SoftPython

Introductive guide to data cleaning and analysis with Python 3

David Leoni, Alessio Zamboni, Marco Caresia

Sep 24, 2020

CONTENTS

Prefazione

Introductive guide to coding, data cleaning and analysis for Python 3, with many worked exercises.

WARNING: THIS ENGLISH VERSION IS IN-PROGRESS, COMPLETION IS DUE BY END OF 2020

ITALIAN VERSION IS HERE: it.softpython.org[1]

Nowadays, more and more decisions are taken upon factual and objective data. All disciplines, from engineering to social sciences, require to elaborate data and extract actionable information by analysing heterogenous sources. This book of practical exercises gives an introduction to coding and data processing using Python[2], a programming language popular both in the industry and in research environments.

News

Old news: link

[1] https://it.softpython.org
[2] https://www.python.org

CHAPTER

ONE

PANORAMICA

1.1 Intended audience

This book can be useful for both novices who never really programmed before, and for students with more techical background, who a desire to know about about data extraction, cleaning, analysis and visualization (among used frameworks there are Pandas, Numpy and Jupyter editor). We will try to process data in a practical way, without delving into more advanced considerations about algorithmic complexity and data structures. To overcome issues and guarantee concrete didactical results, we will present step-by-step tutorials.

1.2 Contents

Overview

- Approach and goals
- Resources

1.2.1 A - Foundations

1.3 Authors

David Leoni (main author): Software engineer specialized in data integration and semantic web, has made applications in open data and medical in Italy and abroad. He frequently collaborates with University of Trento for teaching activities in various departments. Since 2019 is president of CoderDolomiti Association, where along with Marco Caresia manages volunteering movement CoderDojo Trento to teach creative coding to kids. Email: david.leoni@unitn.it Website: davidleoni.it[3]

Marco Caresia (2017 Autumn Edition assistent @DISI, University of Trento): He has been informatics teacher at Scuola Professionale Einaudi of Bolzano. He is president of the Trentino Alto Adige Südtirol delegatioon of the Associazione Italiana Formatori and vicepresident of CoderDolomiti Association.

Alessio Zamboni (2018 March Edition assitent @Sociology Department, University of Trento): Data scientist and software engineer with experience in NLP, GIS and knowledge management. Has collaborated to numerous research projects, collecting experinces in Europe and Asia. He strongly believes that *'Programming is a work of art'*.

Massimiliano Luca (2019 summer edition teacher @Sociology Department, University of Trento): Loves learning new technilogies each day. Particularly interested in knowledge representation, data integration, data modeling and computational social science. Firmly believes it is vital to introduce youngsters to computer science, and has been mentoring at Coder Dojo DISI Master.

[3] https://davidleoni.it

1.4 License

The making of this website and related courses was funded mainly by Department of Information Engineering and Computer Science (DISI)[4], University of Trento, and also Sociology[5] and Mathematics[6] departments.

Technical notes: all website pages are easily modifiable Jupyter notebooks, that were converted to web pages using NBSphinx[8] using template Jupman[9]. Text sources are on Github at address https://github.com/DavidLeoni/softpython-en

1.5 Acknowledgments

We thank in particular professor Alberto Montresor of Department of Information Engineering and Computer Science, University of Trento to have allowed the making of first courses from which this material was born from, and the project Trentino Open Data (dati.trentino.it[10]) for the numerous datasets provided.

Other numerous intitutions and companies that in over time contributed material and ideas are cited in this page

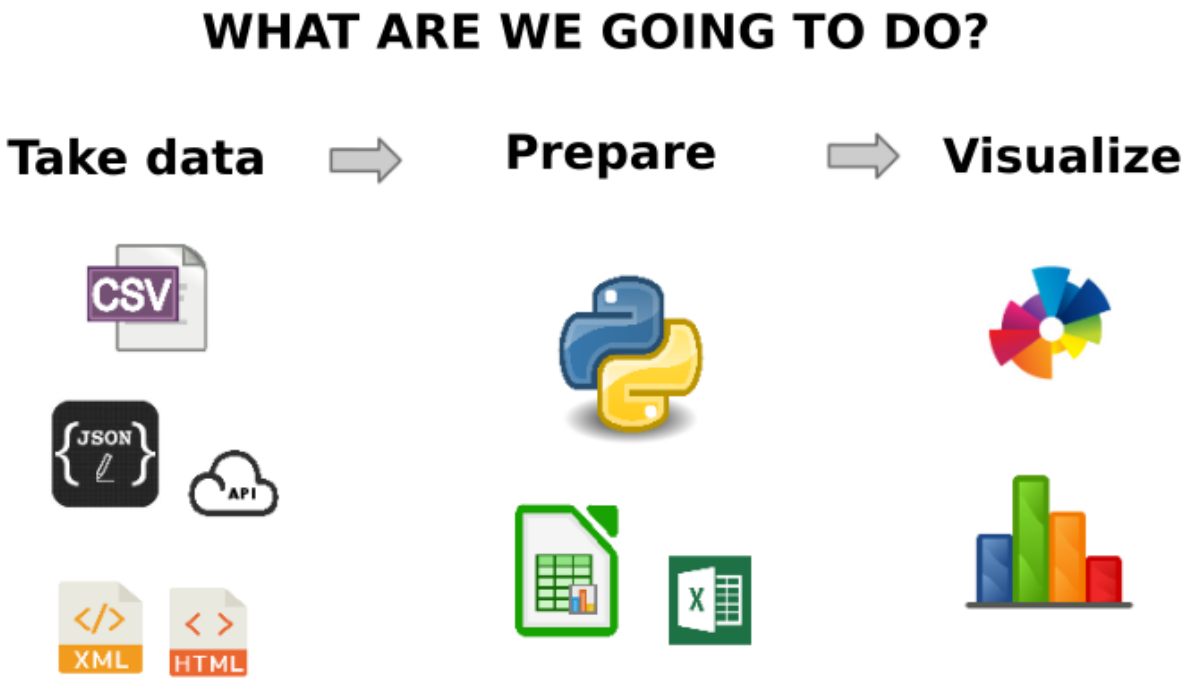
[4] https://www.disi.unitn.it
[5] https://www.sociologia.unitn.it/en
[6] https://www.maths.unitn.it/en
[7] https://en.softpython.org/index.html#Authors
[8] https://nbsphinx.readthedocs.io
[9] https://github.com/DavidLeoni/jupman
[10] https://dati.trentino.it

CHAPTER

TWO

OVERVIEW

To start with we will spend a couple of words on the approach and the goals of the book, then we will deep dive into the code.



2.1 Chapters

The tutorial mostly deal with fundamentals of PYthon 3, data analysis (intended more like raw data processing than statistics) and some applications (dashboard, database, ..)

