity > 90%:\n")
print("Query id\tSubject id\t% ident\tevalue")
for a in alignments:
    if alignments[a]["perc_id"]>90:
        print(a,"\t", alignments[a]["subjectid"],"\t", alignments[a]["perc_id"],"\t",␣
↪alignments[a]["evalue"])
```

(continues on next page)

(continued from previous page)

```
{'ab1_400a': {'subjectid': 'scaffold16155', 'perc_id': 98.698, 'evalue': '0.0'}, 'ab1_
↪400b': {'subjectid': 'scaffold14620', 'perc_id': 98.698, 'evalue': '0.0'}, '92A2_
↪SP6_344a': {'subjectid': 'scaffold14394', 'perc_id': 95.575, 'evalue': '2.92e-44'},
↪'92A2_SP6_344b': {'subjectid': 'scaffold10682', 'perc_id': 97.849, 'evalue': '3.81e-
↪38'}, '92A2_T7_558a': {'subjectid': 'scaffold277', 'perc_id': 88.746, 'evalue': '5.
↪81e-103'}, '92A2_T7_558b': {'subjectid': 'scaffold277', 'perc_id': 89.545, 'evalue':
↪ '6.06e-73'}, '92A2_T7_558c': {'subjectid': 'scaffold1125', 'perc_id': 88.125,
↪'evalue': '7.51e-102'}, 'ab1_675a': {'subjectid': 'scaffold4896', 'perc_id': 100.0,
↪'evalue': '0.0'}, 'ab1_676b': {'subjectid': 'scaffold4896', 'perc_id': 99.552,
↪'evalue': '0.0'}}


Alignments with identity > 90%:

Query id        Subject id      % ident        evalue
ab1_400a        scaffold16155   98.698          0.0
ab1_400b        scaffold14620   98.698          0.0
92A2_SP6_344a          scaffold14394   95.575          2.92e-44
92A2_SP6_344b          scaffold10682   97.849          3.81e-38
ab1_675a        scaffold4896    100.0   0.0
ab1_676b        scaffold4896    99.552          0.0
```

9. The following text (separated with a comma ",") is an extract of protein-protein interactions network stored in the database STRING[40] involving PKLR[41] (Pyruvate kinase, liver and RBC) that plays a key role in glycolysis:

```
#node1,node2,node1_ext_id,node2_ext_id,
ENO1,TPI1,ENSP00000234590,ENSP00000229270
PKLR,ENO1,ENSP00000339933,ENSP00000234590
PKLR,ENO3,ENSP00000339933,ENSP00000324105
PGK1,ENO1,ENSP00000362413,ENSP00000234590
PGK1,TPI1,ENSP00000362413,ENSP00000229270
GPI,TPI1,ENSP00000405573,ENSP00000229270
PKLR,ENO2,ENSP00000339933,ENSP00000229277
PGK1,ENO3,ENSP00000362413,ENSP00000324105
PGK1,ENO2,ENSP00000362413,ENSP00000229277
GPI,PKLR,ENSP00000405573,ENSP00000339933
ENO2,TPI1,ENSP00000229277,ENSP00000229270
PGK2,ENO1,ENSP00000305995,ENSP00000234590
ENO3,PGK2,ENSP00000324105,ENSP00000305995
PGK2,TPI1,ENSP00000305995,ENSP00000229270
ENO3,TPI1,ENSP00000324105,ENSP00000229270
PGK2,ENO2,ENSP00000305995,ENSP00000229277
GPI,ENO3,ENSP00000405573,ENSP00000324105
PKLR,LDHB,ENSP00000339933,ENSP00000229319
PKLR,LDHC,ENSP00000339933,ENSP00000280704
PKLR,TPI1,ENSP00000339933,ENSP00000229270
PGK1,PKLR,ENSP00000362413,ENSP00000339933
GPI,ENO2,ENSP00000405573,ENSP00000229277
PKLR,PGK2,ENSP00000339933,ENSP00000305995
GPI,PGK1,ENSP00000405573,ENSP00000362413
ME3,PKLR,ENSP00000352657,ENSP00000339933
ME3,LDHB,ENSP00000352657,ENSP00000229319
ME3,LDHC,ENSP00000352657,ENSP00000280704
```

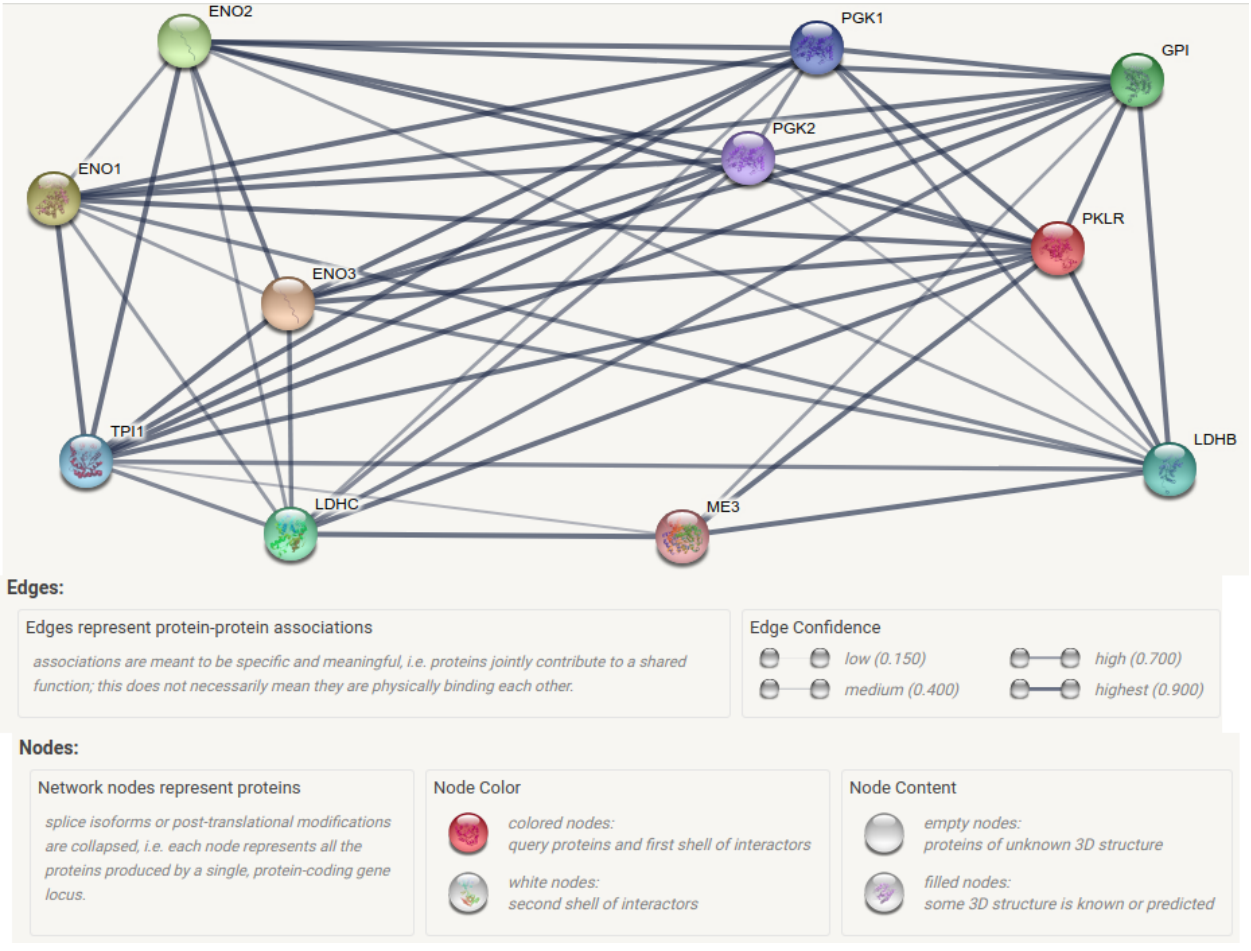
(continues on next page)

[40] https://string-db.org/
[41] https://www.ncbi.nlm.nih.gov/gene/5313

(continued from previous page)

```
GPI,PGK2,ENSP00000405573,ENSP00000305995
GPI,ENO1,ENSP00000405573,ENSP00000234590
GPI,LDHB,ENSP00000405573,ENSP00000229319
ENO3,ENO2,ENSP00000324105,ENSP00000229277
GPI,LDHC,ENSP00000405573,ENSP00000280704
ENO3,LDHB,ENSP00000324105,ENSP00000229319
ENO3,LDHC,ENSP00000324105,ENSP00000280704
ENO1,LDHB,ENSP00000234590,ENSP00000229319
LDHB,TPI1,ENSP00000229319,ENSP00000229270
LDHC,TPI1,ENSP00000280704,ENSP00000229270
PGK2,LDHC,ENSP00000305995,ENSP00000280704
PGK1,LDHB,ENSP00000362413,ENSP00000229319
PGK1,PGK2,ENSP00000362413,ENSP00000305995
ENO1,ENO2,ENSP00000234590,ENSP00000229277
LDHC,ENO1,ENSP00000280704,ENSP00000234590
LDHB,ENO2,ENSP00000229319,ENSP00000229277
LDHC,ENO2,ENSP00000280704,ENSP00000229277
ENO3,ENO1,ENSP00000324105,ENSP00000234590
PGK1,LDHC,ENSP00000362413,ENSP00000280704
GPI,ME3,ENSP00000405573,ENSP00000352657
PGK2,LDHB,ENSP00000305995,ENSP00000229319
ME3,TPI1,ENSP00000352657,ENSP00000229270
```

Here is a graphic representation of the protein-protein interactions:



```
Note: we can assume that relations between nodes are transitive. node1 --> node2
implies node2 --> node1.

1. Store the network information in a dictionary having node1 as key and
the list of all nodes2 associated to it as value (remember to skip the first
line that is the header). Remember transitivity, therefore add also node2 -->node1
2. Find all first neighbours of "PKLR" (i.e. the nodes that are directly connected
to "PKLR") and print them
3. Find all first neighbours of "ME3" (i.e. the nodes that are directly connected
to "PKLR") and print them
4. Find all the second neighbours of "ME3" (i.e. the nodes that are connected to nodes
directly connected to "ME3", but not directly to "ME3").
```

Show/Hide Solution

[30]:
```
networkStr = """#node1,node2,node1_ext_id,node2_ext_id,
ENO1,TPI1,ENSP00000234590,ENSP00000229270
PKLR,ENO1,ENSP00000339933,ENSP00000234590
PKLR,ENO3,ENSP00000339933,ENSP00000324105
PGK1,ENO1,ENSP00000362413,ENSP00000234590
PGK1,TPI1,ENSP00000362413,ENSP00000229270
GPI,TPI1,ENSP00000405573,ENSP00000229270
PKLR,ENO2,ENSP00000339933,ENSP00000229277
PGK1,ENO3,ENSP00000362413,ENSP00000324105
PGK1,ENO2,ENSP00000362413,ENSP00000229277
GPI,PKLR,ENSP00000405573,ENSP00000339933
ENO2,TPI1,ENSP00000229277,ENSP00000229270
PGK2,ENO1,ENSP00000305995,ENSP00000234590
ENO3,PGK2,ENSP00000324105,ENSP00000305995
PGK2,TPI1,ENSP00000305995,ENSP00000229270
ENO3,TPI1,ENSP00000324105,ENSP00000229270
PGK2,ENO2,ENSP00000305995,ENSP00000229277
GPI,ENO3,ENSP00000405573,ENSP00000324105
PKLR,LDHB,ENSP00000339933,ENSP00000229319
PKLR,LDHC,ENSP00000339933,ENSP00000280704
PKLR,TPI1,ENSP00000339933,ENSP00000229270
PGK1,PKLR,ENSP00000362413,ENSP00000339933
GPI,ENO2,ENSP00000405573,ENSP00000229277
PKLR,PGK2,ENSP00000339933,ENSP00000305995
GPI,PGK1,ENSP00000405573,ENSP00000362413
ME3,PKLR,ENSP00000352657,ENSP00000339933
ME3,LDHB,ENSP00000352657,ENSP00000229319
ME3,LDHC,ENSP00000352657,ENSP00000280704
GPI,PGK2,ENSP00000405573,ENSP00000305995
GPI,ENO1,ENSP00000405573,ENSP00000234590
GPI,LDHB,ENSP00000405573,ENSP00000229319
ENO3,ENO2,ENSP00000324105,ENSP00000229277
GPI,LDHC,ENSP00000405573,ENSP00000280704
ENO3,LDHB,ENSP00000324105,ENSP00000229319
ENO3,LDHC,ENSP00000324105,ENSP00000280704
ENO1,LDHB,ENSP00000234590,ENSP00000229319
LDHB,TPI1,ENSP00000229319,ENSP00000229270
LDHC,TPI1,ENSP00000280704,ENSP00000229270
PGK2,LDHC,ENSP00000305995,ENSP00000280704
PGK1,LDHB,ENSP00000362413,ENSP00000229319
PGK1,PGK2,ENSP00000362413,ENSP00000305995
ENO1,ENO2,ENSP00000234590,ENSP00000229277
```

(continues on next page)

(continued from previous page)

```
LDHC,ENO1,ENSP00000280704,ENSP00000234590
LDHB,ENO2,ENSP00000229319,ENSP00000229277
LDHC,ENO2,ENSP00000280704,ENSP00000229277
ENO3,ENO1,ENSP00000324105,ENSP00000234590
PGK1,LDHC,ENSP00000362413,ENSP00000280704
GPI,ME3,ENSP00000405573,ENSP00000352657
PGK2,LDHB,ENSP00000305995,ENSP00000229319
ME3,TPI1,ENSP00000352657,ENSP00000229270"""

netData = dict()

for line in networkStr.split("\n"):

    if not line.startswith("#"):
        info = line.split(",")
        #insert node1
        if info[0] in netData:
            netData[info[0]].append(info[1])
        else:
            netData[info[0]] = [info[1]]
        #insert node2
        if info[1] in netData:
            netData[info[1]].append(info[0])
        else:
            netData[info[1]] = [info[0]]


print(netData)

print("\nPKLR is directly connected to:", netData["PKLR"] )

firstNeigh = netData["ME3"]
print("\nME3 is directly connected to:", firstNeigh  )

secondNeigh = []
for node in firstNeigh:
    if node in netData: #important, the node might not have connections
        neighbours = netData[node]
        for n in neighbours:
            if n not in secondNeigh and  n not in firstNeigh and n != "ME3":
                secondNeigh.append(n)

print("\nME3's second neighbours:", secondNeigh  )
```

```
{'ENO1': ['TPI1', 'PKLR', 'PGK1', 'PGK2', 'GPI', 'LDHB', 'ENO2', 'LDHC', 'ENO3'],
↪'TPI1': ['ENO1', 'PGK1', 'GPI', 'ENO2', 'PGK2', 'ENO3', 'PKLR', 'LDHB', 'LDHC', 'ME3
↪'], 'PKLR': ['ENO1', 'ENO3', 'ENO2', 'GPI', 'LDHB', 'LDHC', 'TPI1', 'PGK1', 'PGK2',
↪'ME3'], 'ENO3': ['PKLR', 'PGK1', 'PGK2', 'TPI1', 'GPI', 'ENO2', 'LDHB', 'LDHC',
↪'ENO1'], 'PGK1': ['ENO1', 'TPI1', 'ENO3', 'ENO2', 'PKLR', 'GPI', 'LDHB', 'PGK2',
↪'LDHC'], 'GPI': ['TPI1', 'PKLR', 'ENO3', 'ENO2', 'PGK1', 'PGK2', 'ENO1', 'LDHB',
↪'LDHC', 'ME3'], 'ENO2': ['PKLR', 'PGK1', 'TPI1', 'PGK2', 'GPI', 'ENO3', 'ENO1',
↪'LDHB', 'LDHC'], 'PGK2': ['ENO1', 'ENO3', 'TPI1', 'ENO2', 'PKLR', 'GPI', 'LDHC',
↪'PGK1', 'LDHB'], 'LDHB': ['PKLR', 'ME3', 'GPI', 'ENO3', 'ENO1', 'TPI1', 'PGK1',
↪'ENO2', 'PGK2'], 'LDHC': ['PKLR', 'ME3', 'GPI', 'ENO3', 'TPI1', 'PGK2', 'ENO1',
↪'ENO2', 'PGK1'], 'ME3': ['PKLR', 'LDHB', 'LDHC', 'GPI', 'TPI1']}

PKLR is directly connected to: ['ENO1', 'ENO3', 'ENO2', 'GPI', 'LDHB', 'LDHC', 'TPI1',
↪ 'PGK1', 'PGK2', 'ME3']
```

(continues on next page)

(continued from previous page)

```
ME3 is directly connected to: ['PKLR', 'LDHB', 'LDHC', 'GPI', 'TPI1']

ME3's second neighbours: ['ENO1', 'ENO3', 'ENO2', 'PGK1', 'PGK2']
```

CHAPTER

SEVEN

PRACTICAL 6

In this practical we will see how to define functions to reuse code, we will talk about the scope of variables and finally will see how to deal with files in Python.

7.1 Slides

The slides of the introduction can be found here: Intro

7.2 Functions

A function is a block of code that has a name and that performs a task. A function can be thought of as a box that gets an input and returns an output.

Why should we use functions? For a lot of reasons including:

1. *Reduce code duplication*: put in functions parts of code that are needed several times in the whole program so that you don't need to repeat the same code over and over again;
2. *Decompose a complex task*: make the code easier to write and understand by splitting the whole program into several easier functions;

both things improve code readability and make your code easier to understand.

The basic definition of a function is:

```
def function_name(input) :
    #code implementing the function
    ...
    ...
    return return_value
```

Functions are defined with the **def** keyword that proceeds the *function_name* and then a list of parameters is passed in the brackets. A colon **:** is used to end the line holding the definition of the function. The code implementing the function is specified by using indentation. A function **might** or **might not** return a value. In the first case a **return** statement is used.

Consider the following example in which we want to compute the sum of the square root of the values in lists X, Y, Z.

[1]:
```
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
```

(continues on next page)

(continued from previous page)

```
Z = [9, 9, 4, 7]

sum_x = 0
sum_y = 0
sum_z = 0

for el in X:
    sum_x += math.sqrt(el)

for el in Y:
    sum_y += math.sqrt(el)

for el in Z:
    sum_z += math.sqrt(el)

print(X, "sum_sqrt:", sum_x)
print(Y, "sum_sqrt:", sum_y)
print(Z, "sum_sqrt:", sum_z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
```

In the above code there is a lot of duplication. Let's try and rewrite the above code taking advantage of functions.

[2]:
```
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
Z = [9, 9, 4, 7]

# This function does not return anything
def print_sum_sqrt(vals):
    tmp = 0
    for el in vals:
        tmp += math.sqrt(el)
    print(vals, "sum_sqrt:", tmp)

print_sum_sqrt(X)
print_sum_sqrt(Y)
print_sum_sqrt(Z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
```

If we want, we can modify `print_sum_sqrt` to output the values:

[3]:
```
import math

X = [1, 5, 4, 4, 7, 2, 1]
Y = [9, 4, 7, 1, 2]
Z = [9, 9, 4, 7]

# This function returns the sum
def sum_sqrt(vals):
```

(continues on next page)

(continued from previous page)

```
    tmp = 0
    for el in vals:
        tmp += math.sqrt(el)

    return tmp

x = sum_sqrt(X)
y = sum_sqrt(Y)
z = sum_sqrt(Z)

print(X, "sum_sqrt:", x)
print(Y, "sum_sqrt:", y)
print(Z, "sum_sqrt:", z)
# we have the sums as numbers, can use them
print("Sum of all: ", x + y + z)
```

```
[1, 5, 4, 4, 7, 2, 1] sum_sqrt: 12.296032850937475
[9, 4, 7, 1, 2] sum_sqrt: 10.059964873437686
[9, 9, 4, 7] sum_sqrt: 10.64575131106459
Sum of all:  33.00174903543975
```

Example: Define a function that implements the sum of two lists of integers (note that there is no check that the two lists actually contain integers and that they have the same size).

[4]:
```
def int_list_sum(l1,l2):
    """implements the sum of two lists of integers having the same size
    """
    ret =[]
    for i in range(len(l1)):
        ret.append(l1[i] + l2[i])
    return ret

L1 = list(range(1,10))
L2 = list(range(20,30))
print("L1:", L1)
print("L2:", L2)

res = int_list_sum(L1,L2)

print("L1+L2:", res)

res = int_list_sum(L1,L1)

print("L1+L1", res)
```

```
L1: [1, 2, 3, 4, 5, 6, 7, 8, 9]
L2: [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
L1+L2: [21, 23, 25, 27, 29, 31, 33, 35, 37]
L1+L1 [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Note that once the function has been defined, it can be called as many times as wanted with different input parameters. Moreover, **a function does not do anything until it is actually called.** A function can return **0** (in this case the return value would be "None"), **1** or **more** results. Notice also that collecting the results of a function is **not mandatory**.

Example: Let's write a function that, given a list of elements, prints only the even-placed ones without returning anything.

[5]:
```
def get_even_placed(myList):
    """returns the even placed elements of myList"""
```

(continues on next page)

(continued from previous page)

```
    ret = [myList[i] for i in range(len(myList)) if i % 2 == 0]
    print(ret)

L1 = ["hi", "there", "from","python","!"]
L2 = list(range(13))

print("L1:", L1)
print("L2:", L2)

print("even L1:")
get_even_placed(L1)
print("even L2:")
get_even_placed(L2)
```

```
L1: ['hi', 'there', 'from', 'python', '!']
L2: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
even L1:
['hi', 'from', '!']
even L2:
[0, 2, 4, 6, 8, 10, 12]
```

Note that the function above is polymorphic (i.e. it works on several data types, provided that we can iterate through them).

Example: Let's write a function that, given a list of integers, returns the number of elements, the maximum and minimum.

[6]:
```
"""easy! this changes the original list!!!"""
def get_info(myList):
    """returns len of myList, min and max value
    (assumes elements are integers) but it would work with str"""

    myList.sort()
    return len(myList), myList[0], myList[-1] #return type is a tuple

A = [7, 1, 125, 4, -1, 0]

print("Original A:", A, "\n")
result = get_info(A)
print("Len:", result[0], "Min:", result[1], "Max:",result[2], "\n" )

print("A now:", A)
```

```
Original A: [7, 1, 125, 4, -1, 0]

Len: 6 Min: -1 Max: 125

A now: [-1, 0, 1, 4, 7, 125]
```

We need to make a copy if we do not want to affect the caller (see why below):

[39]:
```
def get_info(myList):
    """returns len of myList, min and max value
    (assumes elements are integers) but it would work with str"""
    tmp = myList[:] #copy the input list
    tmp.sort()
    return len(tmp), tmp[0], tmp[-1] #return type is a tuple

A = [7, 1, 125, 4, -1, 0]
```

(continues on next page)

(continued from previous page)

```
print("Original A:", A, "\n")
result = get_info(A)
print("Len:", result[0], "Min:", result[1], "Max:",result[2], "\n" )

print("A now:", A)
```

```
Original A: [7, 1, 125, 4, -1, 0]

Len: 6 Min: -1 Max: 125

A now: [7, 1, 125, 4, -1, 0]
```

[8]:
```
def my_sum(myList):
    ret = 0
    for el in myList:
        ret += el # == ret = ret + el
    return ret

A = [1,2,3,4,5,6]
B = [7, 9, 4]
s = my_sum(A)

print("List A:", A)
print("Sum:", s)
s = my_sum(B)
print("List B:", B)
print("Sum:", s)
```

```
List A: [1, 2, 3, 4, 5, 6]
Sum: 21
List B: [7, 9, 4]
Sum: 20
```

Please note that the return value above is actually a tuple. Importantly enough, a function needs to be defined (i.e. its code has to be written) before it can actually be used otherwise the Python interpreter does not know what to do when the function is actually called with some parameters.

