Jupman

The webpage of the Scientific Programming Lab for QCB 2020

Luca Bianco

Oct 05, 2020

CONTENTS

1 General Info **3**
- 1.1 Timetable and lecture rooms . . . 3
- 1.2 Moodle . . . 3
- 1.3 Zoom links . . . 3
- 1.4 Slides . . . 3
- 1.5 Acknowledgements . . . 4

2 Practical 1 **5**
- 2.1 Slides . . . 5
- 2.2 Setting up the environment . . . 5
- 2.3 Our toolbox . . . 6
- 2.4 Installing Python3 in Linux . . . 6
- 2.5 Installing Python3 in Windows/Mac . . . 7
- 2.6 The console . . . 10
- 2.7 Visual Studio Code . . . 12
- 2.8 The debugger . . . 18
- 2.9 A quick Jupyter primer (just for your information, skip if not interested) . . . 20
- 2.10 Exercises . . . 22

3 Practical 2 **27**
- 3.1 Slides . . . 27
- 3.2 Modules . . . 27
- 3.3 Objects . . . 28
- 3.4 Variables . . . 28
- 3.5 Numeric types . . . 30
- 3.6 Strings . . . 33
- 3.7 Exercises . . . 38

4 Practical 3 **49**
- 4.1 Slides . . . 49
- 4.2 Lists . . . 49
- 4.3 Tuples . . . 59
- 4.4 Exercises . . . 62

5 Practical 4 **71**
- 5.1 Slides . . . 71
- 5.2 Execution flow . . . 71
- 5.3 Conditionals . . . 72
- 5.4 Loops . . . 74
- 5.5 Exercises . . . 79

6 83

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.

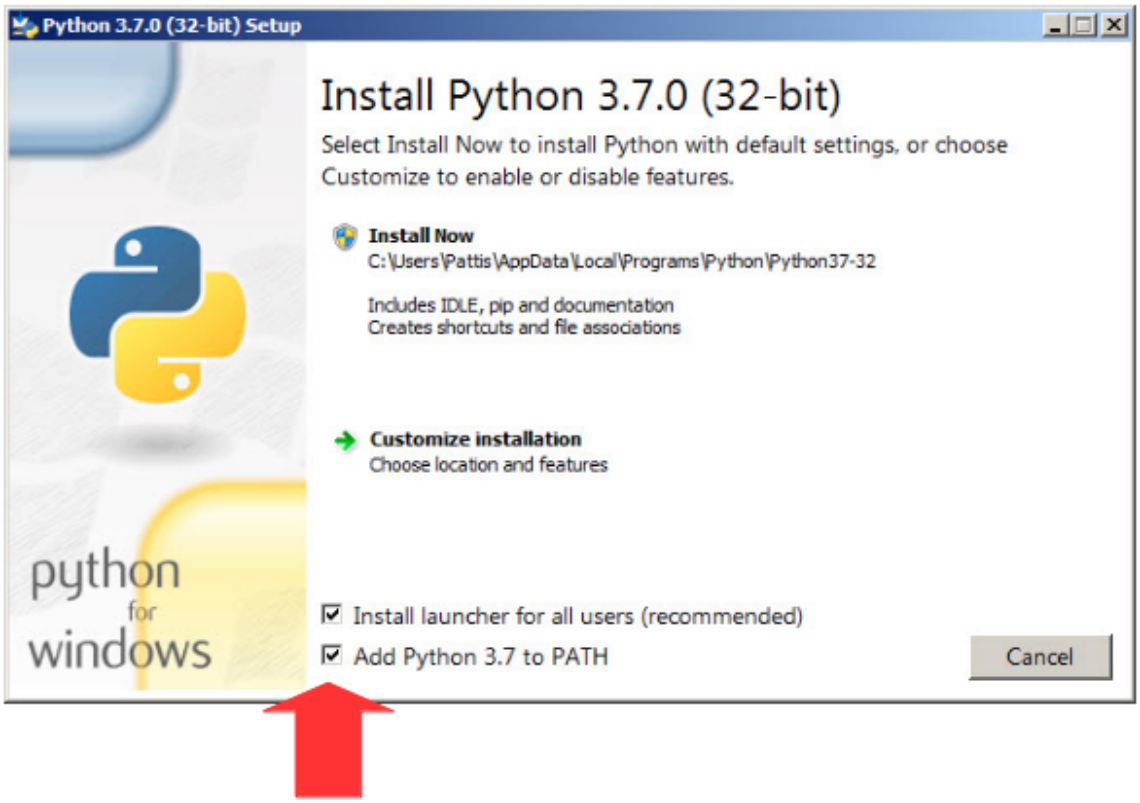
2.5.1 OPTION 1:

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):

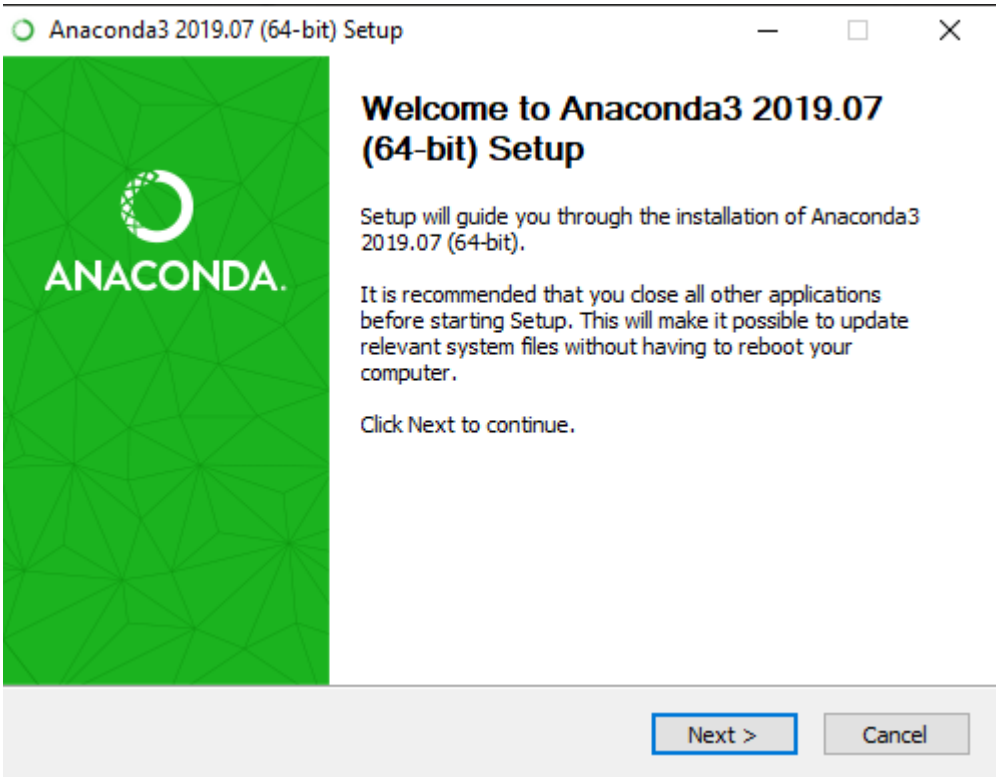
Additional Information:

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!).