What are *not* about:

- object oriented programming theory
- algorithms, computational complexity
- performance
  - no terabytes of data …
- advanced debugging (pdb)

- testing is only mentioned
- machine learning
- web development is only mentioned

2.2 Why Python?

- **Easy** enough to start with
- **Versatile**, very much used for
  - scientific calculus
  - web applications
  - scripting
- **widespread** both in the industry and research environments
  - Tiobe[11] Index
  - popularity on Github[12]
- **Licence** open source & business friendly[13]
  - translated: you can sell commercial products based on Python without paying royalties to its authors

[11] https://www.tiobe.com/tiobe-index/
[12] https://madnight.github.io/githut/#/pull_requests/2020/1
[13] https://docs.python.org/3/license.html

2.3 References

Altro materiale: Citiamo i seguenti due libri, entrambi dall'approccio discorsivo e gratuiti disponibili sia online che in pdf.

2.3.1 Part A Resources

2.3.2 Think Python, by Allen Downey

- Talks a lot, step by step, good for beginners
- online[14]
- printed[15]
- PDF[16]
- Interactive edition[17]

License: Creative Commons CC BY Non Commercial 3.0[18] as reported in the original page[19]

2.3.3 Dive into Python 3, by Mark Pilgrim

- More practical, contains more focused tutorials (i.e. manage XML files)
- online version[20]
- printed[21]
- zip offline[22]
- PDF[23]

Licence: Creative Commons By Share-alike 3.0[24] as reported at the bottom of book website[25]

[14] http://www.greenteapress.com/thinkpython/html/
[15] https://www.amazon.it/Think-Python-Like-Computer-Scientist/dp/1491939362/ref=sr_1_1?ie=UTF8&qid=1537163819&sr=8-1&keywords=think+python
[16] http://greenteapress.com/thinkpython2/thinkpython2.pdf
[17] https://runestone.academy/runestone/static/thinkcspy/index.html
[18] http://creativecommons.org/licenses/by-nc/3.0/deed.it
[19] http://greenteapress.com/wp/think-python-2e/
[20] http://www.diveintopython3.net/
[21] http://www.amazon.com/gp/product/1430224150?ie=UTF8&tag=diveintomark-20&creativeASIN=1430224150
[22] https://github.com/diveintomark/diveintopython3/zipball/master
[23] https://github.com/downloads/diveintomark/diveintopython3/dive-into-python3.pdf
[24] http://creativecommons.org/licenses/by-sa/3.0/
[25] http://www.diveintopython3.net/

2.4 Approach and goals

If you have troubles with programming basics:

- **Exercise difficulty**: ⊛ , ⊛⊛
- Read SoftPython - Parte A - Foundations[26]
- Other useful stuff can be found in the book 'Think Python'

If you already know how to program:

- **Exercise difficulty**: ⊛⊛⊛, ⊛⊛⊛⊛
- Read Python Quick Intro[27] and then go directly to Part B - Data Analysis
- other useful things can be found in the book **Dive into Python 3**

2.5 Doesn't work, what should I do?

While programming you will surely encounter problems, and you will stare at mystierious error messages on the screen. The purpose of this book is not to give a series of recipes to learn by heart and that always work, as much as guide you moving first steps in Python world with some ease. So, if something goes wrong, do not panic and try following this list of steps that might help you. Try following the proposed order:

1. If in class, ask professor (if not in class, see last two points).
2. If in class, ask the classmate who knows more
3. Try finding the error message on Google
    - remove names or parts too specific of your program, like line numbers, file names, variable names
    - Stack overflow[28] is your best friend
4. Look at Appendix A - Debug from the book Think Python[29]
    - Syntax errors[30]
        - I keep making changes and it makes no difference.[31]
    - Runtime errors[32]
        - My program does absolutely nothing.[33]
        - My program hangs.[34]
        - Infinite Loop[35]
        - Infinite Recursion[36]
        - Flow of Execution[37]

[26] https://en.softpython.org/index.html#A---Foundations
[27] https://en.softpython.org/quick-intro/quick-intro-sol.html
[28] https://stackoverflow.com
[29] http://greenteapress.com/thinkpython2/html/thinkpython2021.html
[30] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#235
[31] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec236
[32] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec237
[33] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec238
[34] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec239
[35] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec240
[36] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec2241
[37] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec242

  - When I run the program I get an exception[38]
  - I added so many print statements I get inundated with output[39]
- Semantic errors[40]
  - My program doesn't work[41]
  - ve got a big hairy expression and it doesn't do what I expect.[42]
  - ve got a function that doesn't return what I expect.[43]
  - I'm really, really stuck and I need help.[44]
  - No, I really need help.[45]

5. Gather some courage and ask on a public forum, like Stack overflow or python-forum.io - see *how to ask questions*.

2.5.1 How to ask questions

IMPORTANT

If you want to ask written questions on public chat/forums (i.e. like python-forum.io[46] DO FIRST READ the forum rules (see for example How to ask Smart Questions[47]

In substance, you are always asked to clearly express the problem circumstances, putting an explicative title to the post /mail and showing you spent some time (at least 10 min) trying a solution on your own. If you followed the above rules, and by misfortune you still find programmers who use harsh tones, just ignore them.

2.6 Installation and tools

- If you still haven't installed Python3 and Jupyter, have a look at *Installation*

2.7 Let's start !

- **If you already have some programming skill**: you can look Python quick start
- **If you don't have programming skills**: got to Tools and scripts[48]

[38] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec243
[39] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec244
[40] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec245
[41] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec246
[42] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec247
[43] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec248
[44] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec249
[45] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec250
[46] https://python-forum.io/index.php
[47] https://python-forum.io/misc.php?action=help&hid=19
[48] https://en.softpython.org/tools/tools-sol.html

CHAPTER

THREE

INSTALLATION

We will see how to install Python, additional Python libraries, Jupyter notebook and managing virtual environments.

3.1 Installing Python

There are varioues ways to install Python 3 and its modules: there is the official 'plain' Python distribution but also package managers (i.e. Anaconda) or preset environments (i.e. Python(x,y)) which give you Python plus many various modules. Once completed the installation, Python 3 contains a command `pip` (sometimes called `pip3` in Python 3), which allows to install afterwards other packages you may need.

The best way to choose what to install depends upon which operating system you have and what you intend to do with it. In this book we will use Python 3 and scientific packages, so we will try to create an environment to support this scenario.

Fast online alternatives

If you have difficulties installing and you are anxious to try Python, you can program directly online with Python 3 on repl.it[49]

Another useful resource is Python Tutor[50], which allows to execute one instruction per time while offering a useful visualization of what happens 'under the hood'.

You can also try the online demo of Jupyter[51], but it is not always available.

Whichever online environment you choose, always remember to check it is Python **3** !

Attention: before installing random stuff from the internet, read carefully this guide

We tried to make it generic enough, but we couldn't test all various cases so problems may arise depending on your particular configuration.

Attention: do not mix different Python distribution for the same version !

Given the wide variety of installation methods and the fact Python is available in laready many programs, it might be you already have installed Python without even knowing it, maybe in version 2, but we need the 3! Overlayinh several Python environments with the same version may cause problems, so in case of doubt ask somebody who knows more!

[49] https://repl.it/languages/python
[50] http://pythontutor.com/visualize.html#py=3
[51] http://try.jupyter.org

3.1.1 Windows installation

For Windows, we suggest to install the distribution Anaconda for Python 3.8[52] or greater, which, along with the native Python package manager `pip`, also offers the more generic command line package manager `conda`.