[9]:
```
A = [1,2,3]
my_sum_new(A)

def my_sum_new(myList):
    ret = 0
    for el in myList:
        ret += el
    return ret
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-50f5bebe9695> in <module>
      1 A = [1,2,3]
----> 2 my_sum_new(A)
      3
      4 def my_sum_new(myList):
      5     ret = 0

NameError: name 'my_sum_new' is not defined
```

7.3 Namespace and variable scope

Namespaces are the way Python makes sure that names in a program are unique and can be safely used without any conflicts. They are mappings from *names* to objects, or in other words places where names are associated to objects. Namespaces can be considered as the context and in Python are implemented as dictionaries that map the *name* to the *object*. According to Python's reference a **scope** is a *textual region of a Python program, where a namespace is directly accessible*, which means that Python will look into that *namespace* to find the object associated to a name. Four **namespaces** are made available by Python:

```
1. **Local**: the innermost that contains local names (inside a function or a class);

2. **Enclosing**: the scope of the enclosing function,
it does not contain local nor global names (nested functions) ;

3. **Global**: contains the global names;

4. **Built-in**: contains all built in names
(e.g. print, if, while, for,...)
```

When one refers to a name, Python tries to find it in the current namespace, if it is not found it continues looking in the namespace that contains it until the built-in namespace is reached. If the name is not found there either, the Python interpreter will throw a **NameError** exception, meaning it cannot find the name. The order in which namespaces are considered is: Local, Enclosing, Global and Built-in (LEGB).

Consider the following example:

[10]:
```
def my_function():
    var = 1  #local variable
    print("Local:", var)
    b = "my string"
    print("Local:", b)

var = 7 #global variable
my_function()
print("Global:", var)
print(b)
```

```
Local: 1
Local: my string
Global: 7
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-da11a41b34ff> in <module>
      8 my_function()
      9 print("Global:", var)
---> 10 print(b)
     11

NameError: name 'b' is not defined
```

Variables defined within a function can only be seen within the function. That is why variable b is defined only within the function. Variables defined outside all functions are **global** to the whole program. The namespace of the local variable is within the function my_function, while outside it the variable will have its global value.

And the following:

[11]:
```
def outer_function():
    var = 1 #outer

    def inner_function():
        var = 2 #inner
        print("Inner:", var)
        print("Inner:", B)

    inner_function()
    print("Outer:", var)


var = 3 #global
B = "This is B"
outer_function()
print("Global:", var)
print("Global:", B)
```

```
Inner: 2
Inner: This is B
Outer: 1
Global: 3
Global: This is B
```

Note in particular that the variable B is global, therefore it is accessible everywhere and also inside the inner_function. On the contrary, the value of var defined within the inner_function is accessible only in the namespace defined by it, outside it will assume different values as shown in the example.

In a nutshell, remember the three simple rules seen in the lecture. Within a **def**:

```
1. Name assignments create local names by default;
2. Name references search the following four scopes in the order:
local, enclosing functions (if any), then global and finally built-in (LEGB)
3. Names declared in global and nonlocal statements map assigned names to
enclosing module and function scopes.
```

7.4 Argument passing

Arguments are the parameters and data we pass to functions. When passing arguments, there are three important things to bear in mind:

1. Passing an argument is actually assigning an object to a local variable name;
2. Assigning an object to a variable name within a function **does not affect the caller**;
3. Changing a **mutable** object variable name within a function **affects the caller**

Consider the following examples:

[15]:
```
"""Assigning the argument does not affect the caller"""

def my_f(x):
    x = "local value" #local
    print("Local: ", x)

x = "global value" #global
my_f(x)
```

(continues on next page)

(continued from previous page)

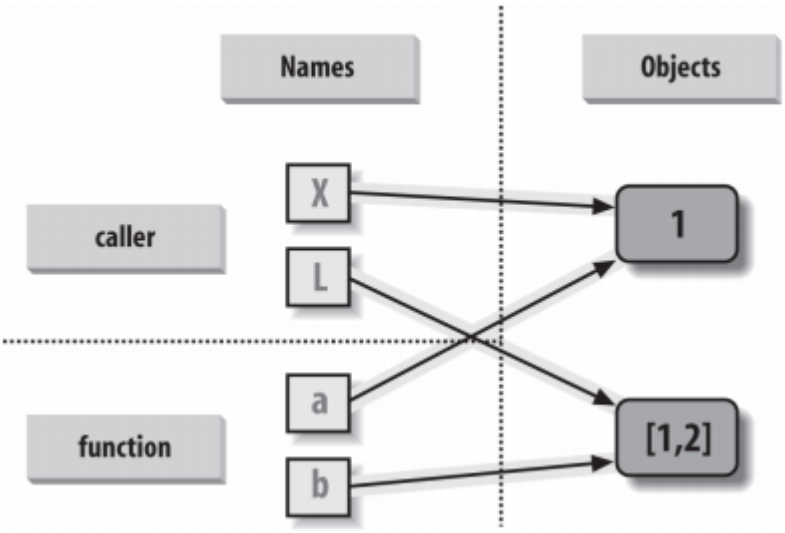
```
print("Global:", x)
my_f(x)
```

```
Local:  local value
Global: global value
Local:  local value
```

[16]:
```
"""Changing a mutable affects the caller"""

def my_f(myList):
    myList[1] = "new value1"
    myList[3] = "new value2"
    print("Local: ", myList)

myList = ["old value"]*4
print("Global:", myList)
my_f(myList)
print("Global now: ", myList)
```

```
Global: ['old value', 'old value', 'old value', 'old value']
Local:  ['old value', 'new value1', 'old value', 'new value2']
Global now:  ['old value', 'new value1', 'old value', 'new value2']
```

Recall what seen in the lecture:



The behaviour above is because **immutable objects** are passed **by value** (therefore it is like making a copy), while **mutable objects** are passed **by reference** (therefore changing them effectively changes the original object).

To avoid making changes to a **mutable object** passed as parameter one needs to **explicitely make a copy** of it.

Consider the example seen before. **Example:** Let's write a function that, given a list of integers, returns the number of elements, the maximum and minimum.

[17]:
```
def get_info(myList):
    """returns len of myList, min and max value (assumes elements are integers)"""
    myList.sort()
```

(continues on next page)

(continued from previous page)

```
    return len(myList), myList[0], myList[-1] #return type is a tuple


def get_info_copy(myList):
    """returns len of myList, min and max value (assumes elements are integers)"""
    tmp = myList[:] #copy the input list!!!!
    tmp.sort()
    return len(tmp), tmp[0], tmp[-1] #return type is a tuple

A = [7, 1, 125, 4, -1, 0]
B = [70, 10, 1250, 40, -10, 0, 10]

print("A:", A)
result = get_info(A)
print("Len:", result[0], "Min:", result[1], "Max:",result[2] )

print("A now:", A) #whoops A is changed!!!

print("\n###### With copy now ########")

print("\nB:", B)
result = get_info_copy(B)
print("Len:", result[0], "Min:", result[1], "Max:",result[2] )

print("B now:", B) #B is not changed!!!
```

```
A: [7, 1, 125, 4, -1, 0]
Len: 6 Min: -1 Max: 125
A now: [-1, 0, 1, 4, 7, 125]

###### With copy now ########

B: [70, 10, 1250, 40, -10, 0, 10]
Len: 7 Min: -10 Max: 1250
B now: [70, 10, 1250, 40, -10, 0, 10]
```

7.4.1 Positional arguments

Arguments can be passed to functions following the order in which they appear in the function definition.

Consider the following example:

[18]:
```
def print_parameters(a,b,c,d):
    print("1st param:", a)
    print("2nd param:", b)
    print("3rd param:", c)
    print("4th param:", d)


print_parameters("A", "B", "C", "D")
```

```
1st param: A
2nd param: B
3rd param: C
4th param: D
```

7.4.2 Passing arguments by keyword

Given the name of an argument as specified in the definition of the function, parameters can be passed using the **name = value** syntax.

For example:

[19]:
```
def print_parameters(a,b,c,d):
    print("1st param:", a)
    print("2nd param:", b)
    print("3rd param:", c)
    print("4th param:", d)


print_parameters(a = 1, c=3, d=4, b=2)
print("\n###############\n")
print_parameters("first","second",d="fourth",c="third")
print("\n###############\n")
```

```
1st param: 1
2nd param: 2
3rd param: 3
4th param: 4

###############

1st param: first
2nd param: second
3rd param: third
4th param: fourth

###############
```

Arguments passed positionally and by name can be used at the same time, but parameters passed by name must always be to the left of those passed by name. The following code in fact is not accepted by the Python interpreter:

[20]:
```
def print_parameters(a,b,c,d):
    print("1st param:", a)
    print("2nd param:", b)
    print("3rd param:", c)
    print("4th param:", d)


#The following won't work
print_parameters(d="fourth",c="third", "first","second")

#Correct code instead:
print_parameters("first","second", d="fourth",c="third")
```

```
  File "<ipython-input-20-0aba188b6596>", line 9
    print_parameters(d="fourth",c="third", "first","second")
                                          ^
SyntaxError: positional argument follows keyword argument
```

7.4.3 Specifying default values

During the definition of a function it is possible to specify default values. The syntax is the following:

```
def my_function(par1 = val1, par2 = val2, par3 = val3):
```

Consider the following example:

[21]:
```
def print_parameters(a="defaultA", b="defaultB",c="defaultC"):
    print("a:",a)
    print("b:",b)
    print("c:",c)

print_parameters("param_A")
print("\n##################\n")
print_parameters(b="PARAMETER_B")
print("\n##################\n")
print_parameters()
print("\n##################\n")
print_parameters(c="PARAMETER_C", b="PAR_B")
```

```
a: param_A
b: defaultB
c: defaultC

##################

a: defaultA
b: PARAMETER_B
c: defaultC

##################

a: defaultA
b: defaultB
c: defaultC

##################

a: defaultA
b: PAR_B
c: PARAMETER_C
```

Another example.

Example. Write a function that rounds a float at a precision (i.e. number of decimals) specified in input. If no precision is specified then the whole number should be returned. Examples:

```
my_round(1.1717413, 3) = 1.172 my_round(1.1717413, 1) = 1.2 my_round(1.1717413)
= 1.17174.
```

[22]:
```
import math

def my_round(val, precision = 0):
    if precision == 0:
        return val
    else:
        return round(val * 10** precision)/ 10**precision
```

(continues on next page)

(continued from previous page)

```
my_val = 1.717413

print(my_val, " precision 2: ",  my_round(my_val,2))
print(my_val, " precision 1: ",  my_round(my_val,1))
print(my_val, " precision max: ",  my_round(my_val))
print("")
my_val = math.pi
print(my_val, " precision 10: ",  my_round(my_val,10))
```

```
1.717413  precision 2:  1.72
1.717413  precision 1:  1.7
1.717413  precision max:  1.717413

3.141592653589793  precision 10:  3.1415926536
```

We can also use the function implemented and apply it to a list of values through list comprehension.

Example: create a list with the square root values of the first 20 integers with 3 digits of precision.

[23]:
```
import math

def my_round(val, precision = 0):
    if precision == 0:
        return val
    else:
        return round(val * 10** precision)/ 10**precision

result = [my_round(math.sqrt(x), 3) for x in range(1,21)]

print(result)
```

```
[1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.646, 2.828, 3.0, 3.162, 3.317, 3.464, 3.606,␣
↪3.742, 3.873, 4.0, 4.123, 4.243, 4.359, 4.472]
```

As a final example of list comprehension, let's print only the values of the list `result` above whose digits sum up to a certain value x. Hint: write another function!

[24]:
```
import math

def my_round(val, precision = 0):
    if precision == 0:
        return val
    else:
        return round(val * 10** precision)/ 10**precision

#version without list comprehension
def sum_of_digits_noList(num, total):
    tmp = str(num)
    tot = 0
    for d in tmp:
        if d != ".":
            tot += int(d)
    if tot == total:
        return True
    else:
        return False
```

(continues on next page)

(continued from previous page)

```
#with list comprehension
def sum_of_digits(num, total):
    tmp = [int(x) for x in str(num) if x != "."]
    return sum(tmp) == total


result = [my_round(math.sqrt(x), 3) for x in range(1,21)]
print("sum is 10:", [x for x in result if sum_of_digits(x, 10)])
print("sum is 13:",[x for x in result if sum_of_digits(x, 13)])
```

```
sum is 10: [1.414, 4.123]
sum is 13: [1.732, 2.236, 4.243]
```

7.5 File input and output

Files are non volatile data stored at specific locations in memory. To read or write from/to a file, one first needs to create a reference to the file. This is called the **handle** to the file.

Recall from the lecture that Python provides programmers with the following functions and methods:

Result	Built-in function	Meaning
`file`	`open(str, [str])`	Get a handle to a file

Result	Method	Meaning
`str`	`file.read()`	Read all the file as a single string
`list of str`	`file.readlines()`	Read all lines of the file as a list of strings
`str`	`file.readline()`	Read one line of the file as a string
`None`	`file.write(str)`	Write one string to the file
`None`	`file.close()`	`Close the file` (i.e. flushes changes to disk)

To read or write to a file one first needs to open it with the python's built-in function **open** that returns a handler to the file. The handler can then be used to read or write the content of the file.

Example: Download the file textFile.txt to your favourite location and read it from python, printing out all its content. **NOTE:** the path of your file might be different from mine.

[25]:
```
fh = open("file_samples/textFile.txt", "r") #read-only mode

print("--- File content ---")

for line in fh:
    print(line, end ="") #no need to add a new line

fh.close()

print("\n--- File closed ---")
```

```
--- File content ---
Hi everybody,
This is my first file
```

(continues on next page)

(continued from previous page)

```
and it contains a total of
four lines!
--- File closed ---
```

The usual interaction with files is done in three steps: **open the file**, **read or write its content**, **close the file**.

7.5.1 Opening a file

As said, a file needs to be opened before working with it. This can be done by using the function **open**.

The standard syntax is:

```
file_handle = open("file_name", "file_mode")
```

where `file_name` is a string representing the path to reach the file (either absolute or relative to the .py module we are writing) and `file_mode` is the string representing the opening mode of the file. `file_mode` can be:

1. `"r"` which is the default and opens the file in read only mode;
2. `"w"` write only mode. **Note that this would overwrite an existing file**;
3. `"a"` append mode (i.e. add information at the end of an existing file or create a new file)
4. `"t"` text mode. The file will be a text file (this is the default).
5. `"b"` binary mode. The file will not be a textual file.
6. `"x"` exclusive creation. This will fail if the file already exists.
7. `"+"` update mode. The file will be opened with read and write permissions.

Opening modes can be combined (ex. `"ra"` would read and append, `"wb"` would write (and overwrite) a binary file, `"r+b"` read and write a binary file).

7.5.2 Closing a file

A opened file always needs to be closed. The basic syntax is:

```
file_handle.close()
```

this will free up resources and will not allow any more operations on the same file.

The following code will in fact crash, because the file has been closed and therefore no more operations can be performed on it.

[26]:
```
fh = open("file_samples/textFile.txt", "r") #read-only mode

print("--- File content ---")

for line in fh:
    print(line, end ="") #no need to add a new line

fh.close()

print("\n--- File closed ---")

print(fh.readline())
```

```
--- File content ---
Hi everybody,
This is my first file
and it contains a total of
four lines!
--- File closed ---
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-0367c74d60af> in <module>
     10 print("\n--- File closed ---")
     11
---> 12 print(fh.readline())

ValueError: I/O operation on closed file.
```

An alternative way to open and implicitly close the file is by using the **with** keyword. The basic syntax is:

```
with open("file") as f:
    #do some stuff
    #until done with it
```

The previous example would become:

[27]:
```
print("--- File content ---")

with open("file_samples/textFile.txt", "r") as f:
    for line in f:
        print(line, end ="") #no need to add a new line

print("\n--- File closed ---")
```

```
--- File content ---
Hi everybody,
This is my first file
and it contains a total of
four lines!
--- File closed ---
```

7.5.3 Reading from a file

There are several ways for reading a file that has been opened in *read* mode having file handle `fh`:

1. `content = fh.read()` reads the whole file in the content string. Good for small and not structured files.
2. `line = fh.readline()` reads the file one line at a time storing it in the string `line`
3. `lines = fh.readlines()` reads all the lines of the file storing them as a list `lines`
4. using the iterator:

   ```
   for line in f:
   #process the information
   ```

   which is the most convenient way for big files.

Let's see some examples. Note that to restart reading a file from the beginning we need to close it and open it gain.

[28]:
```
fh = open("file_samples/textFile.txt", "r") #read-only mode

content = fh.read()
print("--- Mode1 (the whole file in a string) ---")
print(content)
fh.close()
print("")
print("--- Mode2 (line by line) ---")
with open("file_samples/textFile.txt","r") as f:
    print("Line1: ", f.readline(), end = "")
    print("Line2: ", f.readline(), end = "")

print("")
print("--- Mode3 (all lines as a list) ---")
with open("file_samples/textFile.txt","r") as f:
    print(f.readlines())

print("")
print("--- Mode4 (as a stream) ---")
with open("file_samples/textFile.txt","r") as f:
    for line in f:
        print(line, end = "")
```

```
--- Mode1 (the whole file in a string) ---
Hi everybody,
This is my first file
and it contains a total of
four lines!

--- Mode2 (line by line) ---
Line1:  Hi everybody,
Line2:  This is my first file

--- Mode3 (all lines as a list) ---
['Hi everybody,\n', 'This is my first file\n', 'and it contains a total of\n', 'four␣
↪lines!']

--- Mode4 (as a stream) ---
Hi everybody,
This is my first file
and it contains a total of
four lines!
```

7.5.4 Writing to a file

To write some data to a file it must first be opened in write mode (i.e. "w") or append mode ("a", "+"). The actual writing can be performed by the method **write** that has the following basic syntax:

```
file_handle.write(data_to_be_written)
```

To make sure all data is written to the file, we must remember to close the file after writing on it.

Example: Given the matrix represented as list of lists, M = [[1,2,3], [4,5,6], [7,8,9]] let's write it on a file *my_matrix.txt*

$$\text{as } \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

[45]:
```
M =  [[1,2,3], [4,5,6], [7,8,9]]

with open("file_samples/my_matrix.txt", "w") as f:
    for line in M:
        str_line = [str(x) for x in line] #to make this "joinable"
        f.write(" ".join(str_line))
        f.write("\n")

#no need to put the close because we used with

#Equivalent code (without with clause):
M =  [[1,2,3], [4,5,6], [7,8,9]]

f = open("file_samples/my_matrix.txt", "w")
for line in M:
    str_line = [str(x) for x in line]
    f.write(" ".join(str_line))
    f.write("\n")
f.close()
```

7.5.5 String formatting with format

Strings can be formatted with the method **format**. A lot of information on the use of format can be found here[42].

Format can be used to add values to a string in specific placeholders (normally defined with the syntax **{}**) or to format values according to the user specifications (e.g. number of decimal places for floating point numbers).

In the following we will see some simple examples.

[30]:
```
#simple empty placeholders
print("I like {} more than {}.\n".format("python", "java"))

#indexed placeholders, note order
print("I like {0} more than {1} or {2}.\n".format("python", "java", "C++"))
print("I like {2} more than {1} or {0}.\n".format("python", "java", "C++"))

#indexed and named placeholders
print("I like {1} more than {c} or {0}.\n".format("python", "java", c="C++"))

#with type specification
import math
print("The square root of {0} is {1:f}.\n".format(2, math.sqrt(2)))

#with type and format specification (NOTE: {.2f})
print("The square root of {0} is {1:.2f}.\n".format(2, math.sqrt(2)))

#spacing data properly
print("{:2s}|{:5}|{:6}".format("N","sqrt","square"))
for i in range(0,20):
    print("{:2d}|{:5.3f}|{:6d}".format(i,math.sqrt(i),i*i))
```

```
I like python more than java.

I like python more than java or C++.
```

(continues on next page)

[42] https://docs.python.org/3/library/string.html#format-string-syntax

(continued from previous page)

```
I like C++ more than java or python.

I like java more than C++ or python.

The square root of 2 is 1.414214.

The square root of 2 is 1.41.

N |sqrt |square
 0|0.000|     0
 1|1.000|     1
 2|1.414|     4
 3|1.732|     9
 4|2.000|    16
 5|2.236|    25
 6|2.449|    36
 7|2.646|    49
 8|2.828|    64
 9|3.000|    81
10|3.162|   100
11|3.317|   121
12|3.464|   144
13|3.606|   169
14|3.742|   196
15|3.873|   225
16|4.000|   256
17|4.123|   289
18|4.243|   324
19|4.359|   361
```

7.6 Exercises

1. Implement a function that takes in input a string representing a DNA sequence and computes its reverse-complement. Take care to reverse complement any character other than (A,T,C,G,a,t,c,g) to N. The function should preserve the case of each letter (i.e. A becomes T, but a becomes t). For simplicity all bases that do not represent nucleotides are converted to a capital N. **Hint: create a dictionary revDict with bases as keys and their complements as values**. Ex. revDict = {"A" : "T" , "a" : "t", ...}.
    1. Apply the function to the DNA sequence "ATTACATATCATACTATCGCNTTCTAAATA"
    2. Apply the function to the DNA sequence "acaTTACAtagataATACTaccataGCNTTCTAAATA"
    3. Apply the function to the DNA sequence "TTTTACCKKKAKTUUUITTTARRRRRAIUTYYA"
    4. Check that the reverse complement of the reverse complement of the sequence in 1. is exactly as the original sequence.