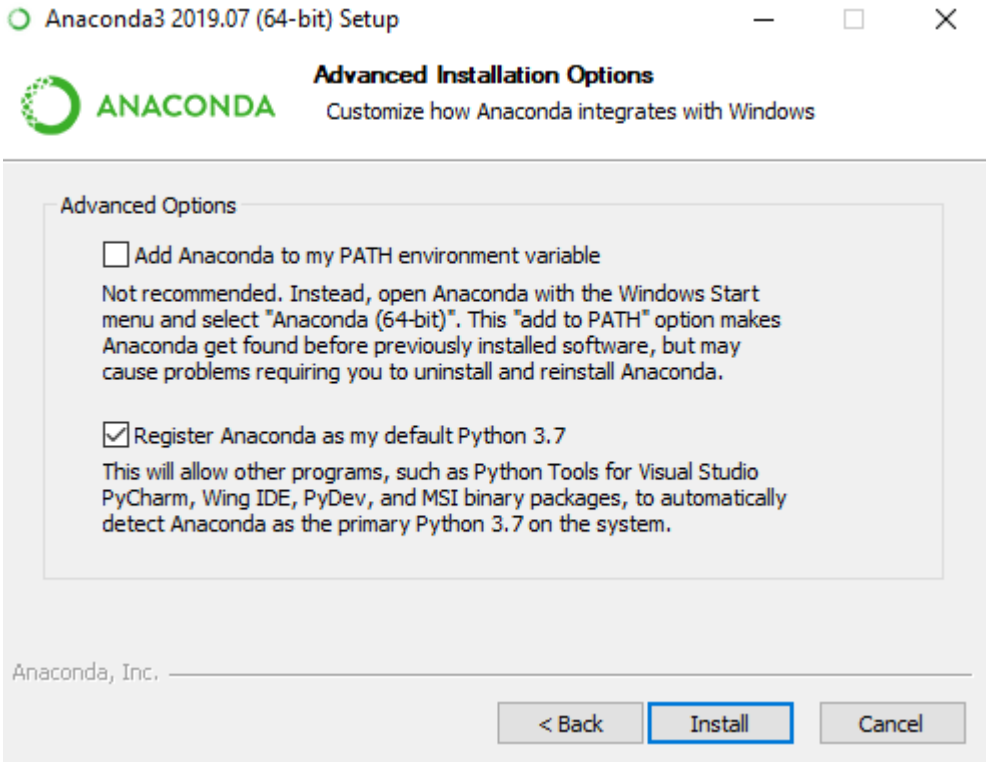
Anaconda is available here[23]

Upon launching the installer you should be prompted something like:

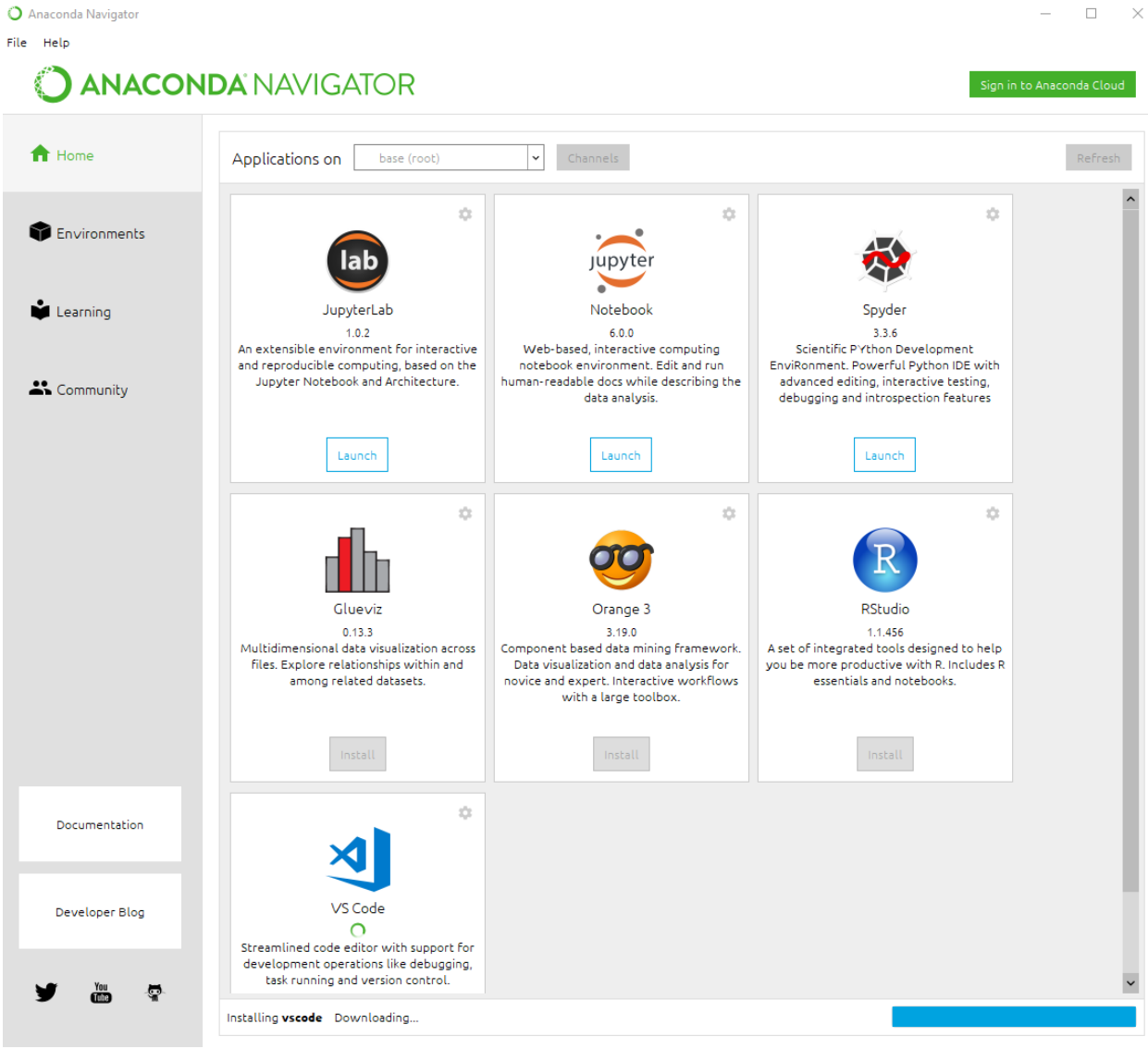
[21] https://code.visualstudio.com/
[22] https://code.visualstudio.com/Download
[23] https://www.anaconda.com/distribution/



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].