Once installed, verify it is working like this:

1. click on the Windows icon in the lower left corner and search for 'Anaconda Prompt'. It should appear a console where to insert commands, with written something like `C:\Users\David>`. NOTE: to launch Anaconda commands, only use this special console. If you use the default Windows console (`cmd`), Windows will not be able to find Python.
2. In Anaconda console, type:

```
conda list
```

It should appear a list of installed packages, like

```
# packages in environment at C:\Users\Jane\AppData\Local\Continuum\Anaconda3:
#
alabaster                 0.7.7                    py35_0
anaconda                  4.0.0                np110py35_0
anaconda-client           1.4.0                    py35_0
...
numexpr                   2.5                  np110py35_0
numpy                     1.10.4                   py35_0
odo                       0.4.2                    py35_0
...
yaml                      0.1.6                         0
zeromq                    4.1.3                         0
zlib                      1.2.8                         0
```

3. Try Python3 by typing in the Anaconda console:

```
C:>  python
```

It should appear something like:

```
Python 3.6.3 (default, Sep 14 2017, 22:51:06)
MSC v.1900 64 bit (Intel)[GCC 5.4.0 20160609] on win64
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Attention: with Anaconda, you must write `python` instead of `python3` !

If you installed Anaconda for Python3, it will automatically use the correct Python version by simply writing `python`. If you write `python3` you will receive an error of file not found !

Attention: if you have Anaconda, always use `conda` to install Python modules ! So if in next tutorials you se written `pip3 install whatever`, you will instead have to use `conda install whatever`

[52] https://www.anaconda.com/download/#windows

3.1.2 Installation Mac

To best manage installed app on Mac independently from Python, usually it is convenient to install a so called *package manager*. There are various, and one of the most popular is Homebrew[53]. So we suggest to first install Homebrew and then with it you can install Python 3, plus eventually other components you might need. As a reference, for installation we took and simplified this guide by Digital Ocean](https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-local-programming-environment-on-macos)

Attention: check if you already have a package manager !

If you already have installed a package manager like for example Conda (in *Anaconda* distribution), *Rudix*, *Nix*, *Pkgsrc*, *Fink* o *MacPorts*, maybe Homebrew is not needed and it's better to use what you already have. In these cases, it may be worth asking somebody who knows more ! If you already have *Conda/Anaconda*, it can be ok as long as it is for Python 3.

— 1 Open the Terminal

MacOS terminal is an application you can use to access command line. As any other application, it's available in *Finder*, navigation in *Applications* folder, and the in the folder *Accessories*. Frotm there, double click on the *Terminal* to open it as any other app. As an alternative, you can use *Spotlight* by pressing *Command* and *Space* to find the Terminal typing the name in the bar that appears.

- 2 Install Homebrew by executing in the terminal this command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

— 3 Add `/usr/local/bin` to `PATH`

In this passage with an unsettling name, once Homebrew installation is completed, you will make sure that apps installed with Homebrew shall always be used instead of those Mac OS X may automatically select.

— 3.1 Open a new Terminal.

— 3.2 From within the terminal, digit the command

```
ls -a
```

You will see the list of all files present in the home folder. In these files, verify if a file exists with the following name: `.profile` (note the dot at the beginning):

- If it exists, go to following step
- If it doesn't exist, to create it type the following command:

```
touch  $HOME/.profile
```

— 3.3 Open with text edit the just created file `.profile` giving the command:

```
open -e  $HOME/.profile
```

— 3.4 In text edit, add to the end of the file the following line:

```
export PATH=/usr/local/bin:$PATH
```

[53] https://brew.sh/

— 3.5 Save and close both Text Edit and the Terminal

— 4 Verify Homebrew is correctly installed, by typing in a new Terminal:

```
brew doctor
```

If there aren't updates to do, the Terminal should show:

```
Your system is ready to brew.
```

Otherwise, you might see a warning which suggest to execute another command like `brew update` to ensure the Homebrew installation is updated.

— 5. Install python3 (Remember the '3' !):

```
brew install python3
```

Along with python 3, Homebrew will also install the internal package manager of Python `pip3` which we will use in the following.

— 6 Verify Python3 is correctly installed. By executing this command the writing `"/usr/local/bin/python3"` should appear:

```
which python3
```

After this, try to launch

```
python3
```

You should see something similar:

```
Python 3.6.3 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on mac
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To exit Python, type `exit()` and press Enter.

3.1.3 Linux installation

Luckily, all Linux distributions are already shipped with package managers to easily install applications.

- If you have Ubuntu:
  1. follow the guide of Dive into Python 3, chapter 0 - Installare Python[54] in particular by going to the subsection installing in Ubuntu Linux[55]
  2. after completing the guide, install also `python3-venv`:

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

- If you *don't* have Ubuntu, read this note[56] and/or ask somebody who knows more.

To verify the installation, try to run from the terminal

[54] https://diveintopython3.problemsolving.io/installing-python.html
[55] https://diveintopython3.problemsolving.io/installing-python.html#ubuntu
[56] https://diveintopython3.problemsolving.io/installing-python.html#other

```
python3
```

You should see something like this:

```
Python 3.6.3 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3.2 Installing packages

You can extend Python by installing several free packages. The best way to do it varies according to the operating system and the installed package manager.

ATTENTION: We will be using *system commands*. If you see `>>>` in the command line, it means you are inside Python interpreter and you must first exit: to do it, type `exit()` and press Enter.

In what follows, to check if everything is working, you can substitute `PACKAGENAME` with `requests` which is a module for the web.

If you have Anaconda:

- click on Windows icon in the lower left corner and search `Anaconda Prompt`. A console should appear where to instert commands, with something written like `C:\Users\David>`. (NOTE: to run commands in Anaconda, use only this special console. If you use the default Windows console (cmd), Windows, will not be able to find Python)
- In the console type `conda install PACKAGENAME`

If you have Linux/Mac open the Terminaland give this command (`--user` install in your home):

- `python3 -m pip install --user PACKAGENAME`
- **NOTE:** If you receive errors which tell you the command `python3` is not found, remove the `3` after `python`

INFO: there is also a system command `pip` (or `pip3` according to your system). You can directl call it with `pip install --user PACKAGENAME`

Instead, we install instead with commands like `python3 -m pip install --user PACKAGENAME` for uniformity and to be sure to install packages for Python 3 version

---

3.3 Jupyter Notebook

3.3.1 Run Jupyter notebook

A handy editor you can use for Python is Jupyter[57]:

- If you installed Anaconda, you should already find it in the system menu and also in the Anaconda Navigator.
- If you didn't install Anaconda, try searching in the system menu anyway, maybe by chance it was already installed

[57] http://jupyter.org/

- If you can't find it in the system menu, you may anyway from command line

Try this:

```
jupyter notebook
```

or, as alternative,

```
python3 -m notebook
```

ATTENTION: `jupyter` is NOT a Python command, it is a *system* command.

If you see written >>> on command line it means you must first exit Python insterpreter by writing 'exit()` and pressing Enter !

ATTENTION: If Jupyter is not installed you will see error messages, in this case don't panic and *go to installation*.

A browser should automatically open with Jupyter, and in the console you should see messages like the following ones. In the browser you should see the files of the folders from which you ran Jupyter.

If no browser starts but you see a message like the one here, then copy the address you see in an internet browser, preferebly Chrome, Safari or Firefox.