Show/Hide Solution

[31]:
```
def reverse_complement(DNA):
    """the function reverse complements a string of DNA"""
    revDict = {"A" : "T", "a" : "t",
               "C" : "G", "c" : "g",
               "G" : "C", "g" : "c",
               "T" : "A", "t" : "a"
```

(continues on next page)

(continued from previous page)

```
            }
    result = ""
    for base in DNA:
        ### dict.get(val, default) : returns default if val not in
        result = revDict.get(base, "N") + result

    return result

# Second version (equivalent)
def reverse_complement2(DNA):
    """the function reverse complements a string of DNA"""
    revDict = {"A" : "T", "a" : "t",
               "C" : "G", "c" : "g",
               "G" : "C", "g" : "c",
               "T" : "A", "t" : "a"
              }

    result = ""
    DNAlist = list(DNA)
    DNAlist.reverse()
    revComp =[revDict.get(x, "N") for x in DNAlist]
    result = "".join(revComp)
    return result

# Third version (equivalent) to version2, "lazy" creation of dictionary
def reverse_complement3(DNA):
    """the function reverse complements a string of DNA"""
    k = "AaTtCcGg"
    v = "TtAaGgCc"
    revDict  = dict()
    for i in range(0,len(k)):
        revDict[k[i]] = v[i]
    result = ""
    DNAlist = list(DNA)
    DNAlist.reverse()
    revComp =[revDict.get(x, "N") for x in DNAlist]
    result = "".join(revComp)
    return result


S1 = "ATTACATATCATACTATCGCNTTCTAAATA"
S2 = "acaTTACAtagataATACTaccataGCNTTCTAAATA"
S3 = "TTTTACCKKKAKTUUUITTTARRRRRAIUTYYA"

print("S1 {}".format(S1) )
print("rc S1", reverse_complement(S1))

print("\nS2   ", S2)
print("rc S2", reverse_complement(S2))

print("\nS3   ", S3)
print("rc S3", reverse_complement(S3))

print("\nrev_comp(rev_comp(S1)): {}".format(reverse_complement(reverse_
↪complement(S1))))
print("\nrev_comp(rev_comp(S1) == S1?)", reverse_complement(reverse_complement(S1))␣
↪== S1)
```

```
S1 ATTACATATCATACTATCGCNTTCTAAATA
rc S1 TATTTAGAANGCGATAGTATGATATGTAAT

S2    acaTTACAtagataATACTaccataGCNTTCTAAATA
rc S2 TATTTAGAANGCtatggtAGTATtatctaTGTAAtgt

S3    TTTTACCKKKAKTUUUITTTARRRRRAIUTYYA
rc S3 TNNANNTNNNNNTAAANNNNANTNNNGGTAAAA

rev_comp(rev_comp(S1)): ATTACATATCATACTATCGCNTTCTAAATA

rev_comp(rev_comp(S1) == S1?) True
```

2. Write the following python functions and test them with some parameters of your choice:
    1. *getDivisors*: the function has a positive integer as parameter and returns a list of all the positive divisors of the integer in input (excluding the number itself). Example: `getDivisors(6) --> [1,2,3]`
    2. *checkSum*: the function has a list and an integer as parameters and returns True if the sum of all elements in the list equals the integer, False otherwise. Example: `checkSum([1,2,3], 6) --> True`, `checkSum([1,2,3],1) --> False`.
    3. *checkPerfect*: the function gets an integer as parameter and returns True if the integer is a perfect number[43], False otherwise. A number is perfect if all its divisors (excluding itself) sum to its value. Example: `checkPerfect(6) --> True` because 1+2+3 = 6. Hint: use the functions implemented before.

Use the three implemented functions to write a fourth function:

getFirstNperfects: the function gets an integer N as parameter and returns a dictionary with the first N perfect numbers. The key of the dictionary is the perfect number, while the value of the dictionary is the list of its divisors. Example: `getFirstNperfects(1) --> {6 : [1,2,3]}`

Get and print the first 4 perfect numbers and finally test if 33550336 is a perfect number.

WARNING: do not try to find more than 4 perfect numbers as it might take a while!!!

Show/Hide Solution

[41]:
```
def getDivisors(intVal):
    """returns the integer divisors of intVal"""
    ret = [x for x in range(1,intVal//2 + 1) if intVal % x == 0]
    #OR:
    #for i in range(1,intVal//2+1):
    #    if intVal % i == 0:
    #        ret.append(i)
    return ret

def checkSum(intList, intVal):
    """checks if the sum of elements in intList equals intVal"""
    s = 0
    for x in intList:
        s += x
    return s == intVal

def checkPerfect(intVal):
    """checks if intVal is a perfect number"""
    divisors = getDivisors(intVal)
    return checkSum(divisors,intVal)
```

(continues on next page)

[43] https://en.wikipedia.org/wiki/Perfect_number

(continued from previous page)

```
def getFirstNPerfects(N):
    """Finds the first N perfect numbers"""
    i = 0
    val = 2
    ret = {}
    while i<N:
        if checkPerfect(val):
            i+=1
            ret[val] = getDivisors(val)
        val += 1

    return ret


perfects = getFirstNPerfects(4) #do not go higher than 4, or it might take a while
perKeys = list(perfects.keys())
perKeys.sort()
for p in perKeys:
    print(p, " = ", "+".join([str(x) for x in perfects[p]]))

print("Is 33550336 a perfect number?", checkPerfect(33550336))
```

```
6  =  1+2+3
28  =  1+2+4+7+14
496  =  1+2+4+8+16+31+62+124+248
8128  =  1+2+4+8+16+32+64+127+254+508+1016+2032+4064
Is 33550336 a perfect number? True
```

3. A pangram is a sentence that contains all the letters of the alphabet at least once (let's ignore numbers and other characters for simplicity). 1. Write a function that gets a text in input and returns True if it is a pangram. 2. Write a function that computes how many unique characters an input sentence has; 3. Modify the function at 1. in such a way that a user can specify an alphabet and check if the provided text is a pangram in that reference alphabet.

Show/Hide Solution

[33]:

```
# 1. Write a function that gets a text in input and returns True if it is a pangram.

def pangram_check(my_str):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    chars = list(alphabet)
    occurrences = [0 for x in chars]
    counts = dict(zip(chars,occurrences))

    for c in my_str.upper():
        if c in counts:
            counts[c] += 1

    found_chars = [x for x in counts if counts[x] > 0]

    return len(found_chars) == len(alphabet)


#2. Write a function that computes how many unique characters an input sentence has
def count_unique(my_str):
    tmp_str = my_str.upper()
```

(continues on next page)

(continued from previous page)

```
    counts = dict()
    for c in tmp_str:
        #if we do not want to count spaces:
        if c != " ":
            if counts.get(c, None) == None:
                counts[c] = 1
    return len(counts)



#3. Modify the function at 1. in such a way that a user can specify an alphabet
# and check if the provided text is a pangram in that reference alphabet.

def pangram_check_v2(my_str, alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ"):
    # the same as before, just comment the first line:
    #alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    chars = list(alphabet)
    occurrences = [0 for x in chars]
    counts = dict(zip(chars,occurrences))

    for c in my_str.upper():
        if c in counts:
            counts[c] += 1

    found_chars = [x for x in counts if counts[x] > 0]

    return len(found_chars) == len(alphabet)

txt = 'The quick brown fox jumps over the lazy dog'
txt1 = 'The quick brown fox jumps over the lzy dog'
txt2 = 'The five boxing wizards jump quickly'
txt3 = "The quick brown fox jumps over the dog"
print("{}, is it a pangram? {}".format(txt, pangram_check(txt)))
print("{}, is it a pangram? {}".format(txt1, pangram_check(txt1)))

print("")
print("{}: {} unique chars".format(txt, count_unique(txt)))
print("{}: {} unique chars".format(txt1, count_unique(txt1)))
print("")
print("{}, is it a pangram? {}".format(txt3, pangram_check_v2(txt3)))
print("{}, is it a pangram in mod alphabet? {}".format(txt3,
                                                       pangram_check_v2(txt3,

↪"BCDEFGHIJKMNOPQRSTUVWX")))
```

```
The quick brown fox jumps over the lazy dog, is it a pangram? True
The quick brown fox jumps over the lzy dog, is it a pangram? False

The quick brown fox jumps over the lazy dog: 26 unique chars
The quick brown fox jumps over the lzy dog: 25 unique chars

The quick brown fox jumps over the dog, is it a pangram? False
The quick brown fox jumps over the dog, is it a pangram in mod alphabet? True
```

4. A restriction enzyme[44] is an enzyme that cleaves DNA into fragments at or near specific recognition sites within molecules known as restriction sites. A lot of restriction enzymes exist that cleave the DNA at different sites. For simplicity let's focus on only the 5'-3' orientation of DNA. Consider the following enzymes, stored as a dictionary:

```
restriction_enzymes = {'EcoRI' : ('GAATTC','G','AATTC'), 'TaqI': ('TCGA','T','CGA'),
↪'PvII: ('CAGCTG','CAG','CTG')}
```

meaning for example that enzyme 'EcoRI' would split a sequence xxxxGAATTCxxxxx into the two sequences xxxxG and AATTCxxxxx.

A. Implement a function (*restrict_DNA(sequence, enzyme)*) that takes a DNA string in input, and the specification of a restriction enzyme (e.g. "TaqI") and returns a list with all the possible pairs of cleaved DNA sequences. B. Extend this code in such a way that it can process a list of DNA sequences rather than a single one and returns a list of lists (write a function *restrict_DNA_sequences(list_of_sequences, enzyme)*. **If the enzyme specified in input is not recognized, the script should output a warning message without crashing**.

Test your function(s) with the following calls:

```
seq = "TACTACCTACGAATTCAAATGAATTCCCTTAAGAATTCATTCGATCGATTAAAGAGCTGATTAGAATATCCAGCTG"

print("Restriction with EcoRI")
print(restrict_DNA(seq, 'EcoRI'))
print("\nRestriction with PvII")
print(restrict_DNA(seq, 'PvII'))
print("\nRestriction with TaqI")
R = restrict_DNA(seq, 'TaqI')
print(R)
print("\nSecond restriction with EcoRI")
print(restrict_DNA_sequences(R, 'EcoRI'))
print(restrict_DNA_sequences(R, 'MyEnzyme'))
```

Show/Hide Solution

[34]:

```
def restrict_DNA(sequence, enzyme):
    restriction_enzymes = { 'EcoRI' : ('GAATTC','G','AATTC'),
                            'TaqI': ('TCGA','T','CGA'),
                            'PvII': ('CAGCTG','CAG','CTG')
                           }

    site = restriction_enzymes.get(enzyme, None)

    if site == None:
        print("Enzyme '{} does not exist".format(enzyme))
    else:
        index = 0

        positions = [x for x in range(len(sequence)) if sequence[x:x + len(site[0])]␣
↪== site[0]]
        ret = [(sequence[0: x+len(site[1])], sequence[x+len(site[1]) : ]) for x in␣
↪positions]
        # if restriction site not present:
        # we need to return [(sequence, "")]
        if len(ret) == 0:
            return [(sequence,'')]
```

(continues on next page)

[44] https://en.wikipedia.org/wiki/Restriction_enzyme

(continued from previous page)

```
        return ret

def restrict_DNA_sequences(seqs, enzyme):
    ret = []
    for s in seqs:
        for i in range(2):
            r = restrict_DNA(s[i], enzyme)
            if r != None:
                ret.append(r)
            else:
                return None
    return ret


seq = "TACTACCTACGAATTCAAATGAATTCCCTTAAGAATTCATTCGATCGATTAAAGAGCTGATTAGAATATCCAGCTG"

print("Restriction with EcoRI")
print(restrict_DNA(seq, 'EcoRI'))
print("\nRestriction with PvII")
print(restrict_DNA(seq, 'PvII'))
print("\nRestriction with TaqI")
R = restrict_DNA(seq, 'TaqI')
print(R)
print("\nSecond restriction with EcoRI")
print(restrict_DNA_sequences(R, 'EcoRI'))
print(restrict_DNA_sequences(R, 'MyEnzyme'))
```

```
Restriction with EcoRI
[('TACTACCTACG', 'AATTCAAATGAATTCCCTTAAGAATTCATTCGATCGATTAAAGAGCTGATTAGAATATCCAGCTG'),
 ↪ ('TACTACCTACGAATTCAAATG', 'AATTCCCTTAAGAATTCATTCGATCGATTAAAGAGCTGATTAGAATATCCAGCTG
↪'), ('TACTACCTACGAATTCAAATGAATTCCCTTAAG',
↪'AATTCATTCGATCGATTAAAGAGCTGATTAGAATATCCAGCTG')]

Restriction with PvII
[('TACTACCTACGAATTCAAATGAATTCCCTTAAGAATTCATTCGATCGATTAAAGAGCTGATTAGAATATCCAG', 'CTG')]

Restriction with TaqI
[('TACTACCTACGAATTCAAATGAATTCCCTTAAGAATTCATT', 'CGATCGATTAAAGAGCTGATTAGAATATCCAGCTG'),
 ↪ ('TACTACCTACGAATTCAAATGAATTCCCTTAAGAATTCATTCGAT', 'CGATTAAAGAGCTGATTAGAATATCCAGCTG
↪')]

Second restriction with EcoRI
[[('TACTACCTACG', 'AATTCAAATGAATTCCCTTAAGAATTCATT'), ('TACTACCTACGAATTCAAATG',
↪'AATTCCCTTAAGAATTCATT'), ('TACTACCTACGAATTCAAATGAATTCCCTTAAG', 'AATTCATT')], [(
↪'CGATCGATTAAAGAGCTGATTAGAATATCCAGCTG', '')], [('TACTACCTACG',
↪'AATTCAAATGAATTCCCTTAAGAATTCATTCGAT'), ('TACTACCTACGAATTCAAATG',
↪'AATTCCCTTAAGAATTCATTCGAT'), ('TACTACCTACGAATTCAAATGAATTCCCTTAAG', 'AATTCATTCGAT')],
 ↪ [('CGATTAAAGAGCTGATTAGAATATCCAGCTG', '')]]
Enzyme 'MyEnzyme does not exist
None
```

5. Blast[45] is a well known tool to perform sequence alignment between a pool of query sequences and a pool of subject sequences. Among the other formats, it can produce an text output that is tab separated (`\t`) capturing user specified output. Comments in the file are written in lines starting with an hash key ("#"). A sample blast output file is blast_sample.tsv, please download it and spend some time to look at it. The meaning of all columns is specified in the file header:

[45] https://www.ncbi.nlm.nih.gov/pubmed/2231712

```
# Fields: query id, subject id, query length, % identity, alignment length,
identical, gap opens, q. start, q. end, s. start, s. end, evalue
```

Write a python program with:

1. A function (*readBlast*) that reads in the blast .tsv file (storing each line in a list) ignoring comment lines;
2. A function (*filterBlast*) that gets a string in input representing a blast alignment and filters it according to the user specified parameters below. It should return True if all filters are passed, false otherwise:
    1. % identity > identity_threshold (default 0%)
    2. evalue < evalue_thrshold (default 0.5)
    3. alignment length / query length > align_threshold (default 0)

The program should report how many entries out of the total passed the filter.

Test several combinations of filters like:

```
1. identity_theshold = 97, all others default
2. evalue_threshold = 0.5, all others default
3. align_threshold = 0.9, all others default
4. align_threshold = 0.9, identity_threshold = 97
```

Show/Hide Solution

[46]:

```
def filterBlast(seq, id_thr = 0, eval_thr = 0.5, al_thr = 0):
    """filters an alignment represented as a string. Returns True if filters OK,␣
↪False otherwise"""
    infos = seq.split("\t")
    return float(infos[3]) > id_thr and float(infos[-1]) < eval_thr and␣
↪float(int(infos[4]))/int(infos[2]) > al_thr

def readBlast(inFile):
    """Reads the blast .tsv file and returns a list"""
    fh = open(inFile, "r")
    alignments = []
    for line in fh:
        if not line.startswith("#"):
            alignments.append(line)
    return alignments

def filterAndCount(alignList, id_thr = 0, eval_thr = 0.5, al_thr = 0):
    """filters alignments and counts the ones passing the filter"""
    passed = [x for x in alignList if filterBlast(x, id_thr, eval_thr, al_thr)]
    print("{} out of {} aligns passed filter (id_thr:{},eval_thr:{},al_thr:{})".
↪format(
        len(passed), len(alignList), id_thr,eval_thr,al_thr))

blastF = "file_samples/blast_sample.tsv"

myAligns = readBlast(blastF)

filterAndCount(myAligns, id_thr=97)
filterAndCount(myAligns, eval_thr=1)
filterAndCount(myAligns, al_thr=0.9)
filterAndCount(myAligns, al_thr=0.9, id_thr=97)
```

(continues on next page)

(continued from previous page)

```
8390 out of 90790 aligns passed filter (id_thr:97,eval_thr:0.5,al_thr:0)
90780 out of 90790 aligns passed filter (id_thr:0,eval_thr:1,al_thr:0)
27711 out of 90790 aligns passed filter (id_thr:0,eval_thr:0.5,al_thr:0.9)
804 out of 90790 aligns passed filter (id_thr:97,eval_thr:0.5,al_thr:0.9)
```

6. Fasta is a format[46] representing nucleotide or peptide sequences. Each entry of a fasta file starts with a ">" followed by the identifier of the entry (the header of the sequence). All the lines following a header represent the sequence belonging to that identifier. Here you can find an example of fasta file: contigs82.fasta. Please download it and have a look at its content.

Write a python program that reads a fasta file and prints to screen the identifier of the sequence and the frequency of all the characters in the sequence (note that sequences might contain all IUPAC codes in case of SNPs). Hint: use a dictionary.

Sample output:

```
MDC020656.85 :
     N has freq 0.052
     T has freq 0.273
     A has freq 0.308
     G has freq 0.203
     C has freq 0.163
MDC001115.177 :
     N has freq 0.018
     G has freq 0.157
     A has freq 0.302
     T has freq 0.316
     C has freq 0.207
     ...
```

Show/Hide Solution

[36]:
```
def countFrequency(seq):
    """gets a sequence in input and returns
    a dictionary with bases as keys and frequence as value"""
    bases = {}
    for b in seq:
        if b not in bases:
            bases[b] = 1
        else:
            bases[b] += 1

    for b in bases:
        bases[b] = bases[b]/len(seq)
    return bases

def printData(ident, freqDict):
    """get the identifier and a dictionary with all frequencies
    and prints both information on the screen"""
    print(ident,":")
    for f in freqDict:
        print("\t {} has freq {:.3f}".format(f,freqDict[f]))

def processFasta(file):
```

(continues on next page)

[46] https://en.wikipedia.org/wiki/FASTA_format

(continued from previous page)

```
    header = ""
    seq = ""
    with open(file, "r") as f:
        for line in f:
            line = line.strip()
            if line.startswith(">"):
                if len(header) == 0:
                    #first entry:
                    header = line[1:]
                else:
                    #this is a new entry
                    freqData = countFrequency(seq)
                    printData(header,freqData)
                    seq = ""
                    header = line[1:]
            else:
                seq +=line
    #processing the final entry
    freqData = countFrequency(seq)
    printData(header,freqData)


inFasta = "file_samples/contigs82.fasta"
processFasta(inFasta)
```

```
MDC020656.85 :
         G has freq 0.203
         A has freq 0.308
         T has freq 0.273
         C has freq 0.163
         N has freq 0.052
MDC001115.177 :
         T has freq 0.316
         G has freq 0.157
         A has freq 0.302
         C has freq 0.207
         N has freq 0.018
MDC013284.379 :
         T has freq 0.321
         A has freq 0.309
         C has freq 0.171
         G has freq 0.194
         N has freq 0.005
MDC018185.243 :
         C has freq 0.204
         A has freq 0.307
         T has freq 0.304
         G has freq 0.181
         N has freq 0.001
         R has freq 0.001
         Y has freq 0.001
         S has freq 0.000
         K has freq 0.000
         M has freq 0.000
         W has freq 0.000
MDC018185.241 :
         A has freq 0.335
```

(continues on next page)

(continued from previous page)

```
        C has freq 0.168
        G has freq 0.183
        T has freq 0.304
        N has freq 0.006
        M has freq 0.001
        R has freq 0.001
        Y has freq 0.002
        S has freq 0.000
        K has freq 0.001
        W has freq 0.000
MDC004527.213 :
        C has freq 0.273
        T has freq 0.332
        G has freq 0.155
        A has freq 0.227
        N has freq 0.013
MDC003661.174 :
        A has freq 0.246
        G has freq 0.232
        T has freq 0.301
        C has freq 0.213
        N has freq 0.007
MDC012176.157 :
        A has freq 0.343
        T has freq 0.253
        G has freq 0.163
        C has freq 0.240
        N has freq 0.001
MDC001204.812 :
        T has freq 0.298
        A has freq 0.293
        G has freq 0.187
        C has freq 0.221
        N has freq 0.000
MDC001204.810 :
        A has freq 0.297
        T has freq 0.340
        G has freq 0.164
        C has freq 0.182
        N has freq 0.017
MDC004389.256 :
        G has freq 0.207
        A has freq 0.285
        C has freq 0.277
        T has freq 0.203
        N has freq 0.028
MDC024257.15 :
        C has freq 0.144
        T has freq 0.331
        A has freq 0.319
        G has freq 0.184
        N has freq 0.022
MDC018297.229 :
        A has freq 0.275
        G has freq 0.209
        T has freq 0.305
        C has freq 0.181
```

(continues on next page)

(continued from previous page)

```
        N has freq 0.031
MDC001802.364 :
        A has freq 0.252
        T has freq 0.328
        G has freq 0.180
        C has freq 0.203
        N has freq 0.037
MDC016621.241 :
        A has freq 0.302
        G has freq 0.168
        C has freq 0.206
        T has freq 0.262
        N has freq 0.063
MDC014057.243 :
        C has freq 0.263
        A has freq 0.291
        T has freq 0.212
        G has freq 0.217
        N has freq 0.016
MDC021015.302 :
        G has freq 0.251
        C has freq 0.143
        A has freq 0.302
        T has freq 0.223
        N has freq 0.081
MDC018185.242 :
        A has freq 0.316
        T has freq 0.289
        G has freq 0.189
        C has freq 0.200
        N has freq 0.000
        R has freq 0.002
        Y has freq 0.001
        K has freq 0.000
        M has freq 0.001
        W has freq 0.000
        S has freq 0.000
MDC051782.000 :
        C has freq 0.180
        T has freq 0.306
        G has freq 0.177
        A has freq 0.285
        Y has freq 0.002
        K has freq 0.001
        W has freq 0.001
        R has freq 0.002
        S has freq 0.001
        M has freq 0.000
        N has freq 0.044
MDC017187.314 :
        A has freq 0.289
        T has freq 0.276
        C has freq 0.206
        G has freq 0.225
        N has freq 0.003
MDC017187.311 :
        A has freq 0.248
```

(continues on next page)

(continued from previous page)

```
         G has freq 0.185
         T has freq 0.315
         C has freq 0.179
         N has freq 0.068
         W has freq 0.001
         R has freq 0.002
         Y has freq 0.001
         S has freq 0.000
         M has freq 0.000
         K has freq 0.000
MDC012865.410 :
         G has freq 0.218
         A has freq 0.289
         C has freq 0.162
         T has freq 0.328
         N has freq 0.003
MDC000427.83 :
         A has freq 0.392
         G has freq 0.159
         T has freq 0.274
         C has freq 0.165
         N has freq 0.010
MDC017187.319 :
         G has freq 0.200
         C has freq 0.181
         A has freq 0.282
         T has freq 0.331
         N has freq 0.006
MDC017187.318 :
         T has freq 0.305
         C has freq 0.187
         A has freq 0.275
         G has freq 0.232
         K has freq 0.000
         N has freq 0.001
MDC004081.319 :
         A has freq 0.172
         N has freq 0.428
         T has freq 0.197
         C has freq 0.106
         G has freq 0.098
MDC021913.275 :
         G has freq 0.143
         C has freq 0.155
         A has freq 0.199
         T has freq 0.169
         N has freq 0.334
MDC015147.205 :
         A has freq 0.298
         G has freq 0.196
         T has freq 0.350
         C has freq 0.154
         N has freq 0.002
MDC000038.355 :
         C has freq 0.170
         A has freq 0.331
         T has freq 0.315
```