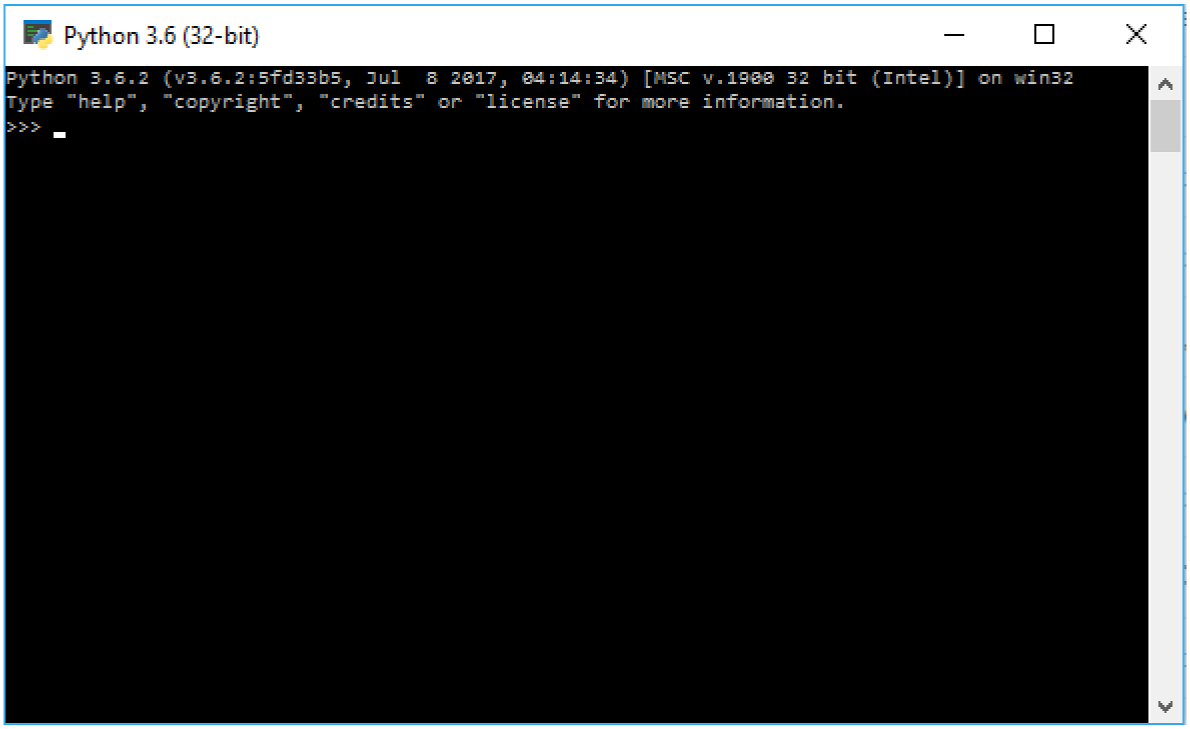
2.6 The console

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)
```

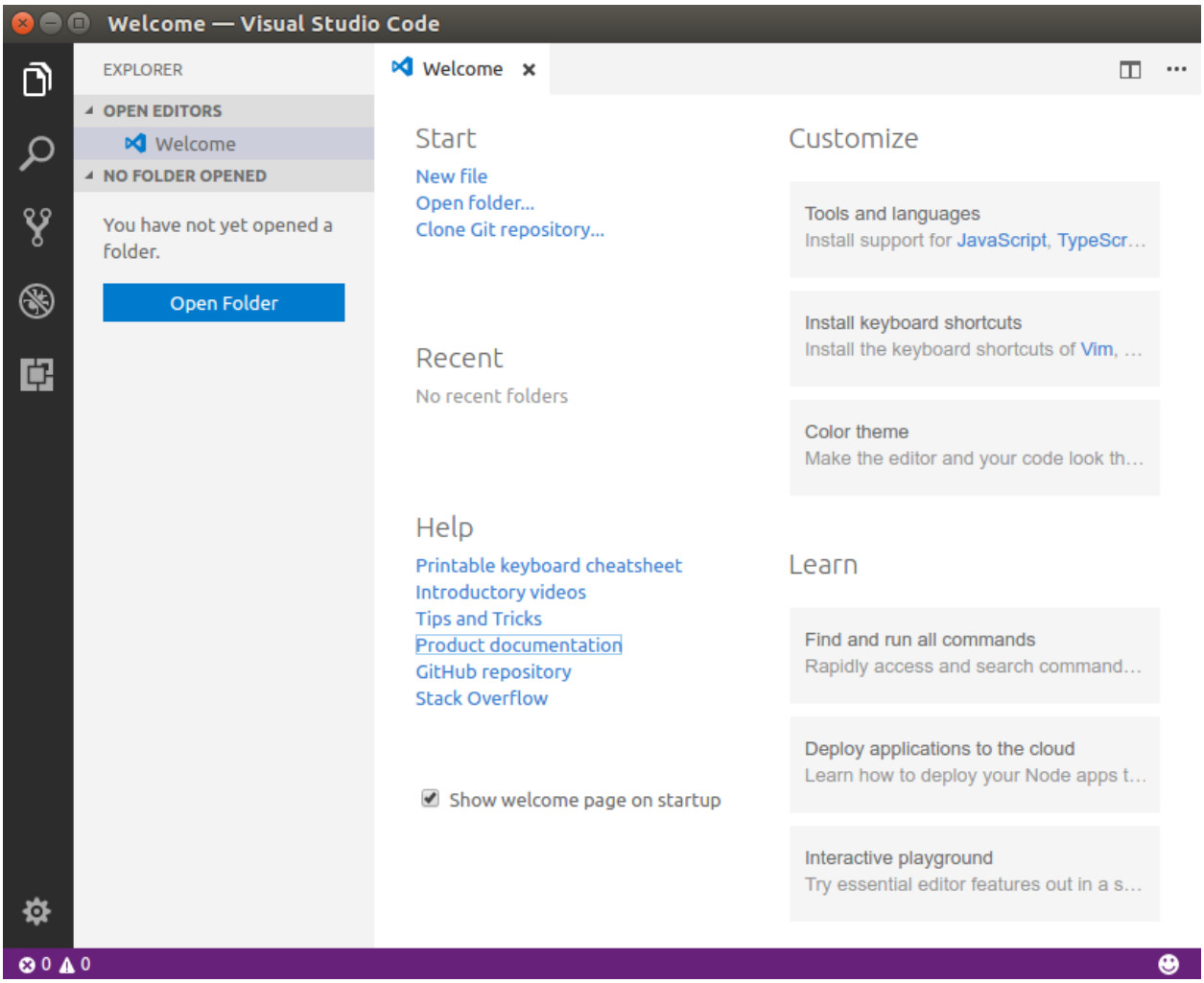
```
sum is: 17  division is: 1.4285714285714286
```

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:



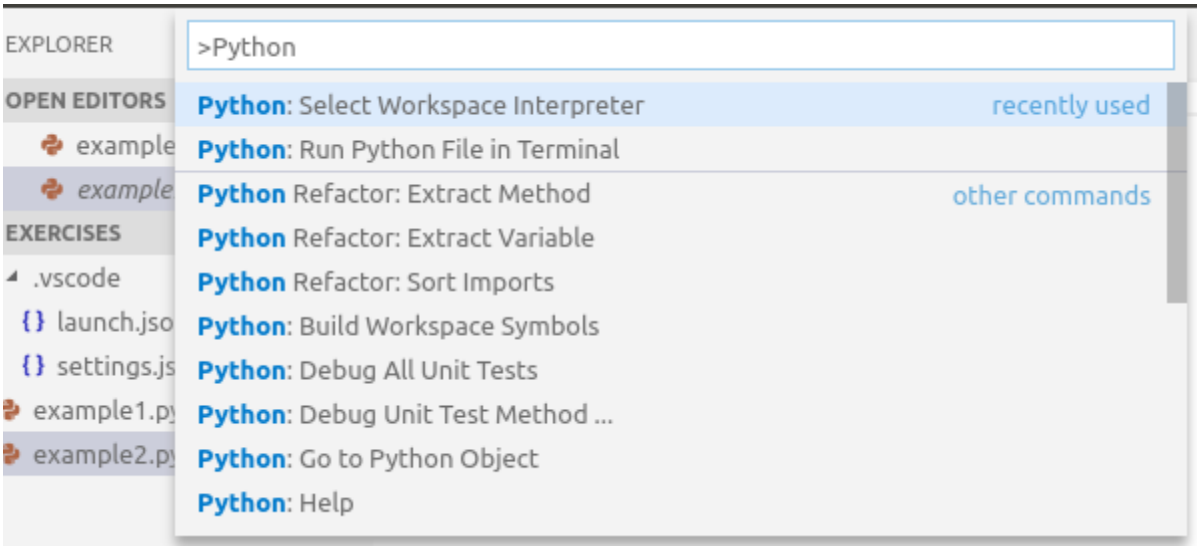
You can find useful information on this tool here[25]. Please spend some time having a look at that page. Once you are done with it you can close this window pressing on the "x".

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



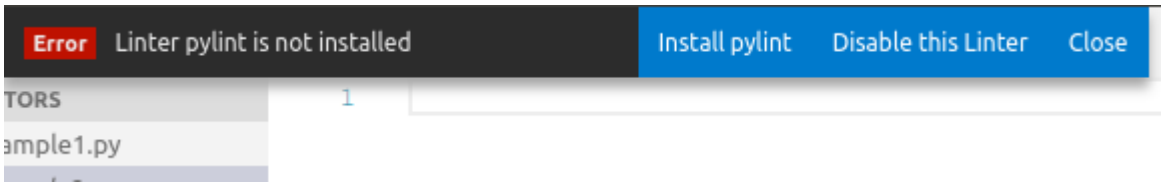
Finally, select the python version you want to use (e.g. Python3.x).