```
$ jupyter notebook
[I 18:18:14.669 NotebookApp] Serving notebooks from local directory: /home/da/Da/prj/
↪softpython/prj
[I 18:18:14.669 NotebookApp] 0 active kernels
[I 18:18:14.669 NotebookApp] The Jupyter Notebook is running at: http://localhost:
↪8888/?token=49d4394bac446e291c6ddaf349c9dbffcd2cdc8c848eb888
[I 18:18:14.669 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
[C 18:18:14.670 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=49d4394bac446e291c6ddaf349c9dbffcd2cdc8c848eb888
```

ATTENTION 1: in this case the address is `http://localhost:8888/?token=49d4394bac446e291c6ddaf349c9dbffcd2cdc8c848eb888`, but youts will surely be different!

ATTENTION 2: While Jupyter server is active, you can't put commands in the terminal !

In the console you see the server output of Jupyter, which is active and in certain sense 'it has taken control' of the terminal. This means that if you write some commands inside the terminal, these **will not** be executed!

3.3.2 Saving Jupyter notebooks

You can save the current notebook in Jupyter by pressing `Control-S` while in the browser.

ATTENTION: DO NOT OPEN THE SAME DOCUMENT IN MANY TABS !!

Be careful to not open the same notebook in more the one tab, as modifications in different tabs may overwrite at random ! To avoid these awful situations, make sure to have only one tab per document. If you accidentally open the same notebook in different tabs, just close the additional tab.

Automated savings

Notebook changes are automatically saved every few minutes.

3.3.3 Turning off Jupyter server

Before closing Jupyter server, remember to save in the browser the notebooks you modified so far.

To correctly close Jupyter, *do not* brutally close the terminal, Instead, from the the terminal where you ran Jupyter, hit `Control-c`, a question should appear to which you should answer `y` (if you don't answer in 5 seconds, you will have to hit `control-c` again).

```
Shutdown this notebook server (y/[n])? y
[C 11:05:03.062 NotebookApp] Shutdown confirmed
[I 11:05:03.064 NotebookApp] Shutting down kernels
```

3.3.4 Navigating notebooks

(Optional) To improve navigation experience in Jupyter notebooks, you may want to install some Jupyter extension, like `toc2` which shows paragraph headers in the sidebar. To install:

Install the Jupyter contrib extensions[58]:

1a. If you have Anaconda: Open Anaconda Prompt, and type:

```
conda install -c conda-forge jupyter_contrib_nbextensions
```

1b. If you don't have Anaconda: Open the terminal and type:

```
python3 -m pip install --user jupyter_contrib_nbextensions
```

2. Install in Jupyter:

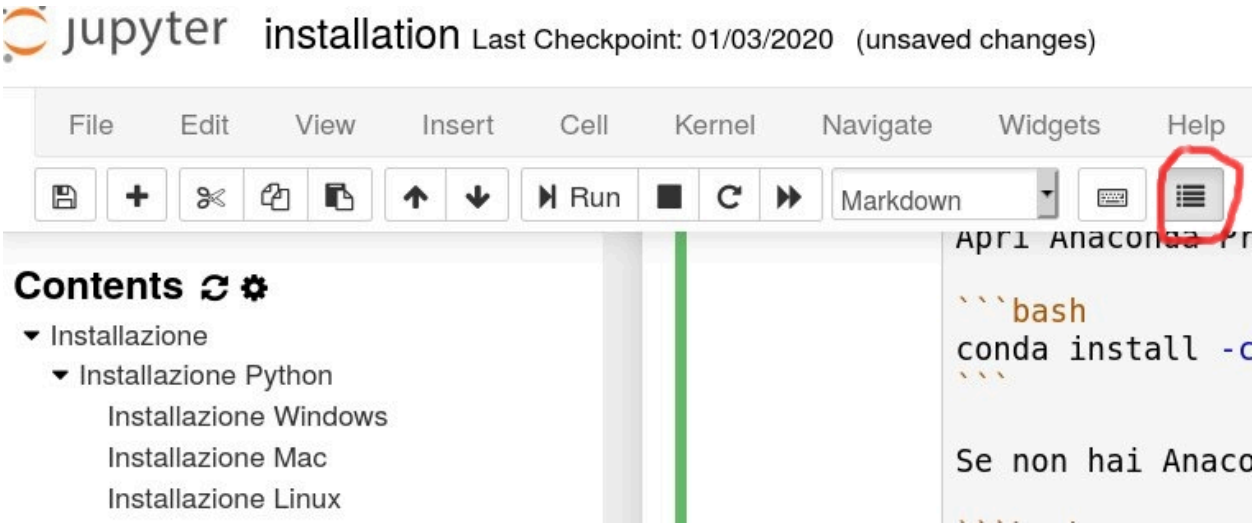
```
jupyter contrib nbextension install --user
```

3. Enable extensions:

```
jupyter nbextension enable toc2/main
```

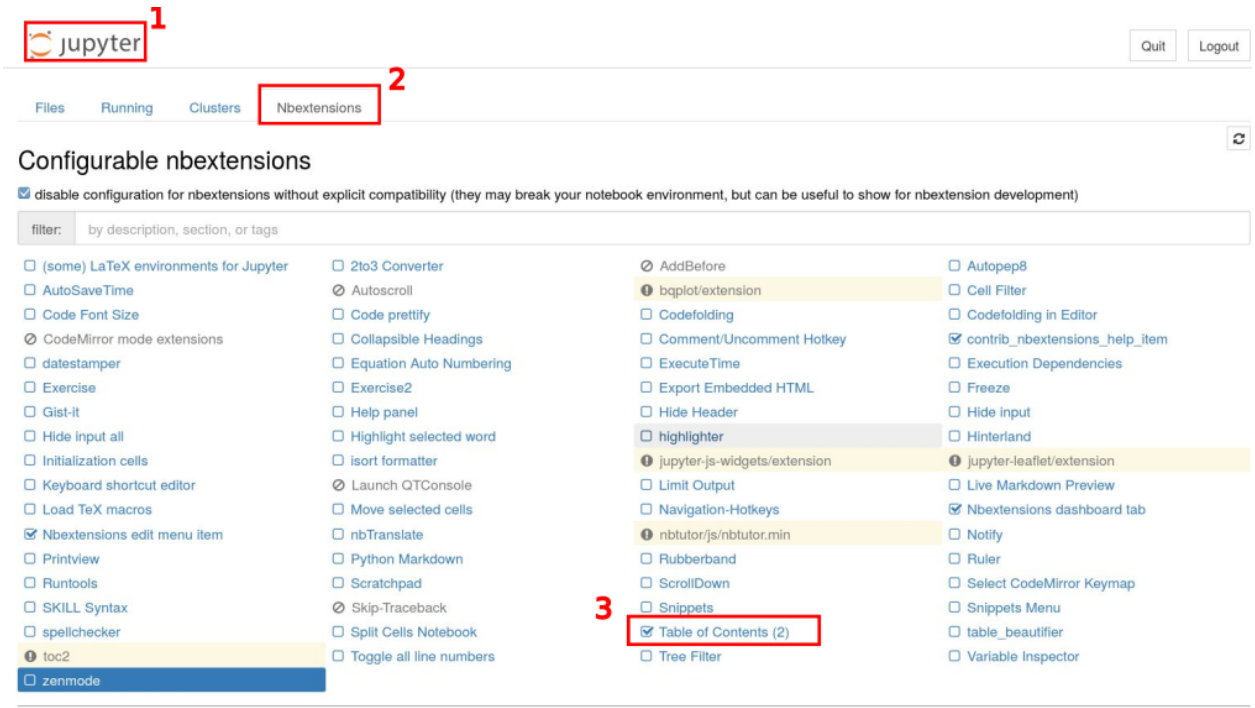
[58] https://github.com/ipython-contrib/jupyter_contrib_nbextensions

Once installed: To see table of contents in a document you will have to press a list button on the right side of the toolbar:



If by chance you don't see the button:

1. go to main Jupyter interface
2. check `Nbextensions` tab
3. make sure `Table of Contents(2)` is enabled
4. Close Jupyter, reopen it, go to a notebook, you should finally see the button



3.3.5 Installing Jupyter notebook - all operating systems

If you didn't manage to *find and/or start Jupyter*, probably it means we need to install it!

You may try installing Jupyter with `pip` (the native package manager of Python)

To install, run this command:

```
python3 -m pip install --user jupyter -U
```

Once installed, follow the section

Una volta installato, segui la sezione *Run Jupyter Notebook*

ATTENTION: you DON'T need to install Jupyter inside *virtual environments* You can consider Jupyter as a system-level application, which should be independent from virtual environments. If you are inside a virtual environment (i.e. the command line begins with a writing in parenthesis like (myprj)``), exit the environment by typeing``deactivate`)

HELP: if you have trouble installing Jupyter, while waiting for help you can always try the online demo version[59] (note: it's not always available) or Google Colab[60]

3.4 Projects with virtual environments

WARNING: If these are your first steps with Python, you can skip this section.

You should read it if you have already done personal projects with Python that you want to avoid compromising, or when you want to make a project to ship to somebody.

When we start a new project with Python, we usually notice quickly that we need to extend Python with particular libraries, like for example to draw charts. Not only that, we might also want to install Python programs which are not written by us and they might as well need their peculiar libraries to work.

Now, we could install all these extra libraries in a unique cauldron for the whole computer, but each project may require its specific versions of each library, and sometimes it might not like versions already installed by other projects. Even worse, it might automatically update packages used by old projects, preventing old code from working anymore. So it is PRACTICALLY NECESSARY to separate well each project and its dependencies from those of other projects: for this purpose you can create a so-called *virtual environment* .

[59] https://try.jupyter.org/
[60] http://colab.research.google.com/

3.4.1 Creating virtual environments

- **If you installed Anaconda**, to create virtual environments you can use its package manager `conda`. Supposing we want to call our project `myprj` (but it could be any name), to put into a folder with the same name `myprj`, we can use this command to create a virtual environment:

```
conda create -n myprj
```

  The command might require you to download packages, you can safely confirm.

- **If you *don't have* Anaconda installed**, to create virtual environments it's best to use the native Python module `venv`:

```
python3 -m venv myprj
```

Both methods create the folder `myprj` and fill it with all required Pyhton files to have a project completely isolated from the rest of the computer. But now, how can we tell Python we want to work right with that project? We must *activate* the environment as follows.

3.4.2 Activate a virtual environment

To activate the virtual environment, we must use different commands according to our operating system (but always from the terminal)

Activate environment in Windows with Anaconda:

```
activate myprj
```

Linux & Mac (without Anaconda):

```
source myprj/bin/activate
```

Once the environment is active, in the command prompt we should see the name of that environment (in this case `myprj`) between round parenthesis at the beginning of the row:

```
(myprj) some/current/folder >
```

The prefix lets us know that the environment `myprj` is currently active, so Python comands we will use all use the settings and libraries of that environment.

Note: inside the virtual environment, we can use the command `python` instead of `python3` and `pip` instead of `pip3`

Deactivate an environment:

Write in the console the command `deactivate`. Once the environment is deactivated, the environment name `(myprj)` at the beginning of the prompt should disappear.

3.4.3 Executing environments inside Jupyter

As we said before, Jupyter is a system-level application, so there should be one and only one Jupyter. Nevertheless, during Jupyter execution, we might want to execute our Python commands in a particular Python environment. To do so, we must configure Jupyter so to use the desired environment. In Jupyter terminology, the configurations are called *kernel*: they define the programs launched by Jupyter (be they Python versions or also other languages like R). The current kernel for a notebook is visibile in the right-upper corner. To select a desired kernel, there are several ways:

With Anaconda

Jupyter should be available in the Navigator. If in the Navigator you enable an environment (like for example Python 3), when you then launch Jupyter and create a notebook you should have the desired environment active, or at least be able to select a kernel with that environment.

Without Anaconda

In this case, the procedure is a little more complex:

— 1 From the terminal þactivate your environment](#Activate-a-virtual-environment)

— 2 Create a Jupyter kernel:

```
python3 -m ipykernel install --user --name myprj
```

NOTE: here `myprj` is the name of the *Jupter kernel.* We use the same name of the environment only for practical reasons.

— 3 Deactivate your environment, by launching

```
deactivate
```

From now on, every time you run Jupyter, if everything went well under the `Kernel` menu in the notebook you should be able to select the kernel just created (in this example, it should have the name `myprj`)

NOTE: the passage to create the kernel must be done only once per project

---

NOTE: you don't need to activate the environment before running Jupyter!

During the execution of Jupyter simply select the desired kernel. Nevertheless, it is convenient to execute Jupyter from the folder of our virtual environment, so we will see all the project files in the Jupyter home.

3.5 Further readings

Go on with the page Tools and scripts[61] to learn how to use other editors and Python architecture.

[61] https://en.softpython.org/tools/tools-sol.html

CHAPTER

FOUR

A - FOUNDATIONS

4.1 Tools and scripts

4.1.1 Download exercises zip

Browse files online[62]

REQUISITES:

- **Having Python 3 and Jupyter installed:** if you haven't already, see Installation[63]

4.1.2 Python interpreter

In these tutorials we will use extensively the notebook editor Jupyter, because it allows to comfortably execute Python code, display charts and take notes. But if we want only make calculations it is not mandatory at all!

The most immediate way (even if not very practical) to execute Python things is by using the *command line* interpreter in the so-called *interactive mode,* that is, having Python to wait commands which will be manually inserted one by one. This usage *does not* require Jupyter, you only need to have installed Python. Note that in Mac OS X and many linux systems like Ubuntu, Python is already installed by default, although sometimes it might not be version 3. Let's try to understand which version we have on our system.

Let's open system console

Open a console (in Windows: system menu -> Anaconda Prompt, in Mac OS X: run the Terminal)

In the console you find the so-called *prompt* of commands. In this *prompt* you can directly insert commands for the operating system.

WARNING: the commands you give in the prompt are commands in the language of the operating system you are using, **NOT** Python language !!!!!

In Windows you should see something like this:

[62] https://github.com/DavidLeoni/softpython-en/tree/master/tools
[63] https://en.softpython.org/installation.html