(continues on next page)

(continued from previous page)

```
        G has freq 0.158
        N has freq 0.025
        Y has freq 0.000
        M has freq 0.000
        R has freq 0.000
        S has freq 0.000
        K has freq 0.000
        W has freq 0.000
MDC016032.95 :
        T has freq 0.286
        C has freq 0.142
        G has freq 0.222
        A has freq 0.342
        N has freq 0.003
        Y has freq 0.001
        S has freq 0.001
        K has freq 0.002
        R has freq 0.002
        W has freq 0.001
MDC052568.000 :
        G has freq 0.178
        A has freq 0.300
        T has freq 0.302
        C has freq 0.207
        Y has freq 0.001
        W has freq 0.001
        R has freq 0.002
        S has freq 0.001
        M has freq 0.001
        K has freq 0.000
        N has freq 0.008
MDC008119.414 :
        T has freq 0.316
        G has freq 0.229
        C has freq 0.245
        A has freq 0.209
        N has freq 0.001
MDC026201.7 :
        G has freq 0.191
        A has freq 0.313
        C has freq 0.247
        T has freq 0.250
MDC003995.601 :
        T has freq 0.334
        A has freq 0.323
        C has freq 0.166
        G has freq 0.167
        N has freq 0.006
        M has freq 0.001
        K has freq 0.000
        Y has freq 0.001
        W has freq 0.001
        S has freq 0.000
        R has freq 0.001
MDC009211.561 :
        C has freq 0.237
        G has freq 0.171
```

(continues on next page)

(continued from previous page)

```
         T has freq 0.286
         A has freq 0.306
         N has freq 0.001
MDC009211.567 :
         T has freq 0.327
         C has freq 0.167
         G has freq 0.181
         A has freq 0.318
         N has freq 0.006
MDC054294.001 :
         A has freq 0.303
         T has freq 0.318
         G has freq 0.186
         C has freq 0.186
         K has freq 0.001
         M has freq 0.001
         W has freq 0.001
         Y has freq 0.002
         R has freq 0.002
         S has freq 0.000
MDC004364.265 :
         A has freq 0.323
         T has freq 0.302
         G has freq 0.198
         C has freq 0.170
         N has freq 0.007
         R has freq 0.001
         Y has freq 0.001
         K has freq 0.000
         S has freq 0.000
         W has freq 0.000
MDC002360.219 :
         G has freq 0.173
         A has freq 0.316
         T has freq 0.303
         C has freq 0.204
         N has freq 0.003
MDC003408.117 :
         A has freq 0.329
         T has freq 0.247
         C has freq 0.215
         G has freq 0.191
         N has freq 0.017
MDC015155.172 :
         C has freq 0.202
         T has freq 0.277
         A has freq 0.272
         G has freq 0.247
         N has freq 0.003
MDC053310.000 :
         C has freq 0.167
         G has freq 0.185
         T has freq 0.310
         A has freq 0.330
         W has freq 0.001
         M has freq 0.001
         R has freq 0.003
```

(continues on next page)

(continued from previous page)

```
         K has freq 0.001
         Y has freq 0.002
         S has freq 0.000
MDC019140.398 :
         T has freq 0.302
         A has freq 0.352
         G has freq 0.178
         C has freq 0.163
         N has freq 0.005
MDC019140.399 :
         G has freq 0.187
         A has freq 0.301
         C has freq 0.165
         T has freq 0.346
         N has freq 0.001
MDC011390.337 :
         A has freq 0.251
         T has freq 0.298
         C has freq 0.228
         G has freq 0.177
         N has freq 0.046
         Y has freq 0.000
         R has freq 0.000
MDC007154.375 :
         T has freq 0.292
         C has freq 0.197
         A has freq 0.307
         G has freq 0.201
         N has freq 0.003
MDC006346.716 :
         A has freq 0.296
         G has freq 0.191
         T has freq 0.290
         C has freq 0.199
         N has freq 0.024
MDC010588.505 :
         C has freq 0.257
         G has freq 0.195
         A has freq 0.305
         T has freq 0.237
         N has freq 0.006
MDC002519.240 :
         T has freq 0.321
         A has freq 0.312
         G has freq 0.188
         C has freq 0.178
         R has freq 0.000
         N has freq 0.001
         Y has freq 0.000
         M has freq 0.000
MDC031322.5 :
         T has freq 0.299
         A has freq 0.342
         G has freq 0.202
         C has freq 0.149
         N has freq 0.002
         R has freq 0.002
```

(continues on next page)

(continued from previous page)

```
        K has freq 0.000
        M has freq 0.001
        Y has freq 0.002
        S has freq 0.000
        W has freq 0.000
MDC010588.502 :
        T has freq 0.311
        G has freq 0.215
        A has freq 0.282
        C has freq 0.184
        N has freq 0.009
MDC006346.711 :
        T has freq 0.295
        G has freq 0.187
        A has freq 0.321
        N has freq 0.006
        C has freq 0.191
        M has freq 0.000
MDC011551.182 :
        G has freq 0.190
        C has freq 0.189
        T has freq 0.297
        A has freq 0.313
        S has freq 0.000
        W has freq 0.000
        N has freq 0.007
        Y has freq 0.001
        R has freq 0.001
        M has freq 0.000
        K has freq 0.000
MDC002717.156 :
        T has freq 0.284
        G has freq 0.196
        A has freq 0.298
        C has freq 0.219
        N has freq 0.003
MDC006346.719 :
        A has freq 0.286
        C has freq 0.183
        G has freq 0.219
        T has freq 0.306
        N has freq 0.006
MDC007838.447 :
        G has freq 0.166
        C has freq 0.200
        T has freq 0.299
        A has freq 0.326
        N has freq 0.009
MDC007018.186 :
        A has freq 0.226
        T has freq 0.262
        C has freq 0.160
        G has freq 0.202
        N has freq 0.151
MDC017873.233 :
        G has freq 0.139
        A has freq 0.262
```

(continues on next page)

(continued from previous page)

```
        T has freq 0.231
        C has freq 0.133
        N has freq 0.235
MDC016296.138 :
        G has freq 0.188
        T has freq 0.346
        A has freq 0.277
        C has freq 0.172
        N has freq 0.016
MDC019067.226 :
        A has freq 0.293
        G has freq 0.181
        T has freq 0.376
        C has freq 0.144
        N has freq 0.007
MDC036568.1 :
        T has freq 0.348
        A has freq 0.331
        G has freq 0.161
        N has freq 0.006
        C has freq 0.154
MDC014019.318 :
        C has freq 0.249
        A has freq 0.221
        T has freq 0.319
        G has freq 0.209
        N has freq 0.002
MDC007995.528 :
        G has freq 0.181
        T has freq 0.340
        C has freq 0.166
        A has freq 0.305
        N has freq 0.009
MDC026961.60 :
        A has freq 0.300
        G has freq 0.206
        T has freq 0.288
        C has freq 0.201
        N has freq 0.005
MDC013443.168 :
        C has freq 0.203
        A has freq 0.289
        T has freq 0.330
        G has freq 0.173
        N has freq 0.005
MDC022800.298 :
        C has freq 0.131
        A has freq 0.323
        T has freq 0.334
        G has freq 0.203
        N has freq 0.008
MDC021558.159 :
        A has freq 0.298
        G has freq 0.190
        T has freq 0.334
        C has freq 0.176
        N has freq 0.002
```

(continues on next page)

(continued from previous page)

```
MDC002479.192 :
         T has freq 0.304
         C has freq 0.151
         A has freq 0.305
         G has freq 0.232
         N has freq 0.009
MDC010751.301 :
         A has freq 0.318
         G has freq 0.214
         T has freq 0.291
         C has freq 0.172
         N has freq 0.006
MDC000219.190 :
         A has freq 0.332
         G has freq 0.168
         T has freq 0.335
         C has freq 0.160
         N has freq 0.003
         Y has freq 0.001
         S has freq 0.000
         M has freq 0.000
         R has freq 0.000
MDC020963.161 :
         G has freq 0.148
         A has freq 0.287
         T has freq 0.315
         C has freq 0.248
         N has freq 0.002
MDC020963.162 :
         A has freq 0.309
         T has freq 0.265
         G has freq 0.207
         C has freq 0.214
         N has freq 0.005
         W has freq 0.000
         S has freq 0.000
         R has freq 0.000
MDC051637.000 :
         T has freq 0.328
         C has freq 0.175
         A has freq 0.307
         G has freq 0.169
         R has freq 0.003
         M has freq 0.001
         S has freq 0.000
         Y has freq 0.005
         N has freq 0.010
         W has freq 0.001
         K has freq 0.001
MDC005174.220 :
         A has freq 0.311
         C has freq 0.198
         T has freq 0.283
         G has freq 0.186
         N has freq 0.022
MDC040033.7 :
         C has freq 0.195
```

(continues on next page)

(continued from previous page)

```
         A has freq 0.297
         T has freq 0.301
         G has freq 0.204
         N has freq 0.003
MDC019674.147 :
         T has freq 0.366
         G has freq 0.167
         A has freq 0.324
         C has freq 0.137
         N has freq 0.004
         Y has freq 0.001
         S has freq 0.000
         W has freq 0.000
         R has freq 0.000
MDC010450.877 :
         A has freq 0.293
         T has freq 0.285
         C has freq 0.190
         G has freq 0.174
         N has freq 0.058
MDC007097.457 :
         G has freq 0.217
         T has freq 0.299
         C has freq 0.126
         A has freq 0.289
         N has freq 0.070
MDC016278.70 :
         A has freq 0.277
         T has freq 0.303
         G has freq 0.153
         C has freq 0.183
         N has freq 0.084
MDC013723.254 :
         C has freq 0.212
         A has freq 0.279
         T has freq 0.329
         G has freq 0.179
         N has freq 0.001
MDC002838.179 :
         C has freq 0.163
         A has freq 0.359
         T has freq 0.297
         G has freq 0.167
         N has freq 0.013
MDC009771.217 :
         T has freq 0.335
         C has freq 0.185
         A has freq 0.315
         G has freq 0.163
         N has freq 0.001
```

7. Write a python program that reads two files. The first is a one column text file (contig_ids.txt) with the identifiers of some contigs that are present in the second file, which is a fasta formatted file (contigs82.fasta). The program will write on a third, fasta formatted file (e.g. filtered_contigs.fasta) only those entries in *contigs82.fasta* having identifier in *contig_ids.txt*.

Show/Hide Solution

[37]:
```
def readIDS(f):
    """reads a one column file in and stores
    the ids in a dictionary that is returned at the end"""
    ret = dict()
    with open(f, "r") as file:
        for line in file:
            line = line.strip()
            if(line not in ret):
                ret[line] = 1
    return ret

def filterFasta(inF, outF, ids2keep):
    oF = open(outF, "w")

    outputME = False
    with open(inF, "r") as file:
        for line in file:
            line = line.strip()
            if(line.startswith(">")):
                #this is the header
                if(line[1:] in ids2keep):
                    oF.write(line +"\n")
                    outputME = True
                    print("Writing contig ", line[1:])
                else:
                    outputME = False
            else:
                if(outputME):
                    oF.write(line +"\n")

    oF.close()


idsFile = "file_samples/contig_ids.txt"
inFasta = "file_samples/contigs82.fasta"
outFasta = "file_samples/filtered_contigs.fasta"

ids = readIDS(idsFile)
filterFasta(inFasta,outFasta, ids)
```

```
Writing contig  MDC001115.177
Writing contig  MDC013284.379
Writing contig  MDC018185.243
Writing contig  MDC018185.241
Writing contig  MDC004527.213
Writing contig  MDC012176.157
Writing contig  MDC001204.810
Writing contig  MDC004389.256
Writing contig  MDC018297.229
Writing contig  MDC001802.364
Writing contig  MDC014057.243
Writing contig  MDC021015.302
Writing contig  MDC017187.314
Writing contig  MDC012865.410
Writing contig  MDC000427.83
Writing contig  MDC017187.319
Writing contig  MDC004364.265
Writing contig  MDC002360.219
```

(continues on next page)

(continued from previous page)

```
Writing contig  MDC015155.172
Writing contig  MDC019140.398
Writing contig  MDC019140.399
Writing contig  MDC011390.337
Writing contig  MDC007154.375
Writing contig  MDC010588.505
Writing contig  MDC002519.240
Writing contig  MDC006346.711
Writing contig  MDC011551.182
Writing contig  MDC002717.156
Writing contig  MDC006346.719
Writing contig  MDC007838.447
Writing contig  MDC007018.186
Writing contig  MDC017873.233
Writing contig  MDC016296.138
Writing contig  MDC019067.226
Writing contig  MDC014019.318
Writing contig  MDC007995.528
Writing contig  MDC026961.60
Writing contig  MDC020963.161
Writing contig  MDC005174.220
Writing contig  MDC040033.7
Writing contig  MDC019674.147
Writing contig  MDC010450.877
Writing contig  MDC007097.457
Writing contig  MDC016278.70
```

8. Write a python program that:

    1. reads the text file sample_text.txt and stores in a dictionary how many times each word appears (hint: the key is the word and the count is the value);
    2. prints to screen how many lines the file has and how many distinct words are in the file;
    3. writes to a text file (scientist_histo.csv) the histogram of the words in comma separated value format (i.e. word,count). Words must be sorted alphabetically;
    4. Finally, write a function that prints to screen (alphabetically) all the words that have a count higher than a threshold N and apply it with N = 5.

Show/Hide Solution

[38]:
```
def wordHisto(myText):
    """this function returns a dictionary
    with the count of each word in myText (separated by " " or "\n")
    """
    myDict = dict()
    tmp = myText.replace("\n"," ")

    for word in tmp.split(" "):
        if(word not in myDict):
            myDict[word] = 1
        else:
            myDict[word] += 1
    return myDict

def writeWordHisto(outF, data):
    """this function writes to outFile
```

(continues on next page)

(continued from previous page)

```
    the word histogram data contained in the dictionary data
    the output format is comma separated
    """
    fh = open(outF, "w")
    dictKeys = list(data.keys())
    dictKeys.sort()
    fh.write("#word,count\n")
    for k in dictKeys:
        curVal = data[k]
        myStr = k + "," + str(curVal)+"\n" #string to write in file
        fh.write(myStr)
    fh.close() #remember to close the file

def printWords(N, data):
    """prints the words in data that have a count higher than N"""
    dictKeys = list(data.keys())
    dictKeys.sort()
    for w in dictKeys:
        cnt = data[w]
        if(cnt > N):
            print("Word \"{}\" is present {} times".format( w, cnt))


file = "file_samples/sample_text.txt"
outFile = "file_samples/sample_text_histo.csv"
wholeText = ""
fh = open(file,"r")
wholeText = fh.read()
wholeText = wholeText.strip() #to remove the final newline character

fh.close()
print("The file {} has {} lines".format(file, wholeText.count("\n") +1 )) #n lines␣
↪have n-1 \n

#Let's do the job:
wordD = wordHisto(wholeText)
writeWordHisto(outFile, wordD)
print("The total number of distinct words is ", len(wordD))
printWords(5,wordD)
```

```
The file file_samples/sample_text.txt has 28 lines
The total number of distinct words is  308
Word "challenge" is present 6 times
Word "future" is present 22 times
Word "reply" is present 7 times
Word "shop" is present 6 times
Word "thunder" is present 11 times
Word "umbrella" is present 10 times
```

CHAPTER

EIGHT

PRACTICAL 7

In this practical we will keep practicing with functions and will see how to get input from the command line.

8.1 Slides

The slides of the introduction can be found here: Intro

8.2 Functions

Reminder. The basic definition of a function is:

```
def function_name(input) :
    #code implementing the function
    ...
    ...
    return return_value
```

Functions are defined with the **def** keyword that proceeds the *function_name* and then a list of parameters is passed in the brackets. A colon **:** is used to end the line holding the definition of the function. The code implementing the function is specified by using indentation. A function **might** or **might not** return a value. In the first case a **return** statement is used.

8.3 Getting input from the command line

To call a program `my_python_program.py` from command line, you just have to open a terminal (in Linux) or the command prompt (in Windows) and, assuming that python is present in the path, you can `cd` into the folder containing your python program, (eg. `cd C:\python\my_exercises\`) and just type in `python3 my_python_program.py` or `python my_python_program.py` In case of arguments to be passed by command line, one has to put them after the specification of the program name (eg. `python my_python_program.py parm1 param2 param3`).

Python provides the module **sys** to interact with the interpreter. In particular, **sys.argv** is a list representing all the arguments passed to the python script from the command line.

Consider the following code:

```
[ ]: import sys
"""Test input from command line in systest.py"""

if len(sys.argv) != 4: #note that this is the number of params +1!!!
    print("Dear user, I was expecting 3 params. You gave me ",len(sys.argv)-1)
    exit(1)
else:
    for i in range(0,len(sys.argv)):
        print("Param {}:{} ({})".format(i,sys.argv[i],type(sys.argv[i])))
```

Invoking the `systest.py` script from command line with the command `python3 exercises/systest.py 1st_param 2nd 3` will return:

```
Param 0: exercises/systest.py (<class 'str'>)
Param 1: 1st_param (<class 'str'>)
Param 2: 2nd (<class 'str'>)
Param 3: 3 (<class 'str'>)
```

Invoking the `systest.py` script from command line with the command `python3 exercises/systest.py 1st_param` will return:

```
Dear user, I was expecting three parameters. You gave me  1
```

Note that the parameter at index 0, `sys.argv[0]` holds the name of the script, and that all parameters are actually **strings** (and therefore need to be cast to numbers if we want to do mathematical operations on them).

Example: Write a script that takes two integers in input, i1 and i2, and computes the sum, difference, multiplication and division on them.

```
[ ]: import sys
"""Maths example with input from command line"""

if len(sys.argv) != 3:
    print("Dear user, I was expecting 2 params. You gave me ",len(sys.argv)-1)
    exit(1)
else:
    i1 = int(sys.argv[1])
    i2 = int(sys.argv[2])
    print("{} + {} = {}".format(i1,i2, i1 + i2))
    print("{} - {} = {}".format(i1,i2, i1 - i2))
    print("{} * {} = {}".format(i1,i2, i1 * i2))
    if i2 != 0:
        print("{} / {} = {}".format(i1,i2, i1 / i2))
    else:
        print("{} / {} = Infinite".format(i1,i2))
```

Which, depending on user input, should give something like:

```
biancol@bluhp:~/Google Drive/work/scripts$ python3.6 /tmp/test.py
Dear user, I was expecting 2 params. You gave me  0
biancol@bluhp:~/Google Drive/work/scripts$ python3.6 /tmp/test.py 75 32
75 + 32 = 107
75 - 32 = 43
75 * 32 = 2400
75 / 32 = 2.34375
biancol@bluhp:~/Google Drive/work/scripts$ python3.6 /tmp/test.py 75 0
75 + 0 = 75
75 - 0 = 75
75 * 0 = 0
75 / 0 = Infinite
biancol@bluhp:~/Google Drive/work/scripts$ python3.6 /tmp/test.py 75 t
Traceback (most recent call last):
  File "/tmp/test.py", line 9, in <module>
    i2 = int(sys.argv[2])
ValueError: invalid literal for int() with base 10: 't'
```

note that we need to check if the values given in input are actually numbers, otherwise the execution will crash (third example). This is easy in case of integers (`str.isdigit()`) but in case of floats it is more complex and might require Exception handling.

A more flexible and powerful way of getting input from command line makes use of the `Argparse` module[47].

8.4 Argparse

Argparse is a command line parsing module which deals with user specified parameters (positional arguments) and optional arguments.

Very briefly, the basic syntax of the `Argparse module` (for more information check the official documentation[48]) is the following.

1. Import the module:

```
import argparse
```

2. Define a argparse object:

```
parser = argparse.ArgumentParser(description="This is the description of the program")
```

note the parameter *description* that is a string to describe the program;

3. Add positional arguments:

```
parser.add_argument("arg_name", type = obj,
                    help = "Description of the parameter)
```

where `arg_name` is the name of the argument (which will be used to retrieve its value). The argument has type `obj` (the type will be automatically checked for us) and a description specified in the `help`string.

4. Add optional arguments:

[47] https://docs.python.org/3/howto/argparse.html
[48] https://docs.python.org/3/howto/argparse.html

```
parser.add_argument("-p", "--optional_arg", type = obj, default = def_val,
                    help = "Description of the parameter)
```

where `-p` is a short form of the parameter (and it is optional), `--optional_arg` is the extended name and it requires a value after it is specified, `type` is the data type of the parameter passed (e.g. str, int, float, ..), `default` is optional and gives a default value to the parameter. If not specified and no argument is passed, the argument will get the value "None". `Help` is again the description string.

5. Parse the arguments:

```
args = parser.parse_args()
```

the parser checks the arguments and stores their values in the `argparse` object that we called `args`.

6. Retrieve and process arguments:

```
myArgName = args.arg_name
myOptArg = args.optional_arg
```

now variables contain the values specified by the user and we can use them.

Let's see the example above again.

Example: Write a script that takes two integers in input, i1 and i2, and computes the sum, difference, multiplication and division on them.

[ ]:
```
import argparse
"""Maths example with input from command line"""
parser = argparse.ArgumentParser(description="""This script gets two integers  in␣
↪input
and performs some operations on them""")
parser.add_argument("i1", type=int,
                    help="The first integer")
parser.add_argument("i2", type=int,
                    help="The second integer")


args = parser.parse_args()

i1 = args.i1
i2 = args.i2
print("{} + {} = {}".format(i1,i2, i1 + i2))
print("{} - {} = {}".format(i1,i2, i1 - i2))
print("{} * {} = {}".format(i1,i2, i1 * i2))
if i2 != 0:
    print("{} / {} = {}".format(i1,i2, i1 / i2))
else:
    print("{} / {} = Infinite".format(i1,i2))
```

That returns the following:

```
biancol@bludell:~/work/courses/QCBsciprolab2020$ python exercises/systest_argparse.py
usage: systest_argparse.py [-h] i1 i2
systest_argparse.py: error: too few arguments
biancol@bludell:~/work/courses/QCBsciprolab2020$ python exercises/systest_argparse.py --help
usage: systest_argparse.py [-h] i1 i2

This script gets two integers in input and performs some operations on them

positional arguments:
  i1          The first integer
  i2          The second integer

optional arguments:
  -h, --help  show this help message and exit
biancol@bludell:~/work/courses/QCBsciprolab2020$ python exercises/systest_argparse.py 32 0
32 + 0 = 32
32 - 0 = 32
32 * 0 = 0
32 / 0 = Infinite
biancol@bludell:~/work/courses/QCBsciprolab2020$ python exercises/systest_argparse.py 32 t
usage: systest_argparse.py [-h] i1 i2
systest_argparse.py: error: argument i2: invalid int value: 't'
biancol@bludell:~/work/courses/QCBsciprolab2020$
```

Note that we did not have to check the types of the inputs (i.e. the last time we run the script) but this is automatically done by argparse.

Example: Let's write a program that gets a string (S) and an integer (N) in input and prints the string repeated N times. Three optional parameters are specified: verbosity (-v) to make the software print a more descriptive output, separator (-s) to separate each copy of the string (defaults to " ") and trailpoints (-p) to add several "." at the end of the string (defaults to 1).