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.

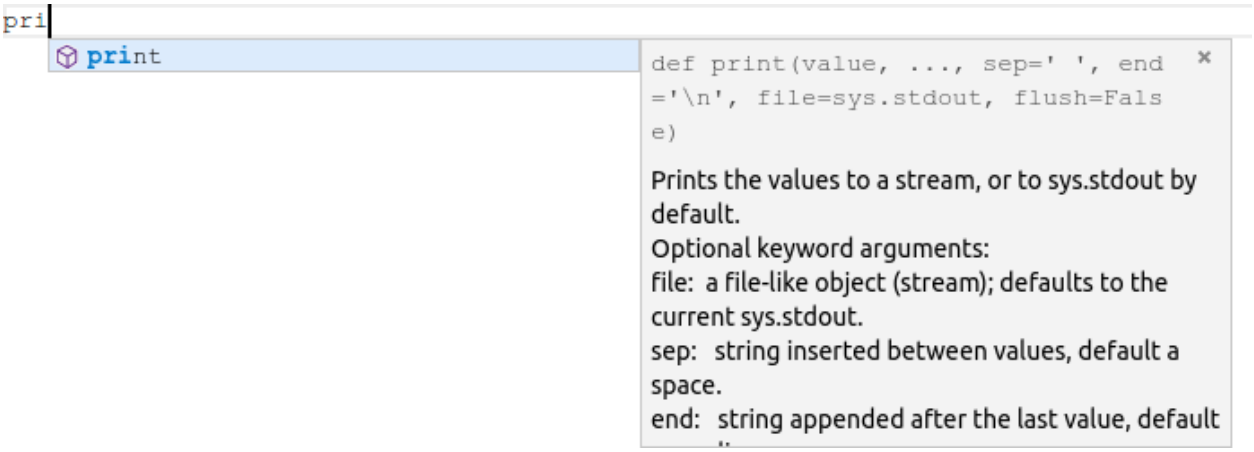
Note

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

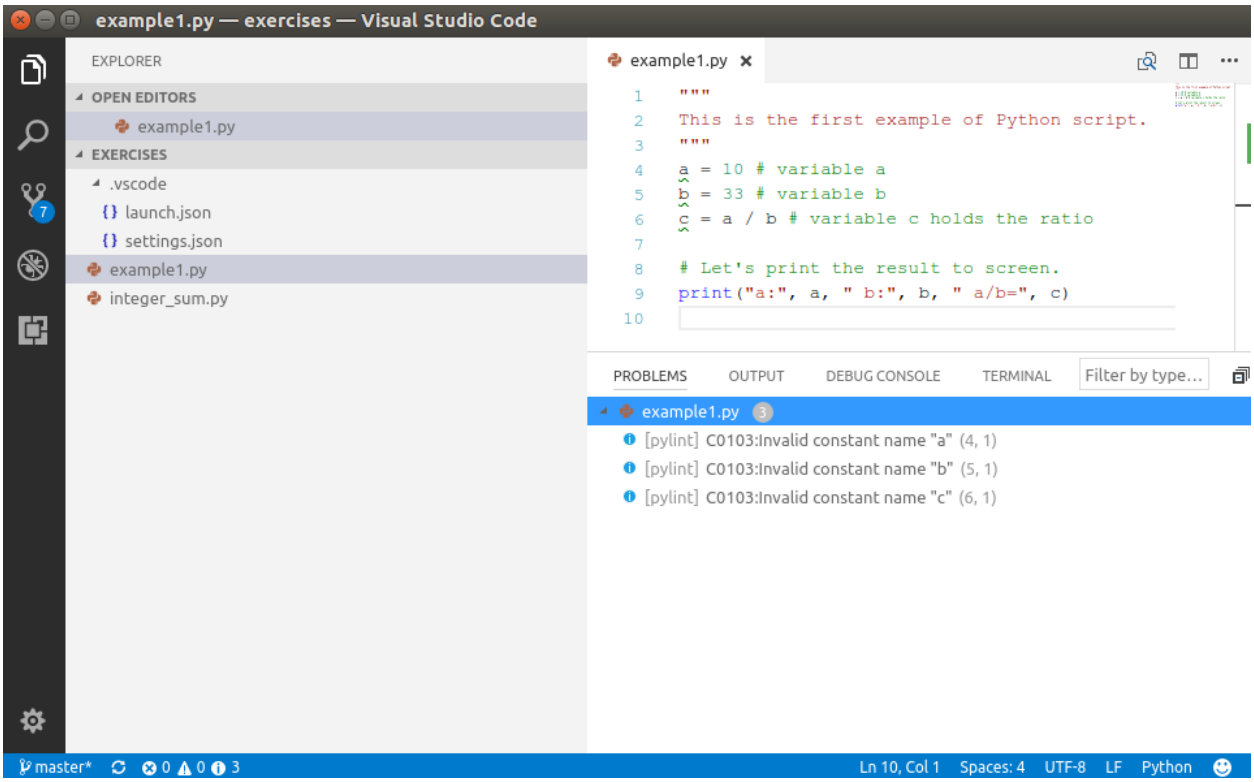
Please remember to comment your code, as it helps readability and will make your life easier when you have to modify or just understand the code you wrote some time in the past.

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)

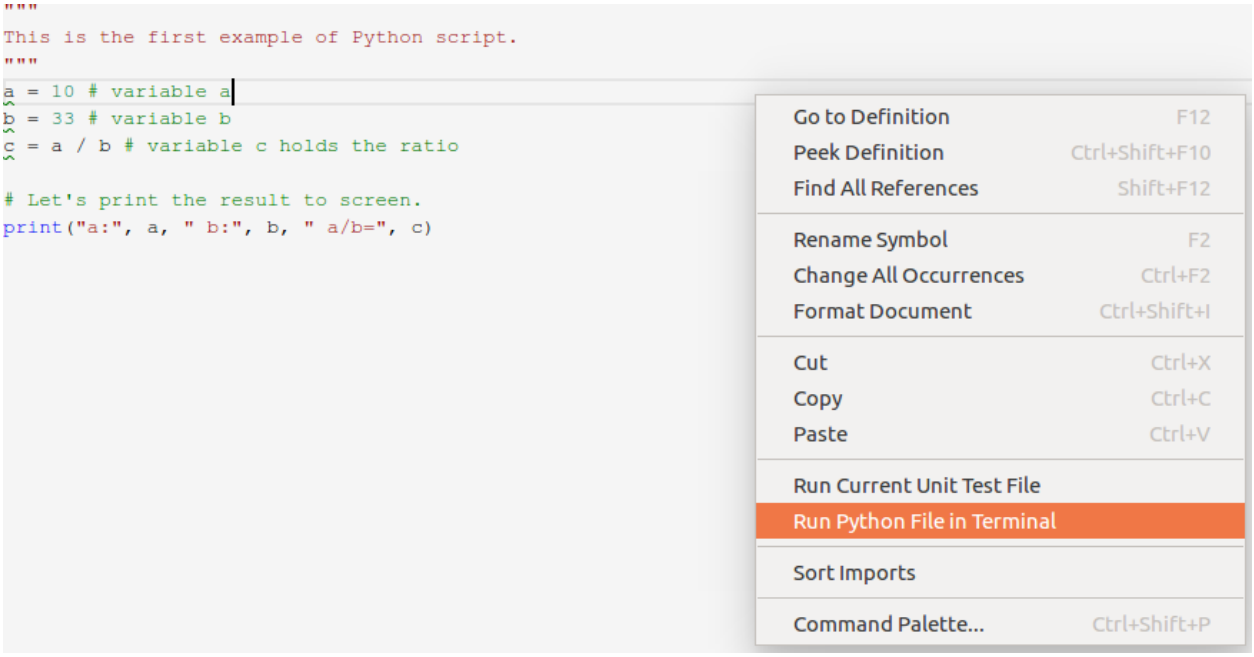
[pylint] C0103:Invalid constant name "a" (4, 1)

[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.

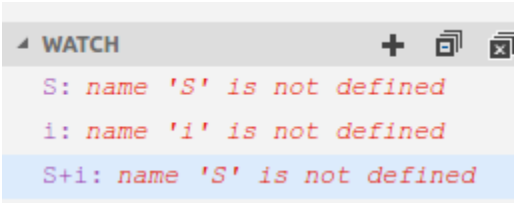