```
C:\Users\David>
```

In Mac / Linux it could be something like this:

```
david@my-computer:~$
```

Listing files and folders

In system console, try for example to

Windows: type the command `dir` and press Enter

Mac or Linux: type the command `ls` and press Enter.

A listing with all the files in the current folder should appear. In my case appears a list like this:

LET ME REPEAT: in this context `dir` and `ls` are commands of *the operating system,* **NOT** of Python !!

Windows:

```
C:\Users\David> dir
Arduino                     gotysc                          program.wav
a.txt                       index.html                      Public
CART                        java0.log                       RegDocente.pdf
backupsys                   java1.log
BaseXData                   java_error_in_IDEA_14362.log
```

Mac / Linux:

```
david@david-computer:~$ ls
Arduino                          gotysc                           program.wav
a.txt                            index.html                       Public
CART                             java0.log                       ␣
↪RegistroDocenteStandard(1).pdf
backupsys                        java1.log                       ␣
↪RegistroDocenteStandard.pdf
BaseXData                        java_error_in_IDEA_14362.log
```

Let's launch the Python interpreter

In the opened system console, simply type the command `python`:

WARNING: If Python does not run, try typing `python3` with the `3` at the end of `python`

```
C:\Users\David> python
```

You should see appearing something like this (most probably won't be exactly the same). Note that Python version is contained in the first row. If it begins with `2.`, then you are not using the right one for this book - in that case try exiting the interpreter (*see how to exit*) and then type `python3`

```
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on windows
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

CAREFUL about the triple greater-than >>> at the beginning!

The triple greater-than >>> at the start tells us that differently from before now the console is expecting commands *in Pyhton language*. So, the system commands we used before (`cd`, `dir`, ...) will NOT work anymore, or will give different results!

Now the console is expecting Python commands, so try inserting `3 + 5` and press Enter:

WARNING DO NOT type >>>, only type the command which appears afterwards!

```
>>> 3 + 5
```

The writing `8` should appear:

```
8
```

Beyond calculations, we might tell PYthon to print something with the function `print("ciao")`

```
>>> print("ciao")
ciao
```

Exiting the interpreter

To get out from the Python interpreter and go back to system prompt (that is, the one which accepts `cd` and `dir`/`ls` commands), type the Python comand `exit()`

After you actually exited the Python interpreter, the triple >>> should be gone (you should see it at the start of the line)

In Windows, you should see something similar:

```
>>> exit()
C:\Users\David>
```

in Mac / Linux it could be like this:

```
>>> exit()
david@my-computer:~$
```

Now you might go back to execute commands for the operating system like `dir` and `cd`:

Windows:

```
C:\Users\David> dir
Arduino                     gotysc                          program.wav
a.txt                       index.html                      Public
CART                        java0.log                       RegDocente.pdf
backupsys                   java1.log
BaseXData                   java_error_in_IDEA_14362.log
```

Mac / Linux:

```
david@david-computer:~$ ls
Arduino                             gotysc                              program.wav
a.txt                               index.html                          Public
CART                                java0.log                           ␣
↪RegistroDocenteStandard(1).pdf
backupsys                           java1.log                           ␣
↪RegistroDocenteStandard.pdf
BaseXData                           java_error_in_IDEA_14362.log
```

4.1.3 Modules

Python Modules are simply text files which have the extension **.py** (for example `my_script.py`). When you write code in an editor, as a matter of fact you are implementing the corresponding module.

In Jupyter we use notebook files with the extension `.ipynb`, but to edit them you necessarily need Jupyter.

With `.py` files (alse said *script*) we can instead use any text editor, and we can then tell the interpreter to execute that file. Let's see how to do it.

Simple text editor

1. With a text editor (*Notepad* in Windows, or *TextEdit* in Mac Os X) creates a text file, and put inside this code

```
x = 3
y = 5
print(x + y)
```

2. Let's try to save it - it seems easy, but it is often definitely not, so read carefully!

WARNING: when you are saving the file, **make sure the file have the extension `.py` !!**

Let's suppose to create the file `my_script.py` inside a folder called `CART`:

- **WINDOWS**: if you use *Notepad*, in the save window you have to to set *Save as* to *All files* (otherwise the file will be wrongly saved like `my_script.py.txt` !)
- **MAC**: if you use *TextEdit,* before saving click *Format* and then *Convert to format Only text:* **if you forget this passage, TextEdit in the save window will not allow you to save in the right format and you will probably end up with a file .rtf which we are not interested in**

3. Open a console (in Windows: system menu -> Anaconda Prompt, in Mac OS X: run the Terminal)

the console opens the so-called *commands prompt*. In this *prompt* you can directly enter commands for the operating system (see *previous paragraph*

WARNING: the commands you give in the prompt are commands in the language of the operating system you are using, **NOT** Python language !!!!!

In Windows you should see something like this:

```
C:\Users\David>
```

In Mac / Linux it could be something like this:

```
david@my-computer:~$
```

Try for example to type the command `dir` (or `ls` for Mac / Linux) which shows all the files in the current folder. In my case a list like this appears:

LET ME REPEAT: in this context `dir` / `ls` are commands of the *operating system,* **NOT** Python.

```
C:\Users\David> dir
Arduino                     gotysc                          program.wav
a.txt                       index.html                      Public
MYFOLDER                    java0.log                       RegDocente.pdf
backupsys                   java1.log
BaseXData                   java_error_in_IDEA_14362.log
```

If you notice, in the list there is the name MYFOLDER, where I put `my_script.py`. To *enter* the folder in the *prompt,* you must first use the operating system command `cd` like this:

4. To enter a folder called MYFOLDER, type `cd MYFOLDER`:

```
C:\Users\David> cd MYFOLDER
C:\Users\David\MYFOLDER>
```

What if I get into the wrong folder?

If by chance you enter the wrong folder, like `DUMBTHINGS`, to go back of one folder, type `cd ..` (NOTE: `cd` is followed by one space and TWO dots `..` *one after the other*)

```
C:\Users\David\DUMBTHINGS> cd ..
C:\Users\David\>
```

5. Mae sure to be in the folder which contains `my_script.py`. If you aren't there, use commands `cd` and `cd ..` like above to navigate the folders.

Let's see what present in MYFOLDER with the system command `dir` (or `ls` if in Mac/Linux):

LET ME REPEAT: inthis context `dir` (or `ls` is a command of the *operating system,* **NOT** Python.

```
C:\Users\David\MYFOLDER> dir

my_script.py
```

`dir` is telling us that inside `CART` there is our file `my_script.py`

6. From within `MYFOLDER`, type `python my_script.py`

```
C:\Users\David\CART>python my_script.py
```

WARNING: if Python does not run, try typing `python3 my_script.py` with `3` at the end of `python`

If everything went fine, you should see

```
8
C:\Users\David\MYFOLDER>
```

WARNING: After executing a script this way, the console is awaiting new *system* commands, **NOT** Python commands (so, there shouldn't be any triple greater-than >>>)

IDE

in these tutorial we work on Jupyter notebooks with extension `.ipynb`, but to edit long `.py` files it's more convenient to use more traditional editors, also called IDE *(Integrated Development Environment).* For Python we can use Spyder[64], Visual Studio Code[65] or PyCharme Community Edition[66].