[ ]:
```
import argparse
parser = argparse.ArgumentParser(description="""This script gets a string
                                 and an integer and repeats the string N times""")
parser.add_argument("string", type=str,
                    help="The string to be repeated")
parser.add_argument("N", type=int,
                    help="The number of times to repeat the string")

parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")

parser.add_argument("-p", "--trailpoints", type = int, default = 1,
                    help="Adds these many trailing points")
parser.add_argument("-s", "--separator", type = str, default = " ",
                    help="The separator between repeated strings")

args = parser.parse_args()

mySTR = args.string + args.separator
trailP = "." * args.trailpoints
answer = mySTR * args.N

answer = answer[:-len(args.separator)] + trailP #to remove the last separator

if args.verbose:
    print("the string {} repeated {} is:".format(args.str, args.N, answer))
else:
    print(answer)
```

(continues on next page)

(continued from previous page)

Executing the program from command line without parameters gives the message:

```
biancol@bluhp:~/Google Drive/work$ python3.6 /tmp/testargparse.py
usage: testargparse.py [-h] [-v] [-p TRAILPOINTS] [-s SEPARATOR] string N
testargparse.py: error: the following arguments are required: string, N
biancol@bluhp:~/Google Drive/work$
```

Calling it with the `-h` flag:

```
biancol@bluhp:~/Google Drive/work$ python3.6 /tmp/testargparse.py -h
usage: testargparse.py [-h] [-v] [-p TRAILPOINTS] [-s SEPARATOR] string N

This script gets a string and an integer and repeats the string N times

positional arguments:
  string                The string to be repeated
  N                     The number of time to repeated the string

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         increase output verbosity
  -p TRAILPOINTS, --trailpoints TRAILPOINTS
                        Adds these many trailing points
  -s SEPARATOR, --separator SEPARATOR
                        The separator between repeated strings
biancol@bluhp:~/Google Drive/work$
```

With the positional arguments `"ciao a tutti"` and `3`:

```
biancol@bluhp:~/Google Drive/work$ python3.6 /tmp/testargparse.py "ciao a tutti" 3
ciao a tutti ciao a tutti ciao a tutti.
biancol@bluhp:~/Google Drive/work$
```

With the positional arguments `"ciao a tutti"` and `3`, and with the optional parameters `-s "___" -p 3 -v`

```
biancol@bluhp:~/Google Drive/work$ python3.6 /tmp/testargparse.py "ciao a tutti" 3 -s "___" -p 3 -v
the string ciao a tutti repeated 3 times is: ciao a tutti___ciao a tutti___ciao a tutti...
biancol@bluhp:~/Google Drive/work$
```

Example: Let's write a program that reads and prints to screen a text file specified by the user. Optionally, the file might be compressed with gzip to save space. The user should be able to read also gzipped files. Hint: use the module gzip which is very similar to the standard file management method (more info here[49]). You can find a text file here textFile.txt and its gzipped version here text.gz:

```
[ ]: import argparse
import gzip

parser = argparse.ArgumentParser(description="""Reads and prints a text file""")
parser.add_argument("filename", type=str, help="The file name")
parser.add_argument("-z", "--gzipped", action="store_true",
                    help="If set, input file is assumed gzipped")
```

(continues on next page)

[49] https://docs.python.org/3/library/gzip.html?highlight=gzip#module-gzip

(continued from previous page)

```
args = parser.parse_args()
inputFile = args.filename
fh = ""
if args.gzipped:
    fh = gzip.open(inputFile, "rt")
else:
    fh = open(inputFile, "r")

for line in fh:
    line = line.strip("\n")
    print(line)

fh.close()
```

The output:

```
biancol@bluhp:~/Google Drive/work/courses/QCBsciprolab$ python3.6 examples/readFile_gz.py file_samples/textFile.txt
Hi everybody,
This is my first file
and it contains a total of
four lines!
biancol@bluhp:~/Google Drive/work/courses/QCBsciprolab$ python3.6 examples/readFile_gz.py file_samples/textFile.gz -z
Hi everybody,
This is my first file
and it contains a total of
four lines!
```

Example: Let's write a program that reads the content of a file and prints to screen some stats like the number of lines, the number of characters and maximum number of characters in one line. Optionally (if flag -v is set) it should print the content of the file. You can find a text file here textFile.txt:

[ ]:
```
import argparse


def readText(f):
    """reads the file and returns a list with
    each line as separate element"""
    myF = open(f, "r")
    ret = myF.readlines() #careful with big files!
    return ret


def computeStats(fileList):
    """returns a tuple (num.lines, num.characters,max_char.line)"""
    num_lines = len(fileList)
    lines_len = [len(x.replace("\n", "")) for x in fileList]
    num_char = sum(lines_len)
    max_char = max(lines_len)
    return (num_lines, num_char, max_char)


parser = argparse.ArgumentParser(description="Computes file stats")
parser.add_argument("inputFile", type=str, help="The input file")
parser.add_argument(
    "-v", "--verbose", action="store_true", help="if set, prints the file content")

args = parser.parse_args()
```

(continues on next page)

(continued from previous page)

```
inFile = args.inputFile

lines = readText(inFile)
stats = computeStats(lines)
if args.verbose:
    print("File content:\n{}\n".format("".join(lines)))
print(
    "Stats:\nN.lines:{}\nN.chars:{}\nMax. char in line:{}".format(
        stats[0], stats[1], stats[2]))
```

Output with -v flag:

```
biancol@bluhp:~/Google Drive/work/courses/QCBsciprolab$ python3 fileStats.py file_samples/textFile.txt -v
File content:
Hi everybody,
This is my first file
and it contains a total of
four lines!

Stats:
N.lines:4
N.chars:71
Max. char in line:26
```

Output without -v flag:

```
biancol@bluhp:~/Google Drive/work/courses/QCBsciprolab$ python3 fileStats.py file_samples/textFile.txt
Stats:
N.lines:4
N.chars:71
Max. char in line:26
```

8.5 Exercises

1. Modify the program of Exercise 6 of Practical 6 in order to allow users to specify the input and output files from command line. Then test it with the provided files. The text of the exercise follows:

Write a python program that reads two files. The first is a one column text file (contig_ids.txt) with the identifiers of some contigs that are present in the second file, which is a fasta formatted file (contigs82.fasta). The program will write on a third, fasta formatted file (e.g. filtered_contigs.fasta) only those entries in *contigs82.fasta* having identifier in *contig_ids.txt*.

Show/Hide Solution

[ ]:
```
"""exercises/filterFasta.py"""

import argparse

def readIDS(f):
    """reads a one column file in and stores
    the ids in a dictionary that is returned at the end"""
    ret = dict()
    with open(f, "r") as file:
        for line in file:
            line = line.strip()
            if(line not in ret):
                ret[line] = 1 #Important. It is like: True
    return ret
```

(continues on next page)

(continued from previous page)

```
def filterFasta(inF, outF, ids2keep):
    oF = open(outF, "w")

    outputME = False
    with open(inF, "r") as file:
        for line in file:
            line = line.strip()
            if line.startswith(">"):
                #this is the header
                if ids2keep.get(line[1:],False):
                    oF.write(line +"\n")
                    outputME = True
                    print("Writing contig ", line[1:])
                else:
                    outputME = False
            else:
                if outputME:
                    oF.write(line +"\n")

    oF.close()


parser = argparse.ArgumentParser(description="Filters a fasta file")
parser.add_argument("inputFasta", type = str, help = "The input fasta file")
parser.add_argument("inputIDS", type = str, help = "The IDS to keep")
parser.add_argument("outputFasta", type = str,
                    help = "The output fasta file with filtered entries")
args = parser.parse_args()
idsFile = args.inputIDS
inFasta = args.inputFasta
outFasta = args.outputFasta

ids = readIDS(idsFile)
filterFasta(inFasta,outFasta, ids)
```

2. Write a python script that takes in input a single-entry .fasta file (specified from the command line) of the amino-acidic sequence of a protein and prints off 1) the total number of aminoacids, 2) for each aminoacid, its count and percentage of the whole. Optionally, it the user specifies the flag "-S" (–search) followed by a string representing an aminoacid sequence, the program should count and print how many times that input sequence appears. Download the Sars-Cov-2 Spike Protein and test your script on it. *Please use functions*.

Show/Hide Solution

[ ]:
```
"""exercises/process_fasta.py"""

""" test example:
python3 process_fasta.py ../file_samples/P0DTC2.fasta.txt -S SSVL """

import argparse

parser = argparse.ArgumentParser(description="Parses a single-entry fasta file and␣
↪returns some stats.")
parser.add_argument("inputFasta", type = str, help = "The input fasta file")
parser.add_argument("-S", "--search", type = str, default = "",
                    help="The (optional) string to look for.")
```

(continues on next page)

(continued from previous page)

```
# Reads a fasta file input_file in
#and returns the tuple (header, sequence)
def read_sequence(input_file):
    sequence = ""
    hdr = ""
    inF = open(input_file)
    for line in inF:
        line = line.strip()
        if not line.startswith(">"):
            sequence += line
        else:
            hdr = line[1:]

    return hdr,sequence


# Gets a sequence seq and returns
# a dictionary with the counts of all elements.
# This function also prints off the counts (and %)
def count_chars(seq):
    char_dict = dict()
    L = len(seq)
    for c in seq:
        if char_dict.get(c, None) == None:
            char_dict[c] = 0
        char_dict[c] += 1

    print("The sequence has length: {}".format(L))
    for el in char_dict:
        print("{} is present {} times ({:.2f} %)".format(el, char_dict[el], 100 *␣
↪char_dict[el]/L))

    return char_dict

# Counts how many times search_s is in seq and returns an integer
def count_str(seq, search_s):
    return seq.count(search_s)


args = parser.parse_args()

inFasta = args.inputFasta

search_str = args.search


h,s = read_sequence(inFasta)
print(h)
print(s)
cnts = count_chars(s)

if len(search_str) > 0:
    C = count_str(s, search_str)
    print("Sequence '{}' is present {} times ".format(search_str, C))
```

The output should be like:

```
The sequence has length: 1273
M is present 14 times (1.10 %)
F is present 77 times (6.05 %)
V is present 97 times (7.62 %)
L is present 108 times (8.48 %)
P is present 58 times (4.56 %)
S is present 99 times (7.78 %)
Q is present 62 times (4.87 %)
C is present 40 times (3.14 %)
N is present 88 times (6.91 %)
T is present 97 times (7.62 %)
R is present 42 times (3.30 %)
A is present 79 times (6.21 %)
Y is present 54 times (4.24 %)
G is present 82 times (6.44 %)
D is present 62 times (4.87 %)
K is present 61 times (4.79 %)
H is present 17 times (1.34 %)
W is present 12 times (0.94 %)
I is present 76 times (5.97 %)
E is present 48 times (3.77 %)
Sequence 'SSVL' is present 2 times
```

3. Cytoscape[50] is a well known tool to perform network analysis. It is well integrated with several online databases housing for example protein-protein interactions like EBI's IntAct[51]. It is also able to read and write a very simple text file called `.sif` to represent interactions between the nodes of a network. Sif formatted files are tab separated (`\t`) and each line represents a connection between the nodes of the network. For example:

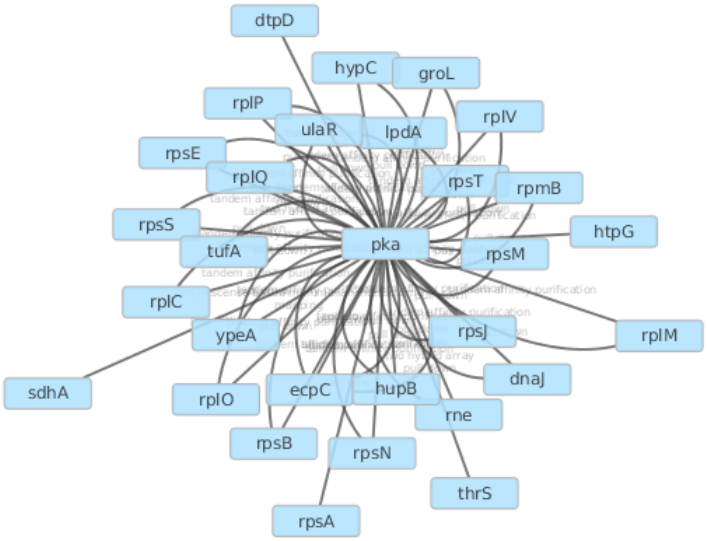
   ```
   node1 interaction1 node2
   node1 interaction2 node3
   node2 interaction1 node3
   ```

   represents two types of interactions between node1, node2 and node3. Normally nodes are represented as circles in a network (graph) and interactions as lines (that can be of different kinds) connecting nodes (edges). The following is an extract from the file pka.sif that has been downloaded by Cytoscape from the database IntAct and represents the interactions of the Protein Kinase A (PKA) of E.coli:

   ```
   P75742  EBI-9168813 P76594
   P21513  EBI-888473  P76594
   P21513  EBI-15543881    P76594
   ```

   the first and third columns represent proteins and the second is the interaction joining them. All the values are identifiers from the IntAct database. The cytoscape representation of the full set of interactions is:

[50] https://www.cytoscape.org
[51] https://www.ebi.ac.uk/intact/



Write a python script that reads in the .sif file (pka.sif is here but even better if any .sif file specified in input by the user) and stores the information in one (or more) suitable objects to be able to:

```
1. Print the interaction that is more present among the nodes;
2. Print the node that is connected to the highest number of
other nodes (no matter if on the left or right of the interaction);
```

Hint: you can store the information in a dictionary having the interaction as key and a list of tuples (node1,node2) as value. Although redundant, it is convenient to keep a list of unique nodes. **Note:** This will use more memory but it is acceptable for small examples as it allows to quickly answer the questions.

Optional: check what these ids refer to on the IntAct database[52].

Show/Hide Solution

[ ]:
```
"""exercises/parseSif.py"""

import argparse

def readSif(fn):
    """reads in the sif file fn and ouptputs a
    dictionary and a list:
    interDict : has "interaction" as key
    and [(node1,node2), (node3,node4)] as values
    nodes list : is the list of unique node names
    (both on the right or left hand of an interaction)
    """
    interDict = {}
    nodes = []
    with open(fn, "r") as myfile:
        for line in myfile:
            line = line.strip()
            [n1,inter,n2] = line.split("\t")
            if inter not in interDict:
                interDict[inter] = [(n1,n2)]
```

(continues on next page)

[52] https://www.ebi.ac.uk/intact/

(continued from previous page)

```
            else:
                interDict[inter].append((n1,n2))

            if n1 not in nodes:
                nodes.append(n1)
            if n2 not in nodes:
                nodes.append(n2)

    return interDict,nodes

def getMostPresentInteraction(iDict):
    """gets the interaction dictionary as defined above and returns the
    most present interaction (with its count). If more than one, all are
    returned comma separated"""
    mpInter = ""
    mpInterCount = 0
    for inter in iDict:

        cnt = len(iDict[inter])

        if mpInterCount < cnt:
            mpInterCount = cnt
            mpInter = inter
        elif mpInterCount == cnt:
            #mpInterCount = cnt #not necessary
            mpInter = mpInter + "," + inter
    return (mpInter,mpInterCount)

def getMostPresentNode(iDict, nodeList):
    """gets the most highly connected node (or nodes, comma separated)
    and returns it with its count
    iDict : the interaction dictionary seen above
    nodeList : the node of unique node names seen above
    """
    mostPresentNode = ""
    mostPresentCount = 0
    #NOTE: iDict : {"inter1" : [(n1,n2), (n2,n3),(n3,n1), (n1,n3)],
    #               "inter2" : [(n5,n1), (n1,n2)] }
    #
    for n in nodeList:
        #the number of times a node is present is the sum of its count
        #in all interactions
        #both as first or second member of the couple
        curCnt = 0
        for i in iDict:
            pairsContainingN = [x for x in iDict[i] if n in x]
            curCnt = curCnt + len(pairsContainingN)
        if(curCnt > mostPresentCount):
            mostPresentNode = n
            mostPresentCount = curCnt
        elif curCnt == mostPresentCount:
            postPresentNode += "," + n
            mostPresentCount = curCnt

    return (mostPresentNode,mostPresentCount)

parser = argparse.ArgumentParser(description="""Reads and processes a .sif file""")
```

(continues on next page)

(continued from previous page)

```
parser.add_argument("filename", type=str, help="The .sif file name")

args = parser.parse_args()
inputFile = args.filename
#inputFile = "file_samples/pka.sif"
interactions, nodeL = readSif(inputFile)
(mpI,mpICount) = getMostPresentInteraction(interactions)
print("The most present interaction(s): {}. Present {} times".format(mpI,mpICount))
(mpN, mpNCount) = getMostPresentNode(interactions,nodeL)
print("The most present node(s) {}. Present {} times".format(mpN,mpNCount))
```

4. Given a fasta[53] file like contigs82.fasta specified in input by a user, write a python script that counts, for each sequence, the number of times that a DNA or protein string specified in input appears.

If we run something like: python3 `find_stringInFasta.py contigs82.fasta TGCTCACAG`

the result should print lines like:

```
TGCTCACAG in MDC052568.000: 1 times
TGCTCACAG in MDC002479.192: 1 times
TGCTCACAG in MDC040033.7: 1 times
```

Modify the program so that it outputs also the list of all the indexes where the string appears in each sequence in the fasta file. Try to look for the following sequences:

```
TTTTCCTAGG
TGCTCCGAGCATGTGATAATCATTCCAAGCTCCAT
TAAACAT
GATTACA
```

Show/Hide Solution

[ ]:
```
"""exercises/find_stringInFasta.py"""


import argparse

def getIndexes(string1,string2):
    """checks if string2 is present in string1 and returns
    all the positions at which string2 occurs in string1"""
    ret = []
    ind = string1.find(string2)

    while (ind > -1 and ind < len(string1)):
        ret.append(ind)
        ind = string1.find(string2,ind + 1)

    return ret

def processFasta(file,testStr):
    """reads a fasta file entry by entry checks if the input string
       testStr is present in each sequence. Reporting how many times and where.
    """
    header = ""
    seq = ""
```

(continues on next page)

[53] https://en.wikipedia.org/wiki/FASTA_format

(continued from previous page)

```
    with open(file, "r") as f:
        for line in f:
            line = line.strip()
            if(line.startswith(">")):
                if(len(header) == 0 ):
                    #first entry:
                    header = line[1:]
                else:
                    #this is a new entry
                    indexes = getIndexes(seq,testStr)
                    if len(indexes) > 0:
                        print("{} in {}: {} times ({})".format(testStr,
                                                               header,
                                                               len(indexes),
                                                               indexes))
                    seq = ""
                    header = line[1:]
            else:
                seq +=line
    #processing the final entry
    indexes = getIndexes(seq,testStr)
    if len(indexes) > 0:
        print("{} in {}: {} times ({})".format(testStr,
                                               header,
                                               len(indexes),
                                               indexes))

parser = argparse.ArgumentParser(
    description="""Checks if a sequence is exactly contained in a fasta file"""
)
parser.add_argument("filename", type=str, help="The fasta file name")
parser.add_argument("query", type=str, help="The query string")

args = parser.parse_args()
#inFasta = args.filename
#testS = args.query
inFasta = "file_samples/contigs82.fasta"
testS = "TAAACAT"
processFasta(inFasta, testS)
```

5. The Fisher's dataset[54] regarding Petal and Sepal length and width in csv format can be found here. These are the measurements of the flowers of fifty plants each of the two species Iris setosa and Iris versicolor.

The header of the file is:

```
Species Number,Species Name,Petal width,Petal length,Sepal length,Sepal width
```

Write a python script that reads this file in input (feel free to hard-code the filename in the code) and computes the average petal length and width and sepal length and width for each of the three different Iris species. Print them to the screen alongside the number of elements.

Show/Hide Solution

[ ]:
```
def readCSV(f):
    """reads the csv dataset and returns a dictionary with
```

(continues on next page)

[54] http://onlinelibrary.wiley.com/doi/10.1111/j.1469-1809.1936.tb02137.x/abstract

(continued from previous page)

```
    species name as key and, as value, a dictionary with
    four keys : petalLen, sepalLen, petalWidth, sepalWidth
    """
    ret = dict()
    with open(f, "r") as file:
        for line in file:
            line = line.strip()
            if not line.startswith("Species Number"):
                data = line.split(",")
                speciesName = data[1]
                pWidth = int(data[2])
                pLen = int(data[3])
                sLen = int(data[4])
                sWidth = int(data[5])
                if(speciesName not in ret):
                    ret[speciesName] = {"petalLen" : [], "sepalLen" : [],
                                        "petalWidth" : [], "sepalWidth" : []
                                       }
                ret[speciesName]["petalLen"].append(pLen)
                ret[speciesName]["sepalLen"].append(sLen)
                ret[speciesName]["sepalWidth"].append(sWidth)
                ret[speciesName]["petalWidth"].append(pWidth)
    return ret

def printData(dataDict):
    for s in dataDict:
        avgPlen = sum(dataDict[s]["petalLen"])/len(dataDict[s]["petalLen"])
        avgPwid = sum(dataDict[s]["petalWidth"])/len(dataDict[s]["petalWidth"])
        avgSlen = sum(dataDict[s]["sepalLen"])/len(dataDict[s]["sepalLen"])
        avgSwid = sum(dataDict[s]["sepalWidth"])/len(dataDict[s]["sepalWidth"])
        print("Species {} has {} measurements:".format(s, len(dataDict[s]["petalLen
↪"])))
        print("\t petal length {}".format(avgPlen))
        print("\t petal width {}".format(avgPwid))
        print("\t sepal length {}".format(avgSlen))
        print("\t sepal width {}".format(avgSwid))



inFile = "file_samples/Fishers_Iris.csv"

data = readCSV(inFile)
printData(data)
```

CHAPTER

NINE

PRACTICAL 8

In this practical we will will practice Pandas.

9.1 Slides

The slides of the introduction can be found here: Intro

9.2 Libraries installation

First things first. Let's start off by installing the required libraries. In particular we will need three libraries. Try and see if they are already available by typing the following commands in the console or put them in a python script:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

if, upon execution, you do not get any error messages, you are sorted. Otherwise, you need to install them.

In **Linux** you can install the libraries by typing in a terminal `sudo pip3 install matplotlib`, `sudo pip3 install pandas` and `sudo pip3 install numpy` (or `sudo python3.X -m pip install matplotlib`, `sudo python3.X -m pip install pandas` and `sudo python3.X -m pip install numpy`), where X is your python version.

In **Windows** you can install the libraries by typing in the command prompt (to open it type `cmd` in the search box) `pip3 install matplotlib`, `pip3 install pandas` and `pip3 install numpy`. If you are using anaconda you need to run these commands from the *anaconda prompt*.

Please install them in this order (i.e. **matplotlib** first, then **pandas** and finally **numpy**). You might not need to install numpy as matplotlib requires it. Once done that, try to perform the above imports again and they should work this time around.

9.3 Pandas

Pandas (the name comes from *panel data*) is a very efficient library to deal with numerical tables and time series. It is a quite complex library and here we will only scratch the surface of it. You can find a lot of information including the documentation on the Pandas website[55].