Info

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:

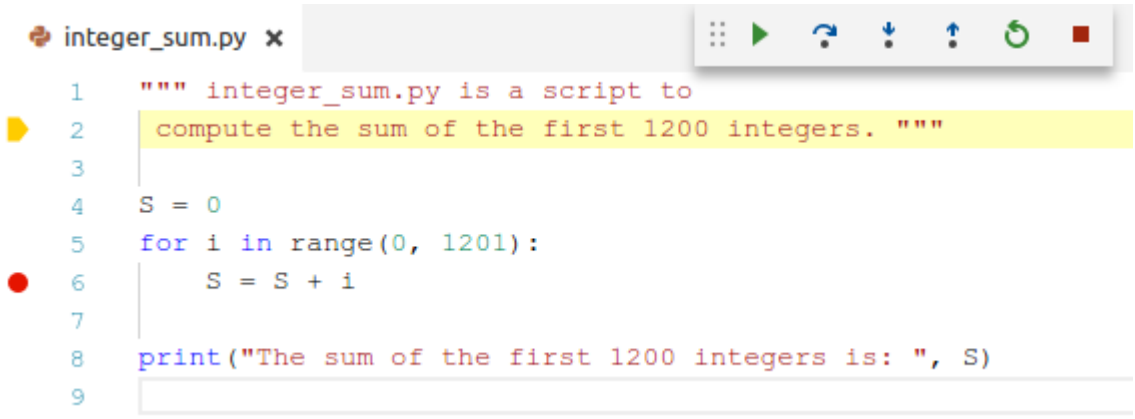
We will use it shortly, but before that, let's focus on what we want to track. On the left hand side of the main panel, a *Watch* panel appeared. This is where we need to add the things we want to monitor as the execution of the program goes. With respect to the code written above, we are interested 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:



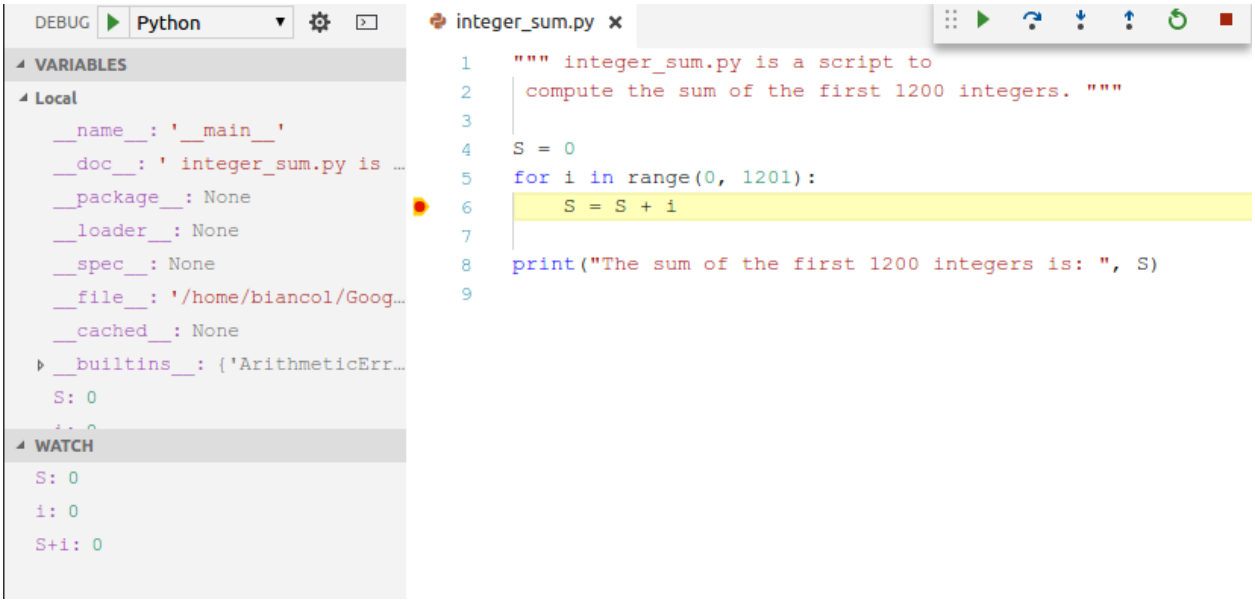
do not worry about the message "*name X is not defined*", this is normal as no execution has taken place yet and the interpreter still does not know the value of these expressions.