Differently from Jupyter, these editors allow more easily code *debugging* and *testing.*

Let's try Spyder, which is the easiest - if you have Anaconda, you find it available inside Anaconda Navigator.

INFO: Whenever you run Spyder, it might ask you to perform an upgrade, in these cases you can just click No.

In the upper-left corner of the editor there is the code of the file `.py` you are editing. Such files are also said *script.* In the lower-right corner there is the console with the IPython interpreter (which is the same at the heart of Jupyter, here in textual form). When you execute the script, it's like inserting commands in that interpreter.

- To execute the whole script: press `F5`
- To execute only the current line or the selection: press `F9`
- To clear memory: after many executions the variables in the memory of the interpreter might get values you don't expect. To clear the memory, click on the gear to the right of the console, and select *Restart kernel*

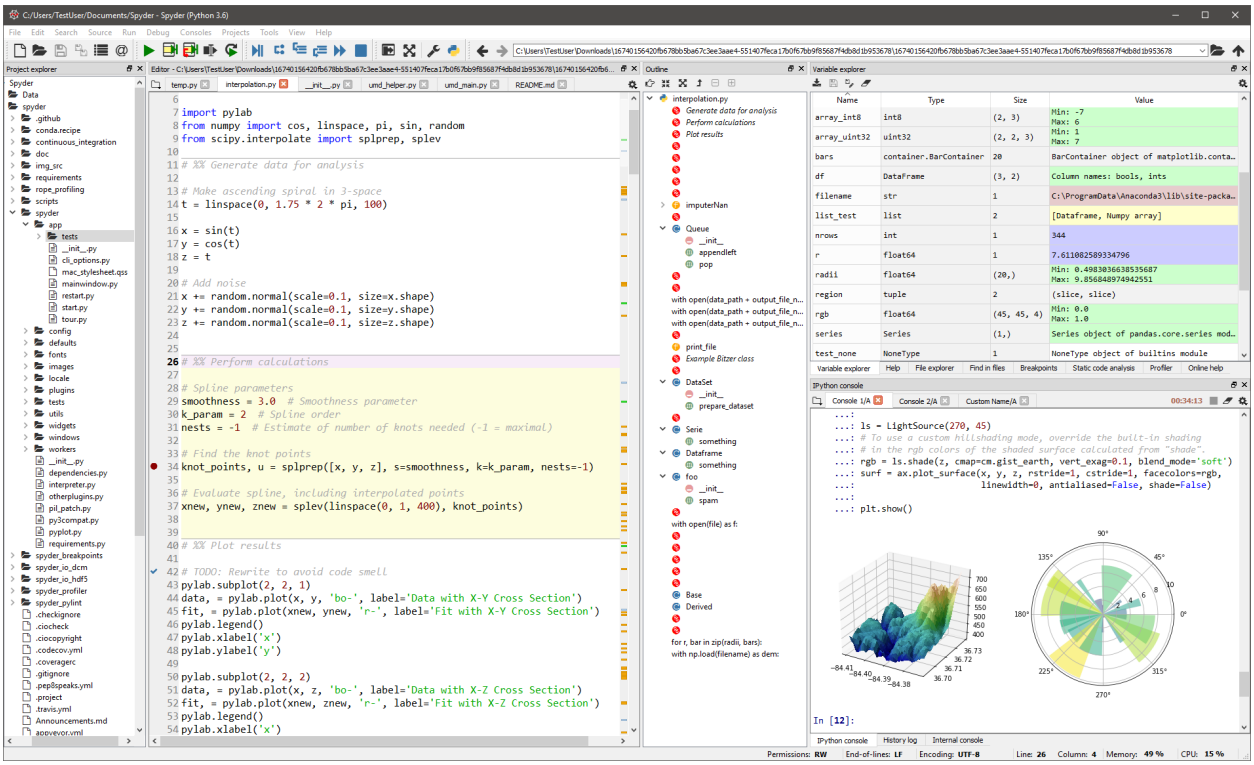
EXERCISE: do some test, taking the file `my_script.py` we created before:

```
x = 3
y = 5
print(x + y)
```

- once the code is in the script, hit `F5`
- select only `print(x+y)` and hit F9
- select only `x=3` and hit F9
- click on th gear the right of the console panel, and select *Restart kernel,* then select only `print(x+y)` and hit F9. What happens?

Remember that if the memory of the interpreter has been cleared with *Restart kernel*, and then you try executing a code row with variables defined in lines which were not exectued before, Python will not know which variables you are referring to and will show a `NameError`.

[64] https://www.spyder-ide.org/
[65] https://code.visualstudio.com/Download
[66] https://www.jetbrains.com/pycharm/download/



4.1.4 Jupyter

Jupyter is an editor that allows to work on so called *notebooks*, which are files ending with the extension `.ipynb`. They are documents divided in cells where in each cell you can insert commands and immediately see the respective output. Let's try opening this.

1. Unzip exercises zip in a folder, you should obtain something like this:

```
tools
    tools-sol.ipynb
    tools.ipynb
    jupman.py
```

WARNING: To correctly visualize the notebook, it MUST be in the unzipped folder.

2. open Jupyter Notebook in that folder. Two things should appear, first a console and then a browser. The browser should show a list with two files: borwser the list and open the notebook `tools.ipynb`

WARNING: open the one WITHOUT the `-sol` at the end!

Seeing now the solutions is too easy ;-)

WARNING: if you don't find Jupyter / something doesn't work, have a look at installation[67]

[67] https://en.softpython.org/installation.html#Jupyter-Notebook

3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells.Exercises are graded by difficulty, from one star ⊛ to four ⊛⊛⊛⊛

> **WARNING**: In this book we use **ONLY PYTHON 3**
>
> If by chance you obtain weird behaviours, check you are using PYthon 3 and not 2. If by chance by typing `python` your operating system runs python 2, try executing the third by typing the command `python3`

Useful shortcuts:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- when something seem wrong in computations, try to clean memory by running `Kernel->Restart and Run all`

EXERCISE: Let's try inserting a PYthon command: type in the cell below here `3 + 5`, then while in that cell press special keys `Control+Enter`. As a result, the number `8` should appear

```
[ ]:
```

EXERCISE: with Python we can write comments by starting a row with a sharp `#`. Like before, type in the next cell `3 + 5` but this time type it in the row under the writing `# write here`:

```
[2]: # write here
```

EXERCISE: In every cell Jupyter only shows the result of last executed row. Try inserting this code in the cell below and execute by pressing `Control+Enter`. Which result do you see?

```
3 + 5
1 + 1
```

```
[3]: # write here
```

EXERCISE: Let's try now to create a new cell.

- While you are with curson the cell, press `Alt+Enter`. A new cell should be created after the current one.
- In the cell just created, insert `2+3` and press `Shift+Enter`. What happens to the cursor? Try the difference swith `Control+Enter`. If you don't understand the difference, try pressing many times `Shit+Enter` and see what happens.