In particular the library pandas provides two data structures: **Series** and **DataFrames**.

9.4 Series

Series are 1-dimensional structures (like lists) containing data. Series are characterized by two types of information: the **values** and the **index** (a list of labels associated to the data), therefore they are a bit like a list and a bit like a dictionary. The index is optional and can be added by the library if not specified.

9.4.1 How to define and access a Series

There are several ways to define a Series. We can specify both the values and the index explicitly, or through a dictionary, or let python add the default index for us. We can access the index with the **Series.index** method and the values with the **Series.values**.

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

print("Values and index explicitly defined")
#values and index explicitely defined
S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("The index:", S.index)
print("The values:", S.values)

print("------------------------\n")
print("From dictionary")
#from a dictionary
S1 = pd.Series({"one" : 1, "two" : 2, "ten": 10,
                "three" : 3, "four": 4, "forty" : 40})
print(S1)
print(S1.index)
print(S1.values)
print("------------------------\n")
print("Default index")
#index added by default
myData = [random.randint(0,10) for x in range(10)]
S2 = pd.Series(myData)

print(S2)
print(S2.index)
print(S2.values)

print("------------------------\n")
print("Same value repeated")
```

(continues on next page)

[55] http://pandas.pydata.org

(continued from previous page)

```
S3 = pd.Series(1.27, range(10))
print(S3)
print(S3.index)
print(S3.values)
```

```
Values and index explicitly defined
A    11
B    17
C    10
D     1
E    11
F     8
G    10
H    18
I    10
L     7
dtype: int64
The index: Index(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'L'], dtype='object')
The values: [11 17 10  1 11  8 10 18 10  7]
------------------------

From dictionary
one       1
two       2
ten      10
three     3
four      4
forty    40
dtype: int64
Index(['one', 'two', 'ten', 'three', 'four', 'forty'], dtype='object')
[ 1  2 10  3  4 40]
------------------------

Default index
0     5
1     4
2     9
3    10
4     7
5     7
6     2
7     5
8     2
9     7
dtype: int64
RangeIndex(start=0, stop=10, step=1)
[ 5  4  9 10  7  7  2  5  2  7]
------------------------

Same value repeated
0    1.27
1    1.27
2    1.27
3    1.27
4    1.27
5    1.27
6    1.27
```

(continues on next page)

(continued from previous page)

```
7    1.27
8    1.27
9    1.27
dtype: float64
RangeIndex(start=0, stop=10, step=1)
[1.27 1.27 1.27 1.27 1.27 1.27 1.27 1.27 1.27 1.27]
```

Data in a series can be accessed by using the **label** (i.e. the index) as in a dictionary or through its **position** as in a list. Slicing is also allowed both by **position** and **index**. In the latter case, we can do `Series[S:E]` with **S and E indexes**, both **S and E are included**.

It is also possible to retrieve some elements by passing a **list** of positions or indexes. **Head** and **tail** methods can also be used to retrieve the top or bottom N elements with **Series.head(N)** or **Series.tail(N)**.

Note: When the method returns more than one element, the return type is a Series.

---

[2]:
```
import pandas as pd
import random

#values and index explicitely defined
S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))
print(S)
print("")

print("Value at label \"A\":", S["A"])
print("Value at index 1:", S[1])
print("")

print("Slicing from 1 to 3:") #note 3 excluded
print(S[1:3])
print("")
print("Slicing from C to H:") #note H included!
print(S["C":"H"])
print("")

print("Retrieving from list:")
print(S[[1,3,5,7,9]])
print(S[["A","C","E","G"]])
print("")

print("Top 3")
print(S.head(3))
print("")
print("Bottom 3")
print(S.tail(3))
```

```
A     2
B    12
C     9
D    11
E     0
F    11
G    15
H    10
```

(continues on next page)

(continued from previous page)

```
I     5
L    16
dtype: int64

Value at label "A": 2
Value at index 1: 12

Slicing from 1 to 3:
B    12
C     9
dtype: int64

Slicing from C to H:
C     9
D    11
E     0
F    11
G    15
H    10
dtype: int64

Retrieving from list:
B    12
D    11
F    11
H    10
L    16
dtype: int64
A     2
C     9
E     0
G    15
dtype: int64

Top 3
A     2
B    12
C     9
dtype: int64

Bottom 3
H    10
I     5
L    16
dtype: int64
```

9.4.2 Operator broadcasting

Operations can automatically be broadcast to the entire Series. This is a quite cool feature and saves us from looping through the elements of the Series.

Example: Given a list of 10 integers and we want to divide them by 2. Without using pandas we would:

[3]:
```
import random

listS = [random.randint(0,20) for x in range(0,10)]

print(listS)

for el in range(0,len(listS)):
    listS[el] /=2  #compact of X = X / 2

print(listS)
```

```
[9, 9, 17, 2, 19, 5, 7, 8, 3, 19]
[4.5, 4.5, 8.5, 1.0, 9.5, 2.5, 3.5, 4.0, 1.5, 9.5]
```

With pandas instead:

[4]:
```
import pandas as pd
import random

S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("")
S1 = S / 2
print(S1)
```

```
A     7
B    13
C     0
D     2
E     3
F     0
G     1
H     5
I    17
L    16
dtype: int64

A    3.5
B    6.5
C    0.0
D    1.0
E    1.5
F    0.0
G    0.5
H    2.5
I    8.5
L    8.0
dtype: float64
```

9.4.3 Filtering

We can also apply boolean operators to obtain only the **sub-Series** with all the values satisfying a specific condition. This allows us to **filter** the Series.

Calling the boolean operator on the series alone (e.g. S > 10) will return a Series with True at the indexes where the condition is met, False at the others. Passing such a Series to a Series of the same length will return only the elements where the condition is True. Check the code below to see this in action.

[5]:
```
import pandas as pd
import random

S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("")
S1 = S>10
print(S1)
print("")
S2 = S[S > 10]
print(S2)
```

```
A     1
B    13
C     9
D    18
E    16
F    16
G    10
H    11
I    10
L     0
dtype: int64

A    False
B     True
C    False
D     True
E     True
F     True
G    False
H     True
I    False
L    False
dtype: bool

B    13
D    18
E    16
F    16
H    11
dtype: int64
```

9.4.4 Missing data

Operations involving Series might have to deal with missing data or non-valid values (both cases are represented as **NaN**, that is **not a number**). Operations are carried out by aligning the Series based on their **indexes**. Indexes not in common end up in NaN values. Although most of the operations that can be performed on series quite happily deal with NaNs, it is possible to **drop** NaN values or to **fill** them (i.e. replace their value with some other value). The sytax is:

```
Series.dropna()
```

or

```
Series.fillna(some_value)
```

Note that these operations do not modify the Series but rather return a new Series.

[6]:
```
import pandas as pd
import random

S = pd.Series([random.randint(0,10) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

S1 = pd.Series([random.randint(0,10) for x in range(0,8)],
              index = list("DEFGHAZY"))

print("The dimensions of these Series:", S.shape)
print("")
print(S1)
print("---- S + S1 ----")
Ssum = S + S1
print(Ssum)
print("---- Dropping NaNs ----")
print(Ssum.dropna())
print("---- Filling NaNs ----")
print(Ssum.fillna("my_value"))
```

```
The dimensions of these Series: (10,)

D    9
E    1
F    7
G    3
H    4
A    0
Z    6
Y    1
dtype: int64
---- S + S1 ----
A    10.0
B     NaN
C     NaN
D    18.0
E     2.0
F    12.0
G     7.0
H     4.0
I     NaN
L     NaN
Y     NaN
```

(continues on next page)

(continued from previous page)

```
Z     NaN
dtype: float64
---- Dropping NaNs ----
A    10.0
D    18.0
E     2.0
F    12.0
G     7.0
H     4.0
dtype: float64
---- Filling NaNs ----
A          10
B    my_value
C    my_value
D          18
E           2
F          12
G           7
H           4
I    my_value
L    my_value
Y    my_value
Z    my_value
dtype: object
```

9.4.5 Computing stats

Pandas offers several operators to compute stats on the data stored in a Series. These include basic stats like **min**, **max** (and relative indexes with **argmin** and **argmax**) **mean**, **std**, **quantile** (to get the quantiles). A description of the data can be obtained by using the method **describe** and the counts for each value can be obtained by **value_counts**. Other methods available are **sum** and **cumsum** (for the sum and cumulative sum of the elements), **autocorr** and **corr** (for autocorrelation and correlation) and many others. For a complete list check the Pandas reference[56].

Note: as said before, when these methods do not return a single value, they return a Series.

Example: Fill the previous Series with the mean values of the series rather than NaNs.

[7]:
```
Ssum = S + S1
print(Ssum)
print("---- Filling with avg value ----")
print(Ssum.fillna(Ssum.mean()))
print("Min:{} (index: {}) Max: {} (index: {})".format(Ssum.min(),
                                                      Ssum.argmin(),
                                                      Ssum.max(),
                                                      Ssum.argmax()))
```

```
A    10.0
B     NaN
C     NaN
D    18.0
E     2.0
F    12.0
G     7.0
```

(continues on next page)

[56] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html?highlight=series

(continued from previous page)

```
H     4.0
I     NaN
L     NaN
Y     NaN
Z     NaN
dtype: float64
---- Filling with avg value ----
A    10.000000
B     8.833333
C     8.833333
D    18.000000
E     2.000000
F    12.000000
G     7.000000
H     4.000000
I     8.833333
L     8.833333
Y     8.833333
Z     8.833333
dtype: float64
Min:2.0 (index: 4) Max: 18.0 (index: 3)
```

Let's see some operators introduced above in action.

[8]:
```
import pandas as pd
import random

S = pd.Series([random.randint(0,10) for x in range(0,10)],
              index = list("ABCDEFGHIL"))
print("The data:")
print(S)
print("")
print("Its description")
print(S.describe())
print("")
print("Specifying different quantiles:")
print(S.quantile([0.1,0.2,0.8,0.9]))
print("")
print("Histogram:")
print(S.value_counts())
print("")
print("The type is a Series:")
print(type(S.value_counts()))
print("Summing the values:")
print(S.sum())
print("")
print("The cumulative sum:")
print(S.cumsum())
```

```
The data:
A     5
B     4
C    10
D     3
E     4
F     1
G     4
```

(continues on next page)

(continued from previous page)

```
H     8
I     7
L     5
dtype: int64

Its description
count    10.000000
mean      5.100000
std       2.601282
min       1.000000
25%       4.000000
50%       4.500000
75%       6.500000
max      10.000000
dtype: float64

Specifying different quantiles:
0.1    2.8
0.2    3.8
0.8    7.2
0.9    8.2
dtype: float64

Histogram:
4     3
5     2
10    1
8     1
7     1
3     1
1     1
dtype: int64

The type is a Series:
<class 'pandas.core.series.Series'>
Summing the values:
51

The cumulative sum:
A     5
B     9
C    19
D    22
E    26
F    27
G    31
H    39
I    46
L    51
dtype: int64
```

Example:

Create two Series from the lists [2, 4, 6, 8, 10, 12, 13, 14, 15, 16], [1, 3, 5, 7, 9, 11, 13, 14, 15, 16] using the same index for both: ['9B47', '468B', 'B228', '3C52', 'AE2E', 'DFF6', 'C38B', '2CE5', '0325', '398F'].

Let's compare the distribution stats of the two Series (mean value, max, min, quantiles) and get the index and value of the positions where the two series are the same. Finally, let's get the sub-series where the first has a value higher than the

first quartile of the second and compute its stats.

[9]:
```
import pandas as pd

L1 = [2, 4, 6, 8, 10, 12, 13, 14, 15, 16]
L2 = [1, 3, 5, 7, 9, 11, 13, 14, 15, 16]
I = ['9B47', '468B', 'B228', '3C52', 'AE2E', 'DFF6', 'C38B', '2CE5', '0325', '398F']
L1Series = pd.Series(L1,index = I)
L2Series = pd.Series(L2, index = I)
#Let's describe the stats
print("Stats of L1Series")
print(L1Series.describe())
print("")
print("Stats of L2Series")
print(L2Series.describe())
print("")
#This is a Series with boolean values (True means the two Series where the same)
Leq = L1Series == L2Series
print("Equality series")
print(Leq)
print("")
#Get the subseries where both are the same
Lsub = L1Series[Leq]
print("Subseries of identicals")
print(Lsub)
print("")
#Get the values that are the same
print("Identical values:")
print(Lsub.values)
print("")
#Get the indexes where the two series are the same
print("Indexes of identical values:")
print(Lsub.index)
print("")
firstQuartile = L2Series.quantile(0.25)
print("The first quartile of L2Series:",firstQuartile)
print("")
#Get the subseries in which L1 is bigger than L2
Lbig = L1Series[L1Series > firstQuartile]
print("The subseries with L1 > L2")
print(Lbig)
```

```
Stats of L1Series
count    10.000000
mean     10.000000
std       4.830459
min       2.000000
25%       6.500000
50%      11.000000
75%      13.750000
max      16.000000
dtype: float64

Stats of L2Series
count    10.00000
mean      9.40000
std       5.25357
min       1.00000
```

(continues on next page)

(continued from previous page)

```
25%       5.50000
50%      10.00000
75%      13.75000
max      16.00000
dtype: float64

Equality series
9B47    False
468B    False
B228    False
3C52    False
AE2E    False
DFF6    False
C38B     True
2CE5     True
0325     True
398F     True
dtype: bool

Subseries of identicals
C38B    13
2CE5    14
0325    15
398F    16
dtype: int64

Identical values:
[13 14 15 16]

Indexes of identical values:
Index(['C38B', '2CE5', '0325', '398F'], dtype='object')

The first quartile of L2Series: 5.5

The subseries with L1 > L2
B228     6
3C52     8
AE2E    10
DFF6    12
C38B    13
2CE5    14
0325    15
398F    16
dtype: int64
```

9.5 Plotting data

Using python's matplotlib it is possible to plot data. The basic syntax is `Series.plot(kind = "type")` the parameter `kind` can be used to produce several types of plots (examples include **line**, **hist**, **pie**, **bar**, see here[57] for all possible choices). Note that **matplotlib** needs to be imported and the pyplot needs to be shown with **pyplot.show()** to display the plot.

Typically, the following syntax is used to import pyplot (remember also to import pandas):

[57] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.plot.html

```
import pandas as pd
import matplotlib.pyplot as plt
```

note the use of the alias `plt` for simplicity.

[10]:
```
import pandas as pd
import matplotlib.pyplot as plt
import random
S = pd.Series([random.randint(0,10) for x in range(0,10)],
              index = list("ABCDEFGHIL"))
print("The data:")
print(S)
S1 = pd.Series([random.randint(0,10) for x in range(0,100)])

S.plot()
plt.show()
plt.close()
S1.plot(kind = "hist")
plt.show()
plt.close()
S.plot(kind = "pie")
plt.show()
plt.close()
```

```
The data:
A     8
B     7
C     9
D     8
E     6
F     6
G     6
H     0
I     1
L    10
dtype: int64
```

```
<matplotlib.figure.Figure at 0x7f75584da240>
```

```
<matplotlib.figure.Figure at 0x7f75047b1160>
```

```
<matplotlib.figure.Figure at 0x7f75047c2748>
```

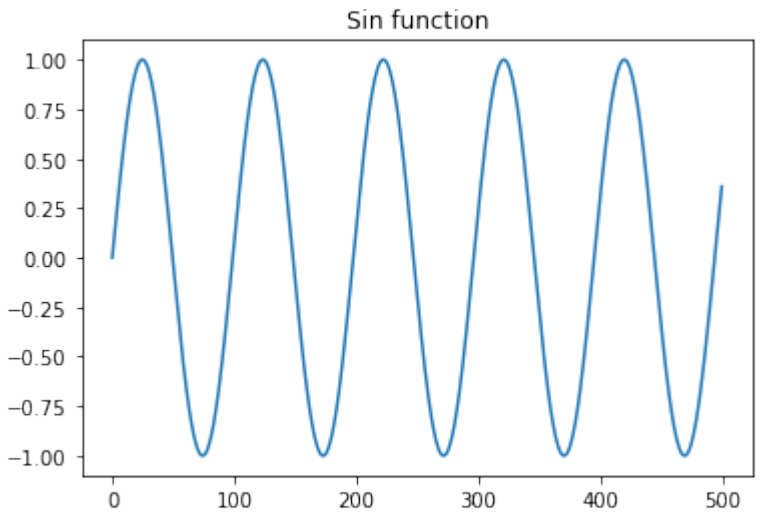
Example: Let's create a series representing the sin, cos and sqrt functions and plot them.

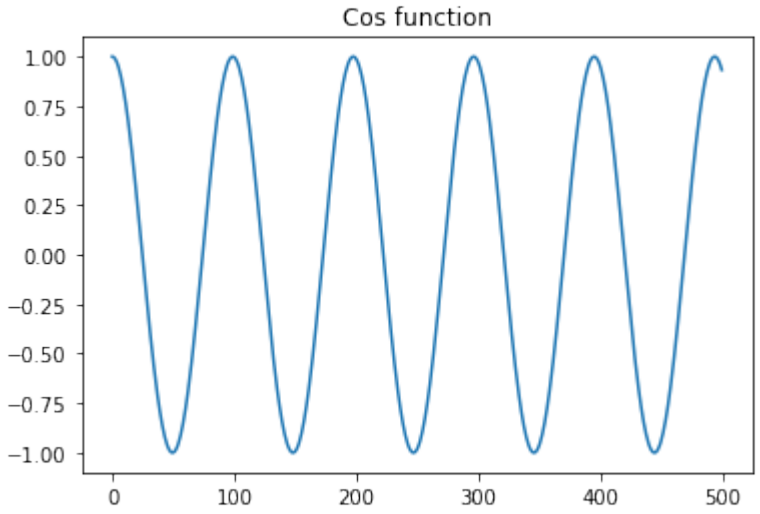
[39]:
```
import math
import matplotlib.pyplot as plt
import pandas as pd

x = [i/10 for i in range(0,500)]


y =  [math.sin(2*i/3.14 ) for i in x]
y1 = [math.cos(2*i/3.14 ) for i in x]
y2 = [math.sqrt(i) for i in x]
#print(x)

ySeries = pd.Series(y)
ySeries1 = pd.Series(y1)
```

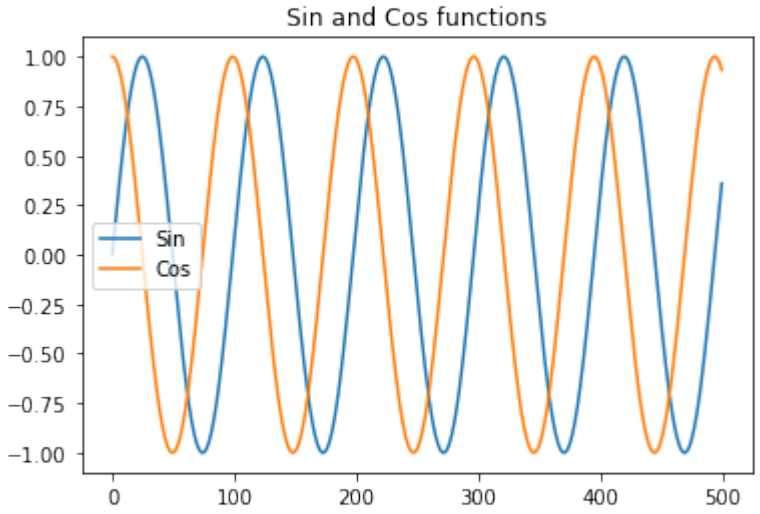
(continues on next page)

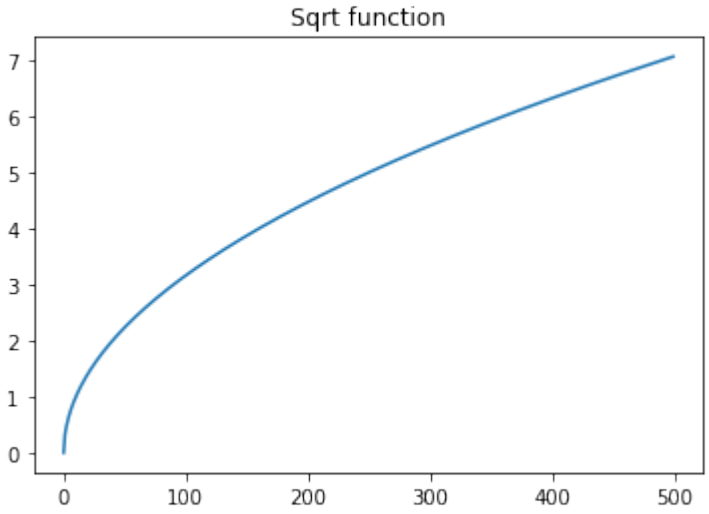
(continued from previous page)

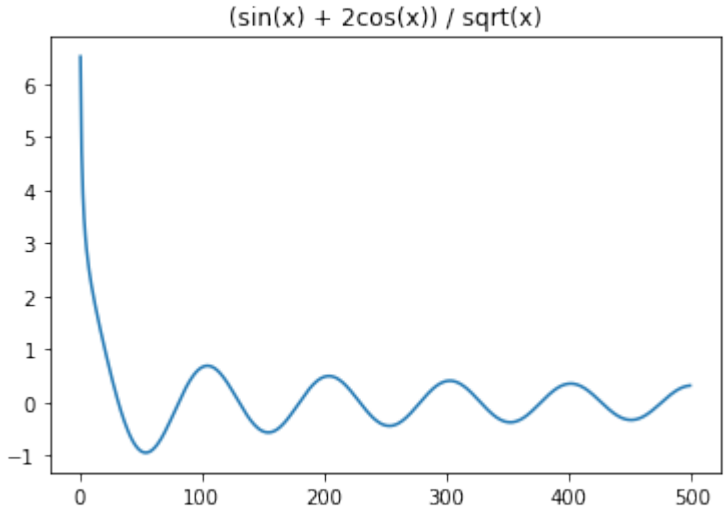
```
ySeries2 = pd.Series(y2)

ySeries.plot()
plt.title("Sin function")
plt.show()
plt.close()
ySeries1.plot()
plt.title("Cos function")
plt.show()
plt.close()
plt.title("Sin and Cos functions")
ySeries.plot()
ySeries1.plot()
plt.legend(["Sin", "Cos"])
plt.show()
plt.close()

ySeries2.plot()
plt.title("Sqrt function")
plt.show()
plt.close()
ySeries2 = (ySeries + 2*ySeries1)/ySeries2
ySeries2.plot()
plt.title("(sin(x) + 2cos(x)) / sqrt(x)")
plt.show()
```




Cos function
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
100
200
300
400
500


Sin and Cos functions
Sin
Cos
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
100
200
300
400
500


Sqrt function
7
6
5
4
3
2
1
0
0
100
200
300
400
500



9.6 Pandas DataFrames

DataFrames in pandas are the 2D analogous of Series. Dataframes are spreadsheet-like data structures with an ordered set of columns that can also be dishomogeneous. We can think about Dataframes as dictionaries of Series, each one representing a named column. Dataframes are described by an **index** that contains the labels of rows and a **columns** structure that holds the labels of the columns.

Note that the operation of extracting a column from a DataFrame returns a Series. Moreover, most (but not all!) of the operations that apply to Series also apply to DataFrames.

9.6.1 Define a DataFrame

There are several different ways to define a DataFrame. It is possible to create a DataFrame starting from a dictionary having Series as values. In this case, they **keys** of the dictionary are the **columns** of the DataFrame.

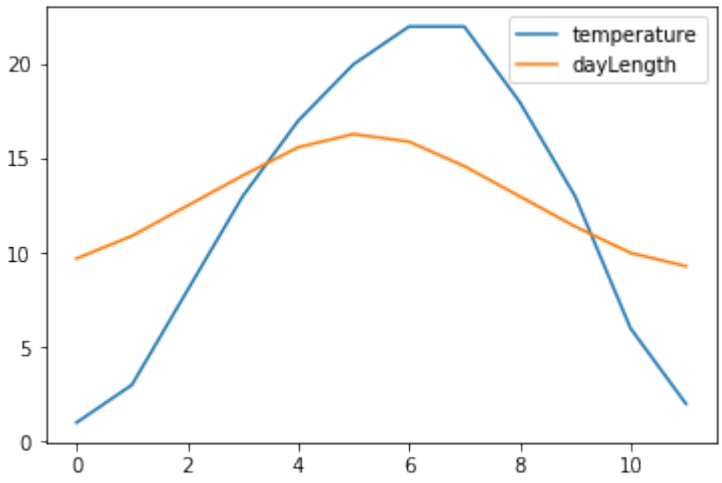
[12]:
```
import pandas as pd

myData = {
    "temperature" : pd.Series([1, 3, 8, 13, 17, 20, 22, 22,18 ,13,6,2],
                              index = ["Jan","Feb", "Mar","Apr","May","Jun",
                                       "Jul","Aug","Sep","Oct","Nov","Dec"]
                               ),
    "dayLength" : pd.Series([9.7, 10.9, 12.5, 14.1, 15.6, 16.3, 15.9,
                             14.6,13,11.4,10,9.3],
                              index = ["Jan","Feb", "Mar","Apr","May","Jun",
                                       "Jul","Aug","Sep","Oct","Nov","Dec"]
                               )