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.

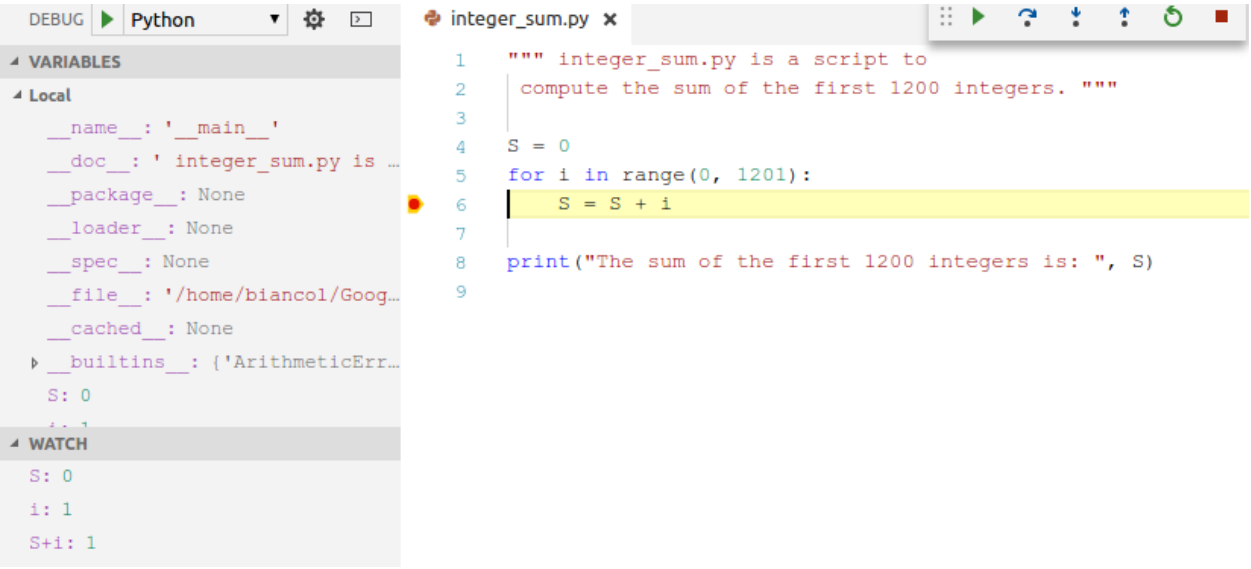
A small reddish dot should appear, place the mouse over the correct line (e.g. the line corresponding to $S = S + I$ and click to add the breakpoint (a red dot should appear once you click).



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):

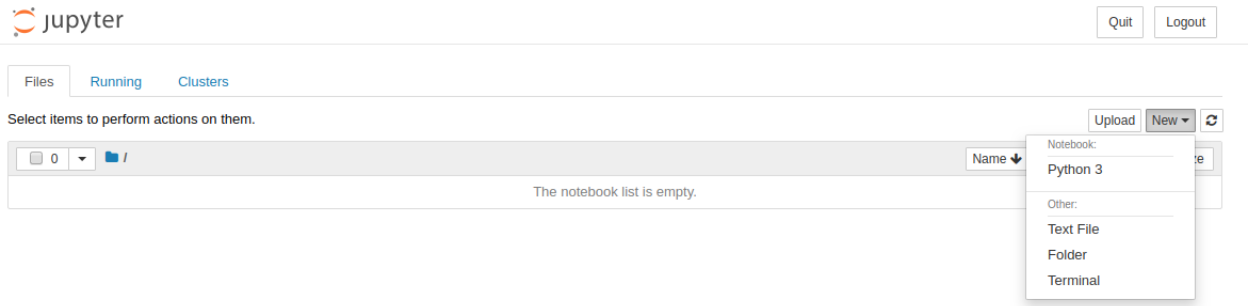
```
python3 -m pip install jupyter
```

you can find more information here[33]

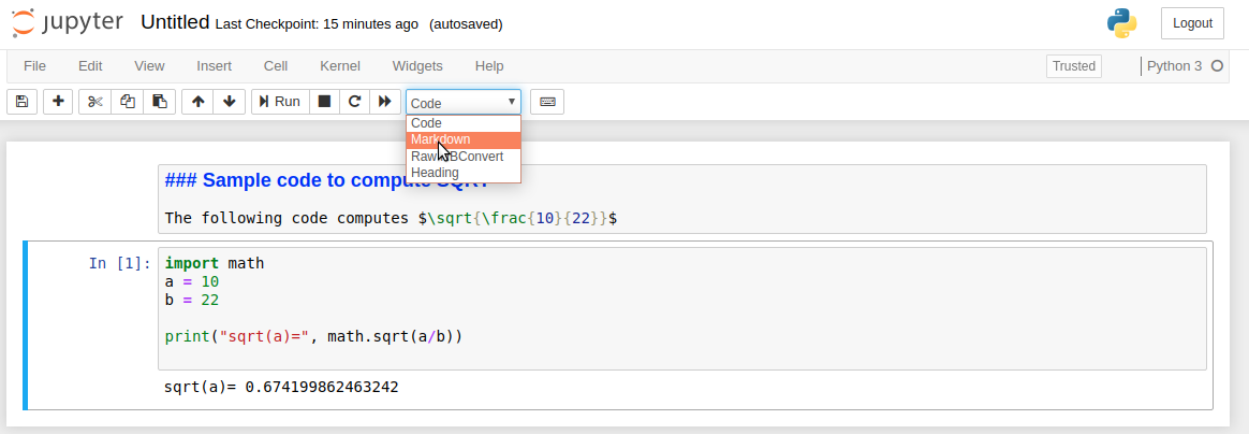
Upon successful installation, you can run it with:

```
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**:

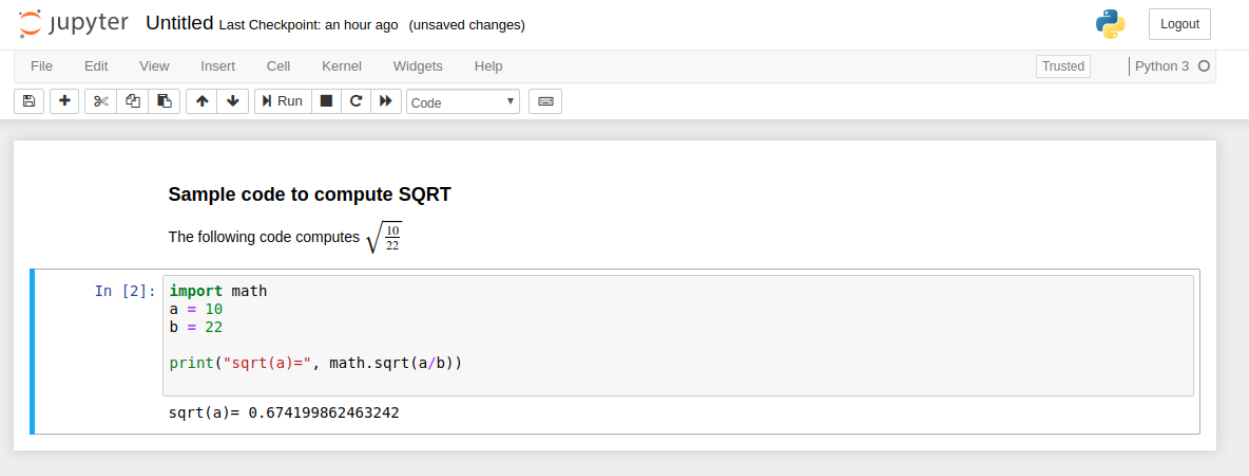


and then you can start adding cells (i.e. containers of code and text). The type of each cell is specified by selecting the cell and selecting the right type in the dropdown list:



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.

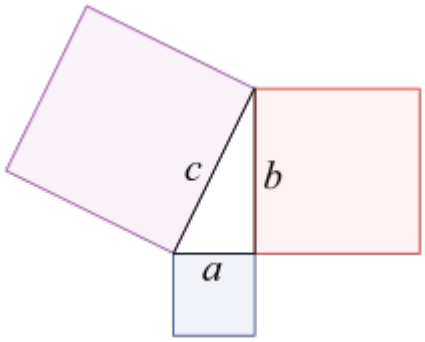
Show/Hide Solution

[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

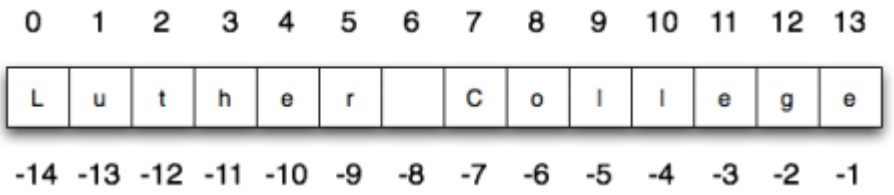
Example A tandem repeat is a short sequence of DNA that is repeated several times in a row. Let's create a string representing the tandem repeat of the motif "ATTCG" repeated 5 times. What is the length of the whole repetitive region? Is the motif "TCGAT" (m1) present in the region? The motif "TCCT" (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)

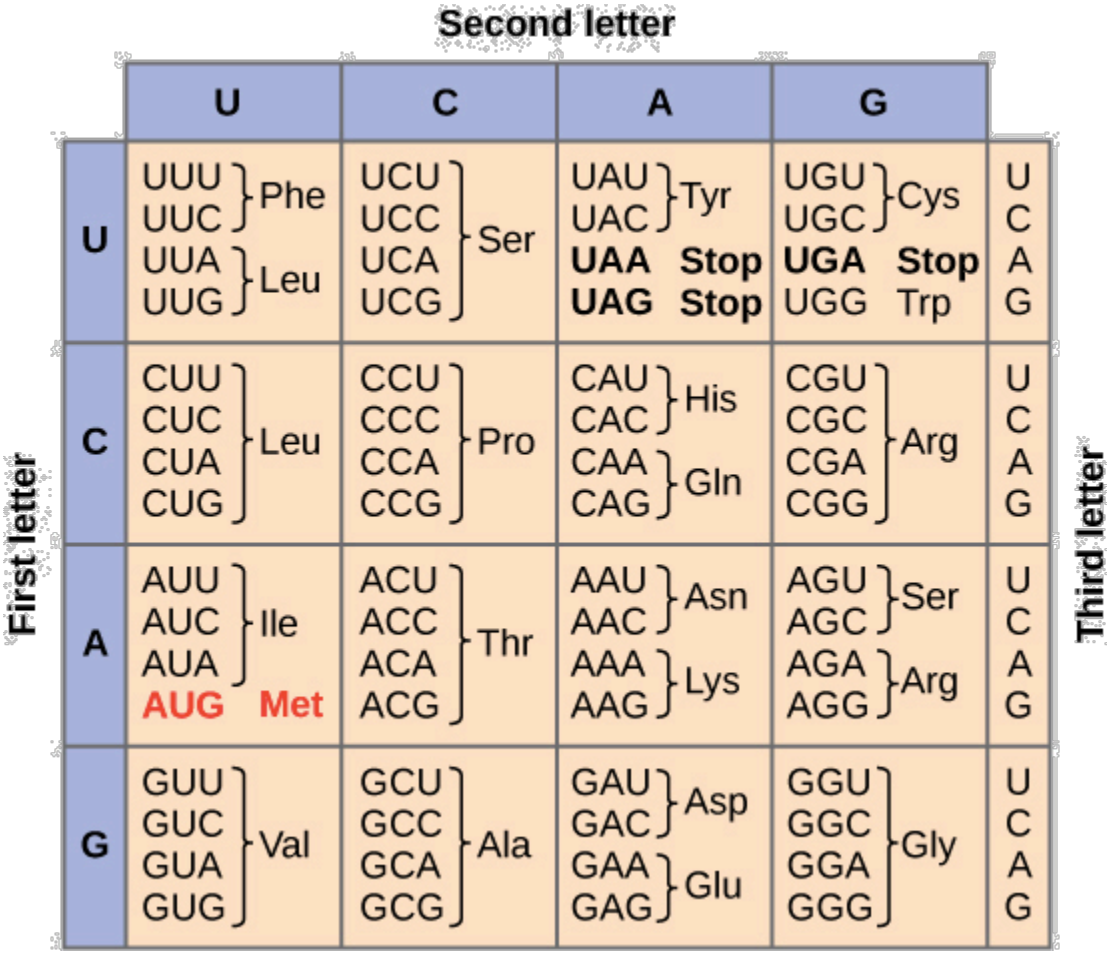
[35] http://www.ensembl.org/Homo_sapiens/Gene/Summary?g=ENSG00000083093;r=16:23603160-23641310