}

DF = pd.DataFrame(myData)
print(DF)

print(DF.columns)
print(DF.index)
```

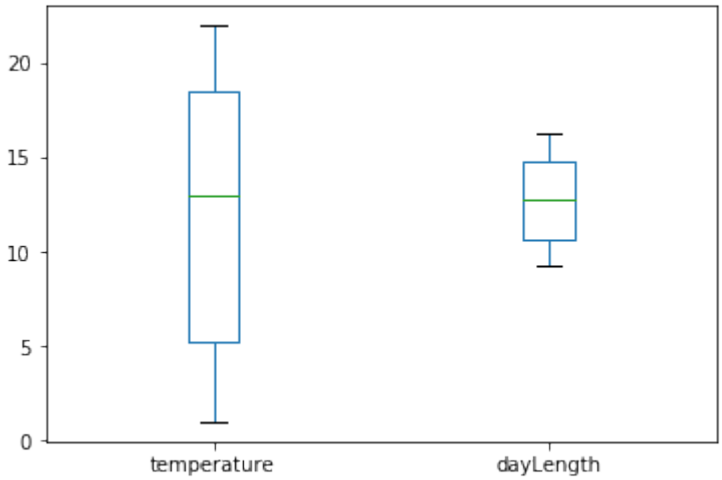
```
     temperature  dayLength
Jan            1        9.7
Feb            3       10.9
Mar            8       12.5
Apr           13       14.1
May           17       15.6
Jun           20       16.3
Jul           22       15.9
Aug           22       14.6
Sep           18       13.0
Oct           13       11.4
Nov            6       10.0
Dec            2        9.3
Index(['temperature', 'dayLength'], dtype='object')
Index(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
       'Nov', 'Dec'],
      dtype='object')
```

If the index is not specified, it is given by default:

[13]:
```
import pandas as pd
import matplotlib.pyplot as plt

myData = {
    "temperature" : pd.Series([1, 3, 8, 13, 17, 20,
                               22, 22,18 ,13,6,2]),
    "dayLength" : pd.Series([9.7, 10.9, 12.5, 14.1, 15.6,
                             16.3,15.9,14.6,13,11.4,10,9.3])

}

DF = pd.DataFrame(myData)
print(DF)
print(DF.index)

DF.plot()
plt.show()
plt.close()
DF.plot(kind ="box")
plt.show()
```

```
    temperature  dayLength
0             1        9.7
1             3       10.9
2             8       12.5
3            13       14.1
4            17       15.6
5            20       16.3
6            22       15.9
7            22       14.6
8            18       13.0
9            13       11.4
10            6       10.0
11            2        9.3
RangeIndex(start=0, stop=12, step=1)
```





It is also possible to define a DataFrame from a list of dictionaries holding a set of values rather than Series. Note that when columns do not have the corresponding information a NaN is added. Indexes and columns can be changed after they have been defined.

[14]:
```
import pandas as pd

myData = [{"A" : 1, "B" : 2, "C" : 3.2, "D" : 10},
          {"A" : 1, "B" : 2, "F" : 3.2, "G" : 10, "H":1},
          {"A" : 1, "B" : 2, "C" : 3.2, "D" : 1,
           "E": 4.1, "F" : 3.2, "G" : 10, "H":1}

         ]

DF = pd.DataFrame(myData)
print(DF)
print("")
#Let's change the columns and indexes
columns = "val1,val2,val3,val4,val5,val6,val7,val8".split(',')
inds = ["Day1", "Day2", "Day3"]

DF.columns = columns
```

(continues on next page)

(continued from previous page)

```
DF.index = inds
print(DF)
```

```
   A  B    C     D    F     G    H    E
0  1  2  3.2  10.0  NaN   NaN  NaN  NaN
1  1  2  NaN   NaN  3.2  10.0  1.0  NaN
2  1  2  3.2   1.0  3.2  10.0  1.0  4.1

      val1  val2  val3  val4  val5  val6  val7  val8
Day1     1     2   3.2  10.0   NaN   NaN   NaN   NaN
Day2     1     2   NaN   NaN   3.2  10.0   1.0   NaN
Day3     1     2   3.2   1.0   3.2  10.0   1.0   4.1
```

9.6.2 Loading data from external files

Pandas also provides methods to load data from external files. In particular, to load data from a .cvs file, we can use the method `pandas.read_csv(filename)`. This method has a lot of parameters, you can see all the details on its usage on the official documentation[58]. Some useful optional parameters are the **separator** (to specify the column separator like `sep="\t"` for tab separated files), the character to identify comments (`comment="#"`) or that to skip some lines (like the header for example or any initial comments to the file) `skiprows=N`. The rows to use as column header can be specified with the parameter `header` that accepts a number or a list of numbers. Similarly, the columns to use as index can be specified with the parameter `index_col`.

Another method to read data in is `pandas.read_excel(filename)` that works in a similar way but can load excel files (see here[59]).

Example: Let's load the data stored in a csv datafile sampledata_orders.csv in a pandas DataFrame.

[15]:
```
import pandas as pd

orders = pd.read_csv("file_samples/sampledata_orders.csv",
                     sep=",", index_col = 0, header = 0)
print("First 5 entries:")
print(orders.head())
print("")
print("The index:")
print(orders.index)
print("")
print("The column names:")
print(orders.columns)
print("")
print("A description of the numerical values:")
print(orders.describe())
```

```
First 5 entries:
        Order ID  Order Date Order Priority  Order Quantity       Sales  \
Row ID
1              3  10/13/2010            Low               6    261.5400
49           293   10/1/2012           High              49  10123.0200
50           293   10/1/2012           High              27    244.5700
```

(continues on next page)

[58] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html#pandas.read_csv
[59] http://pandas.pydata.org/pandas-docs/version/0.20/generated/pandas.read_excel.html

(continued from previous page)

```
80           483   7/10/2011           High              30   4965.7595
85           515   8/28/2010  Not Specified              19    394.2700

        Discount       Ship Mode   Profit  Unit Price  Shipping Cost  \
Row ID
1           0.04     Regular Air  -213.25       38.94          35.00
49          0.07  Delivery Truck   457.81      208.16          68.02
50          0.01     Regular Air    46.71        8.69           2.99
80          0.08     Regular Air  1198.97      195.99           3.99
85          0.08     Regular Air    30.94       21.78           5.94

             Customer Name Province   Region Customer Segment  \
Row ID
1       Muhammed MacIntyre  Nunavut  Nunavut   Small Business
49            Barry French  Nunavut  Nunavut         Consumer
50            Barry French  Nunavut  Nunavut         Consumer
80           Clay Rozendal  Nunavut  Nunavut        Corporate
85          Carlos Soltero  Nunavut  Nunavut         Consumer

       Product Category            Product Sub-Category  \
Row ID
1       Office Supplies          Storage & Organization
49      Office Supplies                      Appliances
50      Office Supplies  Binders and Binder Accessories
80           Technology    Telephones and Communication
85      Office Supplies                      Appliances

                                           Product Name Product Container  \
Row ID
1        Eldon Base for stackable storage shelf, platinum         Large Box
49      1.7 Cubic Foot Compact "Cube" Office Refrigera…        Jumbo Drum
50       Cardinal Slant-D® Ring Binder, Heavy Gauge Vinyl         Small Box
80                                                R380          Small Box
85                            Holmes HEPA Air Purifier        Medium Box

        Product Base Margin   Ship Date
Row ID
1                      0.80  10/20/2010
49                     0.58   10/2/2012
50                     0.39   10/3/2012
80                     0.58   7/12/2011
85                     0.50   8/30/2010

The index:
Int64Index([   1,   49,   50,   80,   85,   86,   97,   98,  103,  107,
            …
            6492, 6526, 6657, 7396, 7586, 7765, 7766, 7906, 7907, 7914],
           dtype='int64', name='Row ID', length=8399)

The column names:
Index(['Order ID', 'Order Date', 'Order Priority', 'Order Quantity', 'Sales',
       'Discount', 'Ship Mode', 'Profit', 'Unit Price', 'Shipping Cost',
       'Customer Name', 'Province', 'Region', 'Customer Segment',
       'Product Category', 'Product Sub-Category', 'Product Name',
       'Product Container', 'Product Base Margin', 'Ship Date'],
      dtype='object')
```

(continues on next page)

(continued from previous page)

```
A description of the numerical values:
           Order ID  Order Quantity         Sales     Discount        Profit  \
count   8399.000000     8399.000000   8399.000000  8399.000000   8399.000000
mean   29965.179783       25.571735   1775.878179     0.049671    181.184423
std    17260.883447       14.481071   3585.050525     0.031823   1196.653372
min        3.000000        1.000000      2.240000     0.000000 -14140.700000
25%    15011.500000       13.000000    143.195000     0.020000    -83.315000
50%    29857.000000       26.000000    449.420000     0.050000     -1.500000
75%    44596.000000       38.000000   1709.320000     0.080000    162.750000
max    59973.000000       50.000000  89061.050000     0.250000  27220.690000

         Unit Price  Shipping Cost  Product Base Margin
count  8399.000000    8399.000000          8336.000000
mean     89.346259      12.838557             0.512513
std     290.354383      17.264052             0.135589
min       0.990000       0.490000             0.350000
25%       6.480000       3.300000             0.380000
50%      20.990000       6.070000             0.520000
75%      85.990000      13.990000             0.590000
max    6783.020000     164.730000             0.850000
```

9.6.3 Extract values by row and column

Once a DataFrame is populated we can access its content. Several options are available:

1. Select by column `DataFrame[col]` returns a Series
2. Select by row label `DataFrame.loc[row_label]` returns a Series
3. Select row by integer location `DataFrame.iloc[row_position]` returns a Series
4. Slice rows `DataFrame[S:E]` (S and E are labels, both included) returns a DataFrame
5. Select rows by boolean vector `DataFrame[bool_vect]` returns a DataFrame

Note that if names are well formed (i.e. no spaces, no strange characters…) we can use DataFrame.col instead of DataFrame[col]. Here are some ways to extract data from the orders dataframe:

[16]:
```
import pandas as pd

orders = pd.read_csv("file_samples/sampledata_orders.csv", sep=",", index_col =0,␣
↪header=0)

print("The Order Quantity column (top 5)")
print(orders["Order Quantity"].head(5))
print("")
print("The Sales column (top 10)")
print(orders.Sales.head(10))
print("")
print("The row with ID:50")
r50 = orders.loc[50]
print(r50)
print("")
print("The third row:")
print(orders.iloc[3])

print("The Order Quantity, Sales, Discount and Profit of the 2nd,4th, 6th and 8th row:
↪")
```

(continues on next page)

(continued from previous page)

```
print(orders[1:8:2][["Order Quantity", "Sales","Discount", "Profit"]])
print("The Order Quantity, Sales, Discount and  Profit of orders with discount > 10%:
↪")
print(orders[orders["Discount"] > 0.1][["Order Quantity", "Sales","Discount", "Profit
↪"]])
```

```
The Order Quantity column (top 5)
Row ID
1      6
49    49
50    27
80    30
85    19
Name: Order Quantity, dtype: int64

The Sales column (top 10)
Row ID
1        261.5400
49     10123.0200
50       244.5700
80      4965.7595
85       394.2700
86       146.6900
97        93.5400
98       905.0800
103     2781.8200
107      228.4100
Name: Sales, dtype: float64

The row with ID:50
Order ID                                                      293
Order Date                                                10/1/2012
Order Priority                                                High
Order Quantity                                                  27
Sales                                                       244.57
Discount                                                      0.01
Ship Mode                                                Regular Air
Profit                                                         46.71
Unit Price                                                      8.69
Shipping Cost                                                   2.99
Customer Name                                           Barry French
Province                                                     Nunavut
Region                                                       Nunavut
Customer Segment                                            Consumer
Product Category                                     Office Supplies
Product Sub-Category                    Binders and Binder Accessories
Product Name            Cardinal Slant-D® Ring Binder, Heavy Gauge Vinyl
Product Container                                           Small Box
Product Base Margin                                              0.39
Ship Date                                                 10/3/2012
Name: 50, dtype: object

The third row:
Order ID                                    483
Order Date                                7/10/2011
Order Priority                             High
Order Quantity                               30
```

(continues on next page)

(continued from previous page)

```
Sales                                          4965.76
Discount                                          0.08
Ship Mode                                  Regular Air
Profit                                         1198.97
Unit Price                                      195.99
Shipping Cost                                     3.99
Customer Name                             Clay Rozendal
Province                                       Nunavut
Region                                         Nunavut
Customer Segment                             Corporate
Product Category                            Technology
Product Sub-Category    Telephones and Communication
Product Name                                       R380
Product Container                           Small Box
Product Base Margin                               0.58
Ship Date                                    7/12/2011
Name: 80, dtype: object
The Order Quantity, Sales, Discount and Profit of the 2nd,4th, 6th and 8th row:
        Order Quantity       Sales  Discount   Profit
Row ID
49                  49  10123.0200      0.07   457.81
80                  30   4965.7595      0.08  1198.97
86                  21    146.6900      0.05     4.43
98                  22    905.0800      0.09   127.70
The Order Quantity, Sales, Discount and  Profit of orders with discount > 10%:
        Order Quantity    Sales  Discount  Profit
Row ID
176                 11  663.784      0.25 -481.04
3721                22  338.520      0.21  -17.75
37                  43  586.110      0.11   98.44
2234                 1   27.960      0.17   -9.13
4900                49  651.900      0.16  -74.51
```

9.6.4 Broadcasting, filtering and computing stats

These work pretty much like on Series and pandas takes care of adding NaNs when it cannot perform some operations due to missing values and so on.

Obviously some operators, when applied to entire tables, might not always make sense (like mean or sum of strings).

[17]:
```
import pandas as pd

orders = pd.read_csv("file_samples/sampledata_orders.csv", sep=",", index_col =0,␣
↪header=0)

orders_10 = orders[["Sales","Profit", "Product Category"]].head(10)
orders_5 = orders[["Sales","Profit", "Product Category"]].head()
orders_20 = orders[["Sales","Profit", "Product Category"]].head(20)
print(orders_20)
print("")

#Summing over the entries, does not make sense but...
print("Top10 + Top5:")
print(orders_10 + orders_5)
print("")
```

(continues on next page)

(continued from previous page)

```
prod_cost = orders_20["Sales"] - orders_20["Profit"]
print(prod_cost)

print("")
print("Technology orders")
print(orders_20[orders_20["Product Category"] == "Technology"])
print("")
print("Office Supplies positive profit")
print(orders_20[ (orders_20["Product Category"] == "Office Supplies") & (orders_20[
↪"Profit"] >0) ])
```

```
             Sales   Profit Product Category
Row ID
1         261.5400  -213.25  Office Supplies
49      10123.0200   457.81  Office Supplies
50        244.5700    46.71  Office Supplies
80       4965.7595  1198.97       Technology
85        394.2700    30.94  Office Supplies
86        146.6900     4.43        Furniture
97         93.5400   -54.04  Office Supplies
98        905.0800   127.70  Office Supplies
103      2781.8200  -695.26  Office Supplies
107       228.4100  -226.36  Office Supplies
127       196.8500  -166.85  Office Supplies
128       124.5600   -14.33  Office Supplies
134       716.8400   134.72  Office Supplies
135      1474.3300   114.46       Technology
149        80.6100    -4.72  Office Supplies
160      1815.4900   782.91        Furniture
161       248.2600    93.80  Office Supplies
175      4462.2300   440.72        Furniture
176       663.7840  -481.04        Furniture
203       834.9040   -11.68       Technology

Top10 + Top5:
            Sales   Profit                Product Category
Row ID
1          523.080  -426.50  Office SuppliesOffice Supplies
49       20246.040   915.62  Office SuppliesOffice Supplies
50         489.140    93.42  Office SuppliesOffice Supplies
80        9931.519  2397.94           TechnologyTechnology
85         788.540    61.88  Office SuppliesOffice Supplies
86             NaN      NaN                             NaN
97             NaN      NaN                             NaN
98             NaN      NaN                             NaN
103            NaN      NaN                             NaN
107            NaN      NaN                             NaN

Row ID
1        474.7900
49      9665.2100
50       197.8600
80      3766.7895
85       363.3300
86       142.2600
97       147.5800
```

(continues on next page)

(continued from previous page)

```
98      777.3800
103    3477.0800
107     454.7700
127     363.7000
128     138.8900
134     582.1200
135    1359.8700
149      85.3300
160    1032.5800
161     154.4600
175    4021.5100
176    1144.8240
203     846.5840
dtype: float64

Technology orders
            Sales   Profit Product Category
Row ID
80      4965.7595  1198.97       Technology
135     1474.3300   114.46       Technology
203      834.9040   -11.68       Technology

Office Supplies positive profit
           Sales  Profit Product Category
Row ID
49      10123.02  457.81  Office Supplies
50        244.57   46.71  Office Supplies
85        394.27   30.94  Office Supplies
98        905.08  127.70  Office Supplies
134       716.84  134.72  Office Supplies
161       248.26   93.80  Office Supplies
```

Statistics can be computed by **column** (normally the default) or by **row** (specifying `axis=1`) or on the **entire table**.

Here[60] you can find the complete list of methods that can be applied.

[18]:
```
import pandas as pd

data = pd.read_csv("file_samples/random.csv", sep=",", header=0)

print("Global description")
print(data.describe())
print("")

print(data)
print("")

print("Let's reduce A in [0,1]")
print(data["A"]/ data["A"].max())

print("Let's reduce first row in [0,1]")
print(data.iloc[0]/ data.loc[0].max())

print("Mean and std values (by column)")
print(data.mean())
```

(continues on next page)

[60] http://pandas.pydata.org/pandas-docs/version/0.20/api.html#api-dataframe-stats

(continued from previous page)

```
print(data.std())
print("")
print("Mean and std values (by column) - top 10 values")
print(data.head(10).mean(axis=1))
print(data.head(10).std(axis=1))

print("Cumulative sum (by column)- top 10 rows")
print(data.head(10).cumsum())

print("Cumulative sum (by row) - top 10 rows")
print(data.head(10).cumsum(axis = 1 ))
```

```
Global description
                A           B           C           D           E           F  \
count   50.000000   50.000000   50.000000   50.000000   50.000000   50.000000
mean    45.240000   49.760000   54.120000   50.500000   57.860000   49.180000
std     28.393417   32.193015   25.165282   28.321226   32.443364   28.488229
min      0.000000    0.000000    5.000000    1.000000    2.000000    1.000000
25%     21.750000   18.000000   35.750000   28.250000   29.250000   25.000000
50%     42.000000   49.500000   52.500000   49.500000   68.000000   52.000000
75%     63.250000   76.250000   73.750000   76.000000   86.750000   75.500000
max     99.000000   99.000000  100.000000   95.000000  100.000000   97.000000

                G          H           I           L
count    50.000000   50.00000   50.000000   50.000000
mean     49.780000   47.78000   46.920000   51.780000
std      29.673248   29.75978   31.056788   30.613649
min       1.000000    1.00000    0.000000    3.000000
25%      26.500000   24.50000   19.000000   26.250000
50%      51.000000   42.50000   44.000000   46.500000
75%      76.750000   72.00000   73.000000   82.000000
max     100.000000   99.00000   98.000000  100.000000

     A   B    C   D    E   F    G   H   I    L
0   28  69    5  41   12  62   90  32  33   85
1   91  59   64  94    4  51   45  52  96    5
2   54  74   53  92   20  84   85  21  98   73
3   99   0   39  12   90  15   62  38   4   67
4   98  10   35  77   46  97   88  72  15   37
5   44   9   91  57   98  63   52  77  62    7
6   76  17   12  41   69  34  100  29   0   91
7   52  25   43  59   87  91    1  90  23   90
8   56  33   38  91   13   1   34  17  22   82
9   29  18   42  48   34  16   64   7  46    9
10  81  77   38  93   63  90   57  58  52  100
11  57  97   23  59   38  93   46  49  88   86
12  57   0   79  78  100  76   16  31  79    8
13  18   1   67  44    2  17   53  51   6   23
14  77  95   94  88    9  25   54  23  36   35
15  87  42   63  33   95  53   11  58  53    3
16  34  24   24  28   74  79   86  98  42   90
17   1  31   20  63   70   3   29  39   5   87
18  35  17   77  15    9  63   86  95  95   21
19   0  85   13  13   60  74   44  78   7   56
20  61  20   47  21   85  78   53  14  37   70
21  81  15   89  54   38  59   72  35  59   12
22  20  81   76  15   64  18    9  47  11   81
```

(continues on next page)

(continued from previous page)

```
23  57  94   73  19   21  79   81  45  71   40
24   2  58   83  67   99  46   76  70  50   45
25  64  13   52  60   32  70   32   4  73   20
26  27  49  100  48   70  30   58  48  42   36
27  82  89   93  73   30  64   18  54  93   82
28  19  85   98   7   84  71   77  20  18   63
29  34  95   68  23   68  38   73   1  28   10
30  33  60   66  80   94  41   53  99  29   57
31   0  99   18   2   29  28   11  30  36   56
32  20  61   33  89   52  59   50  55   4   33
33  82  72   41  33   92  16   18  99  73   91
34   8  28   22  29   83  53   47  10  42   34
35  64  50   24  59    6  83   92  91  87   98
36  42  73   50  17   91   9   39  37  97   50
37  27  49   67  61   21  25   94  96  16   98
38  44  44   48  70   86  85    9  31   4   46
39  36  93   64   1   96  89   14  75  10   84
40  20  13   34  87   11  10    2   5  91   27
41  54   2   61  80   98  53   96  40   2   26
42  42  88   79  31   41  23   50  95  77   26
43   7  14   56  95   70  28   78  31  34   95
44  40  65   34  44   97  36    3  90  55   13
45  97  61   45  32   91  26   84  33  57   80
46  60  49   47  92   81  51    7  18  66   47
47  39  98   90  33   68   5   26  72  76   37
48   6  18   74  51   77  19   28  21  81   46
49  20  69   54  26   25  80   36   8  65   31

Let's reduce A in [0,1]
0     0.282828
1     0.919192
2     0.545455
3     1.000000
4     0.989899
5     0.444444
6     0.767677
7     0.525253
8     0.565657
9     0.292929
10    0.818182
11    0.575758
12    0.575758
13    0.181818
14    0.777778
15    0.878788
16    0.343434
17    0.010101
18    0.353535
19    0.000000
20    0.616162
21    0.818182
22    0.202020
23    0.575758
24    0.020202
25    0.646465
26    0.272727
27    0.828283
```

(continues on next page)

(continued from previous page)

```
28    0.191919
29    0.343434
30    0.333333
31    0.000000
32    0.202020
33    0.828283
34    0.080808
35    0.646465
36    0.424242
37    0.272727
38    0.444444
39    0.363636
40    0.202020
41    0.545455
42    0.424242
43    0.070707
44    0.404040
45    0.979798
46    0.606061
47    0.393939
48    0.060606
49    0.202020
Name: A, dtype: float64
Let's reduce first row in [0,1]
A    0.311111
B    0.766667
C    0.055556
D    0.455556
E    0.133333
F    0.688889
G    1.000000
H    0.355556
I    0.366667
L    0.944444
Name: 0, dtype: float64
Mean and std values (by column)
A    45.24
B    49.76
C    54.12
D    50.50
E    57.86
F    49.18
G    49.78
H    47.78
I    46.92
L    51.78
dtype: float64
A    28.393417
B    32.193015
C    25.165282
D    28.321226
E    32.443364
F    28.488229
G    29.673248
H    29.759780
I    31.056788
L    30.613649
```

(continues on next page)

(continued from previous page)

```
dtype: float64

Mean and std values (by column) - top 10 values
0    45.7
1    56.1
2    65.4
3    42.6
4    57.5
5    56.0
6    46.9
7    56.1
8    38.7
9    31.3
dtype: float64
0    29.424291
1    33.013297
2    27.785688
3    35.647035
4    33.069792
5    30.342489
6    34.853503
7    32.942543
8    29.431842
9    18.826990
dtype: float64
Cumulative sum (by column)- top 10 rows
     A    B    C    D    E    F    G    H    I    L
0   28   69    5   41   12   62   90   32   33   85
1  119  128   69  135   16  113  135   84  129   90
2  173  202  122  227   36  197  220  105  227  163
3  272  202  161  239  126  212  282  143  231  230
4  370  212  196  316  172  309  370  215  246  267
5  414  221  287  373  270  372  422  292  308  274
6  490  238  299  414  339  406  522  321  308  365
7  542  263  342  473  426  497  523  411  331  455
8  598  296  380  564  439  498  557  428  353  537
9  627  314  422  612  473  514  621  435  399  546
Cumulative sum (by row) - top 10 rows
    A    B    C    D    E    F    G    H    I    L
0  28   97  102  143  155  217  307  339  372  457
1  91  150  214  308  312  363  408  460  556  561
2  54  128  181  273  293  377  462  483  581  654
3  99   99  138  150  240  255  317  355  359  426
4  98  108  143  220  266  363  451  523  538  575
5  44   53  144  201  299  362  414  491  553  560
6  76   93  105  146  215  249  349  378  378  469
7  52   77  120  179  266  357  358  448  471  561
8  56   89  127  218  231  232  266  283  305  387
9  29   47   89  137  171  187  251  258  304  313
```

9.6.5 Merging DataFrames

It is possible to merge together DataFrames having a common column name. The merge can be done with the pandas `merge` method. Upon merging, the two tables will be concatenated into a bigger one containing information from both DataFrames. The basic syntax is:

```
pandas.merge(DataFrame1, DataFrame2, on="col_name", how="inner/outer/left/right")
```

The column on which the merge has to be performed is specified with the parameter `on` followed by the column name, while the behaviour of the merging depends on the parameter `how` that tells pandas how to behave when dealing with non-matching data. In particular:

```
1. how = inner : non-matching entries are discarded;
2. how = left : ids are taken from the first DataFrame;
3. how = right : ids are taken from the second DataFrame;
4. how = outer : ids from both are retained.
```

[19]:
```
import pandas as pd

Snp1 = {"id": ["SNP_FB_0411211","SNP_FB_0412425","SNP_FB_0942385",
               "CH01f09","Hi05f12x","SNP_FB_0942712" ],
        "type" : ["SNP","SNP","SNP","SSR","SSR","SNP"]
       }
Snp2 = {"id" : ["SNP_FB_0411211","SNP_FB_0412425",
                "SNP_FB_0942385", "CH01f09","SNP_FB_0428218" ],
      "chr" : ["1", "15","7","9","1"]
     }

DFs1 = pd.DataFrame(Snp1)
DFs2 = pd.DataFrame(Snp2)

print(DFs1)
print(DFs2)
print("")
print("Inner merge (only common in both)")
inJ = pd.merge(DFs1,DFs2, on = "id", how = "inner")
print(inJ)
print("")
print("Left merge (IDS from DFs1)")
leftJ = pd.merge(DFs1,DFs2, on = "id", how = "left")
print(leftJ)
print("")
print("Right merge (IDS from DFs2)")
rightJ = pd.merge(DFs1,DFs2, on = "id", how = "right")
print(rightJ)
print("")
print("Outer merge (IDS from both)")
outJ = pd.merge(DFs1,DFs2, on = "id", how = "outer")
print(outJ)
```

```
               id type
0  SNP_FB_0411211  SNP
1  SNP_FB_0412425  SNP
2  SNP_FB_0942385  SNP
3         CH01f09  SSR
4        Hi05f12x  SSR
5  SNP_FB_0942712  SNP
```

(continues on next page)

(continued from previous page)

```
               id chr
0  SNP_FB_0411211   1
1  SNP_FB_0412425  15
2  SNP_FB_0942385   7
3         CH01f09   9
4  SNP_FB_0428218   1

Inner merge (only common in both)
               id type chr
0  SNP_FB_0411211  SNP   1
1  SNP_FB_0412425  SNP  15
2  SNP_FB_0942385  SNP   7
3         CH01f09  SSR   9

Left merge (IDS from DFs1)
               id type  chr
0  SNP_FB_0411211  SNP    1
1  SNP_FB_0412425  SNP   15
2  SNP_FB_0942385  SNP    7
3         CH01f09  SSR    9
4        Hi05f12x  SSR  NaN
5  SNP_FB_0942712  SNP  NaN

Right merge (IDS from DFs2)
               id type chr
0  SNP_FB_0411211  SNP   1
1  SNP_FB_0412425  SNP  15
2  SNP_FB_0942385  SNP   7
3         CH01f09  SSR   9
4  SNP_FB_0428218  NaN   1

Outer merge (IDS from both)
               id type  chr
0  SNP_FB_0411211  SNP    1
1  SNP_FB_0412425  SNP   15
2  SNP_FB_0942385  SNP    7
3         CH01f09  SSR    9
4        Hi05f12x  SSR  NaN
5  SNP_FB_0942712  SNP  NaN
6  SNP_FB_0428218  NaN    1
```

Note that the columns we merge on do not necessarily need to be the same, we can specify a correspondence between the row of the first dataframe (the one on the left) and the second dataframe (the one on the right) specifying which columns must have the same values to perform the merge. This can be done by using the parameters `right_on = column_name` and `left_on = column_name`:

[20]:
```
#%reset -s -f
%clear

import pandas as pd

d = dict({"A" : [1,2,3,4], "B" : [3,4,73,13]})
d2 = dict({"E" : [1,4,3,13], "F" : [3,1,71,1]})

DF = pd.DataFrame(d)
DF2 = pd.DataFrame(d2)
```

(continues on next page)

(continued from previous page)

```
merged_onBE = DF.merge(DF2, left_on = 'B', right_on = 'E', how = "inner")
merged_onAF = DF.merge(DF2, right_on = "F", left_on = "A", how = "outer")
print("DF:")
print(DF)
print("DF2:")
print(DF2)
print("\ninner merge on BE")
print(merged_onBE)
print("\nouter merge on AF:")
print(merged_onAF)
```

```
DF:
   A   B
0  1   3
1  2   4
2  3  73
3  4  13
DF2:
    E   F
0   1   3
1   4   1
2   3  71
3  13   1

inner merge on BE
   A   B   E   F
0  1   3   3  71
1  2   4   4   1
2  4  13  13   1

outer merge on AF:
     A     B     E     F
0  1.0   3.0   4.0   1.0
1  1.0   3.0  13.0   1.0
2  2.0   4.0   NaN   NaN
3  3.0  73.0   1.0   3.0
4  4.0  13.0   NaN   NaN
5  NaN   NaN   3.0  71.0
```

9.6.6 Grouping

Given a DataFrame, it is possible to apply the so called **split-apply-aggregate** processing method. Basically, this helps to group the data according to the value of some columns, apply a function on the different groups and aggregate the results in a new DataFrame.

Grouping of a DataFrame can be done by using the **DataFrame.groupby(columnName)** method which returns a `pandas.DataFrameGroupBy` object, which basically is composed of two information: the **name** shared by each group and a **DataFrame** containing all the elements belonging to that group.

Given a grouped DataFrame, each group can be obtained with the **get_group(groupName)** it is also possible to loop over the groups:

```
for groupName, groupDataFrame in grouped:
    #do something
```

It is also possible to aggregate the groups by applying some functions to each of them (please refer to the documentation[61]).

Let's see these things in action.

Example Given a list of labels and integers group them by label and compute the mean of all values in each group.

[21]:
```
import pandas as pd

test = {"x": ["a","a", "b", "b", "c", "c" ],
        "y" : [2,4,0,5,5,10]
       }

DF = pd.DataFrame(test)
print(DF)
print("")
gDF = DF.groupby("x")
for i,g in gDF:
    print("Group: ", i)
    print(g)
    print(type(g))

aggDF = gDF.aggregate(pd.DataFrame.mean)
print(aggDF)

#without looping through the groups...
print("\nThe 'a' group:")
print(gDF.get_group('a'))
print("\nThe 'c' group:")
print(gDF.get_group('c'))
```

```
   x   y
0  a   2
1  a   4
2  b   0
3  b   5
4  c   5
5  c  10

Group:  a
   x  y
0  a  2
1  a  4
<class 'pandas.core.frame.DataFrame'>
Group:  b
   x  y
2  b  0
3  b  5
<class 'pandas.core.frame.DataFrame'>
Group:  c
   x   y
4  c   5
5  c  10
<class 'pandas.core.frame.DataFrame'>
```

(continues on next page)

[61] http://pandas.pydata.org/pandas-docs/version/0.20/api.html#api-dataframe-stats

(continued from previous page)

```
     y
x
a  3.0
b  2.5
c  7.5

The 'a' group:
   x  y
0  a  2
1  a  4

The 'c' group:
   x   y
4  c   5
5  c  10
```

Example Given the orders data seen above, let's get the columns "Sales", "Profit" and "Product Category" and compute the total sum and the mean of the sales and profits for each product category.

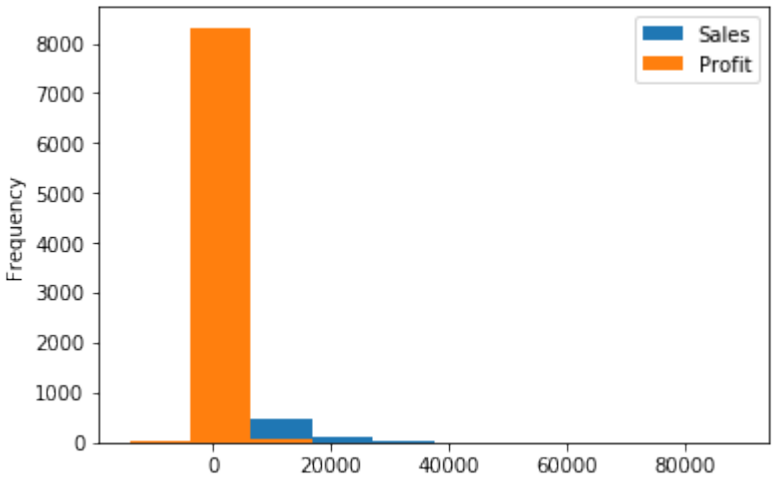
[22]:
```
import pandas as pd
import matplotlib.pyplot as plt

orders = pd.read_csv("file_samples/sampledata_orders.csv", sep=",",
                     index_col =0, header=0)

SPC = orders[["Sales","Profit", "Product Category"]]
print(SPC.head())

SPC.plot(kind = "hist", bins = 10)
plt.show()

print("")
grouped = SPC.groupby("Product Category")
for i,g in grouped:
    print("Group: ", i)

print("")
print("Count elements per category:") #get the series corresponding to the column
                                      #and apply the value_counts() method
print(orders["Product Category"].value_counts())
print("")
print("Total values:")
print(grouped.aggregate(pd.DataFrame.sum)[["Sales"]])

print("Mean values (sorted by profit):")
mv_sorted = grouped.aggregate(pd.DataFrame.mean).sort_values(by="Profit")
print(mv_sorted)
print("")
print("The most profitable is {}".format(mv_sorted.index[-1]))
```

```
              Sales   Profit Product Category
Row ID
1         261.5400  -213.25  Office Supplies
49      10123.0200   457.81  Office Supplies
50        244.5700    46.71  Office Supplies
80       4965.7595  1198.97       Technology
```

(continues on next page)

(continued from previous page)

```
85         394.2700    30.94  Office Supplies
```




```
Group:  Furniture
Group:  Office Supplies
Group:  Technology

Count elements per category:
Office Supplies    4610
Technology         2065
Furniture          1724
Name: Product Category, dtype: int64

Total values:
                         Sales
Product Category
Furniture          5178590.542
Office Supplies    3752762.100
Technology         5984248.182
Mean values (sorted by profit):
                         Sales      Profit
Product Category
Furniture          3003.822820   68.116607
Office Supplies     814.048178  112.369072
Technology         2897.941008  429.207516

The most profitable is Technology
```

9.7 Exercises

1. The file top_3000_words.txt is a one-column file representing the top 3000 English words. Read the file and for each letter, count how many words start with that letter. Store this information in a dictionary. Create a pandas series from the dictionary and plot an histogram of all initials counting more than 100 words starting with them.

Show/Hide Solution

2. The file filt_aligns.tsv is a tab separated value file representing alignments of paired-end reads on some apple chromosomes. Paired end reads have the property of being X bases apart from each other as they have been sequenced from the two ends of some size-selected DNA molecules.

Each line of the file has the following information `readID\tChrPE1\tAlignmentPosition1\tChrPE2\tAlignmentPosition2`. The two ends of the same pair have the same readID. Load the read pairs aligning on the same chromosome into two dictionaries. The first (`inserts`) having readID as keys and the insert size (i.e. the absolute value of AlignmentPosition1 - AlignmentPosition2) as value. The second dictionary (`chrs`)will have readID as key and chromosome ID as value. Example:

```
readID Chr11 31120 Chr11 31472
readID1 Chr7 12000 Chr11 11680
```

will result in:

```
inserts = {"readID" : 352, "readID1" : 320}
chrs = {"readID" : "Chr11", "readID1" : "Chr7"}
```

Once you have the two dictionaries:

```
1. Create a Series with all the insert sizes and show some
of its stats with the method **describe**. What is the mean
insert size? How many paried end are we using to create
this distribution?
2. Display the first 5 values of the series
3. Make a box plot to assess the distribution of the values.
4. Make an histogram to see the values in a different way.
How does this distribution look like?
5. Create another series from the chromosome info and print
the first 10 elements
6. Create a DataFrame starting from the two Series and print
the first 10 elements
7. For each chromosome, get the average insert size of the
paired aligned to it (hint:use group by).
8. Make a box plot of the average insert size per chromosome.
```

Show/Hide Solution

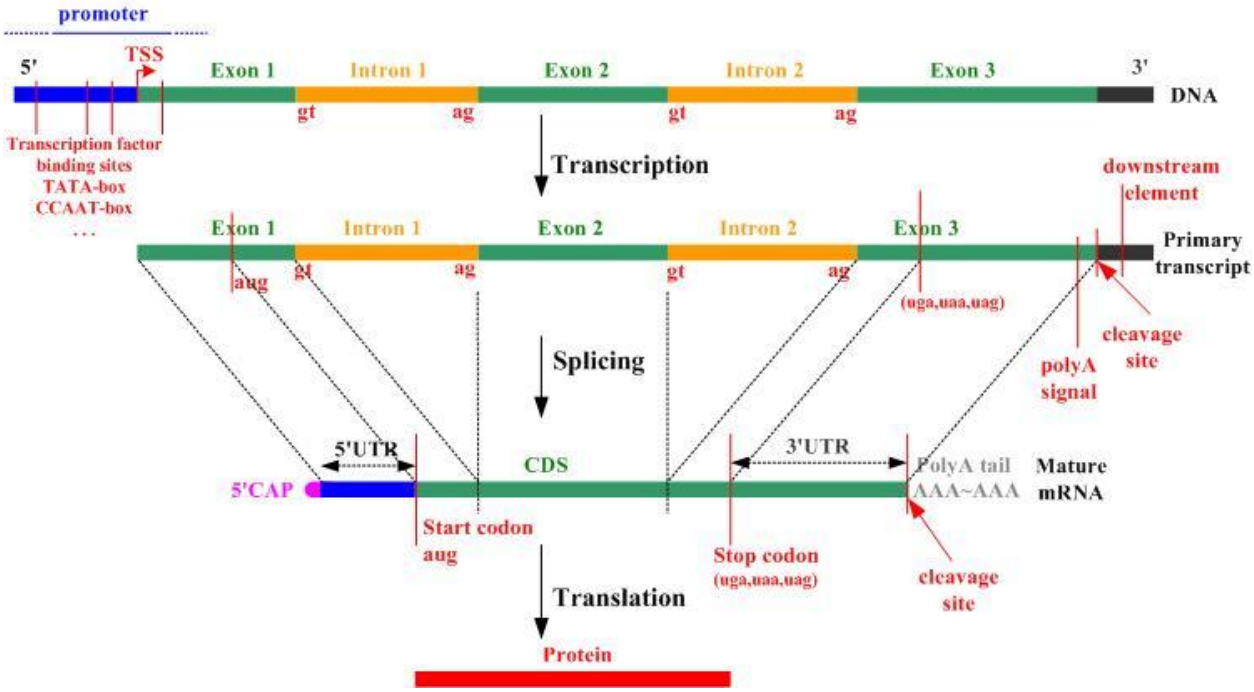
3. Download the train.csv dataset. As the name says it is a .csv file. The file contains information regarding loans given or refused to applicants. Information on the gender, marital status, education, work and income of the applicant is reported alongside the amount and length of the loan and credit history (i.e. 0 if no previous loan was given, 1

otherwise). Open it in a text editor or excel and inspect it first. Then, answer the following questions (if you have any doubts check here[62]):

0. Load it into a pandas DataFrame (use column `Loan_ID` as index. Hint: use parm index_col).
1. Get an idea of the data by visualizing its first 5 entries;
2. How many total entries are present in the file? How many males and females?
3. What is the average applicant income? Does the gender affect the income? Compute the average of the applicant income on the whole dataset and the average of the data grouped by Gender. How many Females have an income > than the average?
4. How many loans have been given (i.e. Loan_Status equals Y)? What is the percentage of the loans given and that of the loans refused?
5. What is the percentage of given/refused loans in the case of married people?
6. What is the percentage of given/refused loans in the case of applicants with positive credit history (i.e. Credit_History equals 1)?

Show/Hide Solution

4. [Thanks to Stefano Teso] DNA transcription and translation into proteins follows this schema:



The file gene_table.csv is a comma separated value file representing a summary of the annotation of several human genes based on the Ensembl[63] annotation. For each gene it contains the following information:

```
gene_name,gene_biotype,chromosome,strand,transcript_count
```

where `gene_name` is based on the HGNC[64] nomenclature. `gene_biotype` represents the biotype (refer to VEGA[65] like protein_coding, pseudogene, lincRNA, miRNA etc. `chromosome` is where the feature is located, `strand` is a + or a - for the forward or reverse strand and `transcript_count` reports the number of isoforms of the gene.

[62] http://pandas.pydata.org/pandas-docs/version/0.20/api.html
[63] http://www.ensembl.org/index.html
[64] http://www.genenames.org/
[65] http://vega.sanger.ac.uk/info/about/gene_and_transcript_types.html

A sample of the file follows:

```
TSPAN6,protein_coding,chrX,-,5
TNMD,protein_coding,chrX,+,2
DPM1,protein_coding,chr20,-,6
SCYL3,protein_coding,chr1,-,5
C1orf112,protein_coding,chr1,+,9
FGR,protein_coding,chr1,-,7
CFH,protein_coding,chr1,+,6
FUCA2,protein_coding,chr6,-,3
GCLC,protein_coding,chr6,-,13
```

Write a python program that: 0. Loads the gene_table.csv in a DataFrame (inspect the first entries with head to check the content of the file); 1. Computes the number of genes annotated for the human genome; 2. Computes the minimum, maximum, average and median number of known isoforms per gene (consider the transcript_count column as a series). 3. Plots a histogram and a boxplot of the number of known isoforms per gene 4. Computes the number of different biotypes. How many genes do we have for each genotype? Plot the number of genes per biotype in a horizontal bar plot (hint: use also figsize = (10,10) to make it visible; 5. Computes the number of different chromosomes 6. Computes, for each chromosome, the number of genes it contains, and prints a horizontal barplot with the number of genes per chromosome. 7. Computes, for each chromosome, the percentage of genes located on the + strand 8. Computes, for each biotype, the average number of transcripts associated to genes belonging to the biotype. Finally, plots them in a vertical bar plot

Show/Hide Solution

5. Write a function that creates and returns a data frame having columns with the labels specified through a list taken in input and ten rows of random data between 1 and 100.
    1. Create two random DataFrames one with labels l = [“A”, “B”, “C” ,”D”,”E”] and l1 = [“W”, “X”,”Y”,”Z”]
    2. Print the first 5 elements to inspect the two DataFrames and plot the valuse of the two DataFrames;
    3. Create a new Series S1 with values: [1, 1, 1, 0, 0, 1, 1, 0, 0, 1]. Get the Series corresponding to the column “C” and multiply it point-to-point by the Series S1 (hint use: Series.multiply(S1))
    4. Add this new series as a “C” column of the second DataFrame.
    5. Merge the two DataFrames based on the value of C. Perform an inner, outer, left and right merge and see the difference

Show/Hide Solution

CHAPTER

TEN