(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

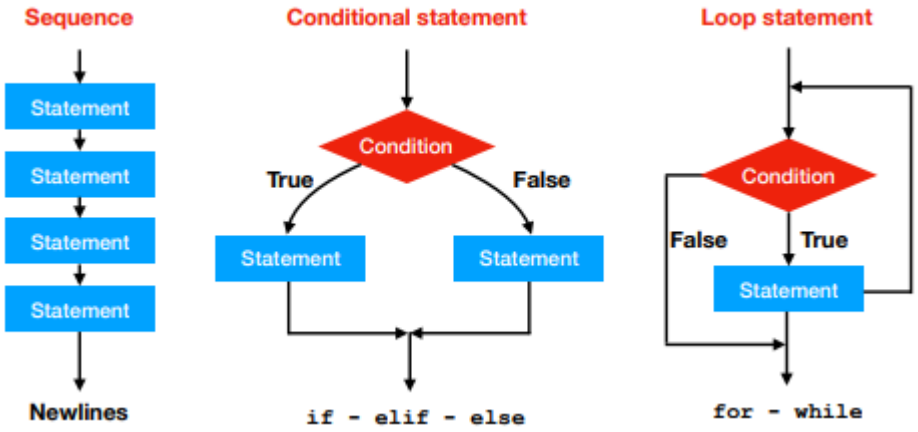
In this practical we will work with conditionals (branching) and loops.

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

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 and 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

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

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

```
+
++
+++
```

(continues on next page)

(continued from previous page)

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

Show/Hide Solution

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.

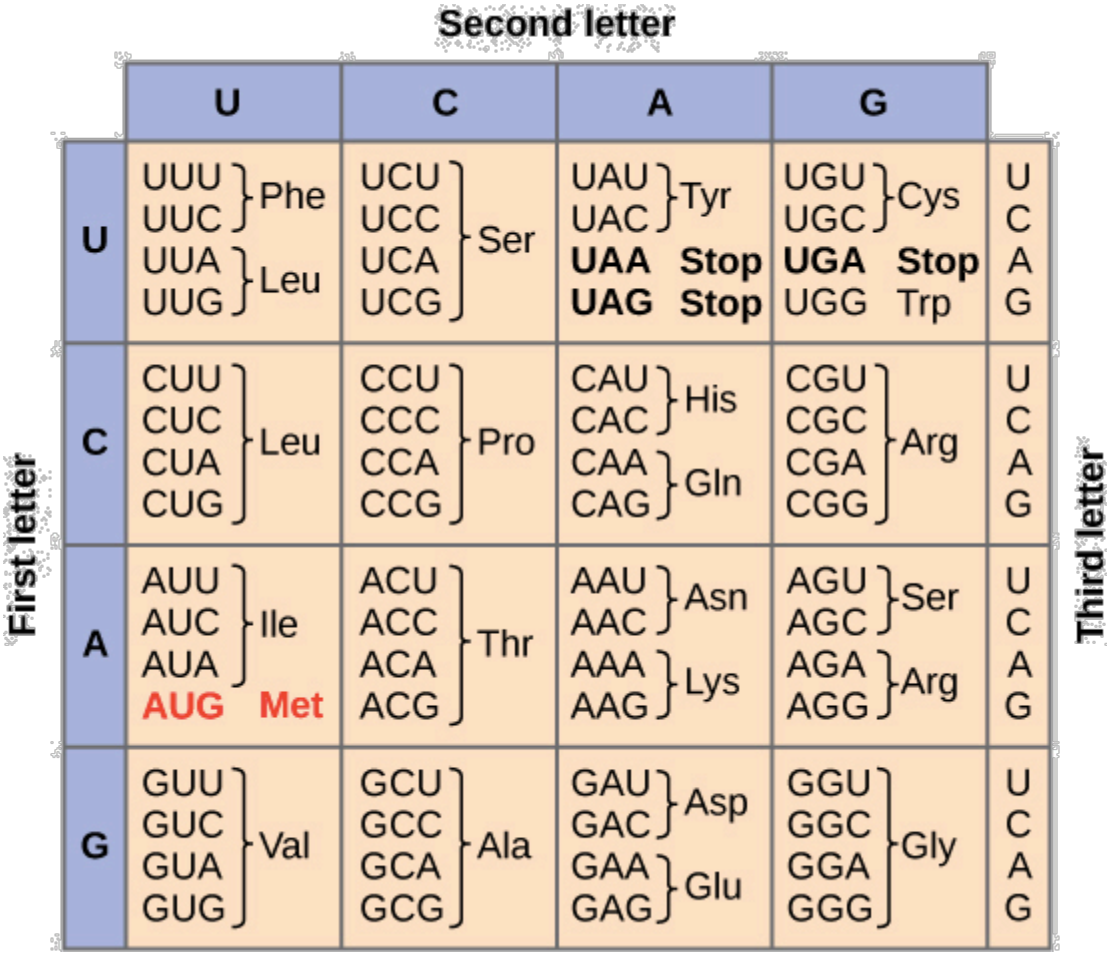
Output example: base: C index: 15 qual:1 phred: 16.

Show/Hide Solution

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

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

CHAPTER

SIX