Printing an expression

Let's try to assign an expression to a variable:

[4]:
```
coins = 3 + 2
```

Note the assignment by itself does not produce any output in the Jupyter cell. We can ask Jupyter the value of the variable by simply typing again the name in a cell:

[5]:
```
coins
```

[5]:
```
5
```

The effect is (almost always) the same we would obtain by explictly calling the function `print`:

[6]:
```
print(coins)
```

```
5
```

What's the difference? For our convenience Jupyter will directly show the result of the last executed expression in the cell, but only the last one:

[7]:
```
coins = 4
2 + 5
coins
```

[7]:
```
4
```

If we want to be sure to print both, we need to use the function `print`:

[8]:
```
coins = 4
print(2 + 5)
print(coins)
```

```
7
4
```

Furthermore, the result of last expression is shown only in Jupyter notebooks, if you are writig a normal `.py` script and you want to see results you must in any case use `print`.

If we want to print more expressions in one row, we can pass them as different parameters to `print` by separating them with a comma:

[9]:
```
coins = 4
print(2+5,coins)
```

```
7 4
```

To `print` we can pass as many expressions as we want:

[10]:
```
coins = 4
print(2 + 5, coins, coins*3)
```

```
7 4 12
```

If we also want to show some text, we can write it by creating so-called *strings* between double quotes (we will see strings much more in detail in next chapters):

[11]:
```
coins = 4
print("We have", coins, "golden coins, but we would like to have double:", coins * 2)
```

```
We have 4 golden coins, but we would like to have double: 8
```

QUESTION: Have a look at following expressions, and for each one of them try to guess the result it produces. Try cerifying your guesses both in Jupyter and another editor of files `.py` like Spyder:

1.
```
x = 1
x
x
```

2.
```
x = 1
x = 2
print(x)
```

3.
```
x = 1
x = 2
x
```

4.
```
x = 1
print(x)
x = 2
print(x)
```

5.
```
print(zam)
print(zam)
zam = 1
zam = 2
```

6.
```
x = 5
print(x,x)
```

7.
```
x = 5
print(x)
print(x)
```

8.
```
carpets = 8
length = 5
print("If I have", carpets, "carpets in sequence I walk for", carpets * length,
↪"meters.")
```

9.
```
carpets = 8
length = 5
print("If", "I","have", carpets, "carpets","in", "sequence","I", "walk", "for",␣
↪carpets * length, "meters.")
```

Exercise - Castles in the air

Given two variables

```
castles = 7
dirigibles = 4
```

write some code to print:

```
I've built 7 castles in the air
I have 4 steam dirigibles
I want a dirigible parked at each castle
So I will buy other 3 at the Steam Market
```

- **DO NOT** put numerical constants in your code like `7`, `4` or `3`! Write generic code which only uses the provided variables.

[12]:
```
castles = 7
dirigibles = 4
# write here
print("I've built",castles, "castles in the air")
print("I have", dirigibles, "steam dirigibles")
print("I want a dirigible parked at each castle")
print("So I will buy other", castles - dirigibles, "at the Steam Market")
```

```
I've built 7 castles in the air
I have 4 steam dirigibles
I want a dirigible parked at each castle
So I will buy other 3 at the Steam Market
```

4.1.5 Visualizing the execution with Python Tutor

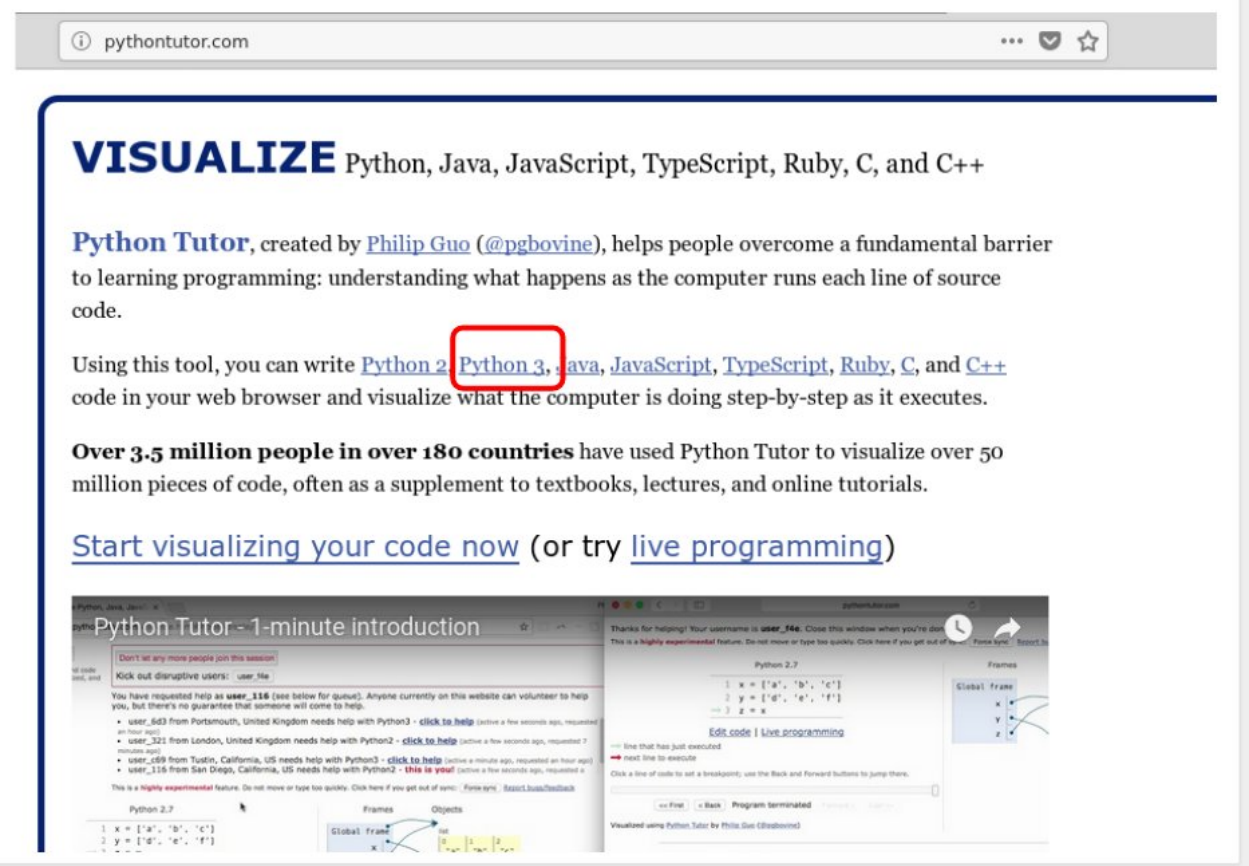
We have seen some of the main data types. Before going further, it's good to see the right tools to understand at best what happens when we execute the code.

Python tutor[68] is a very good website to visualize online Python code execution, allowing to step forth and *back* in code flow. Exploit it as much as you can, it should work with many of the examples we shall see in the book. Let's try an example

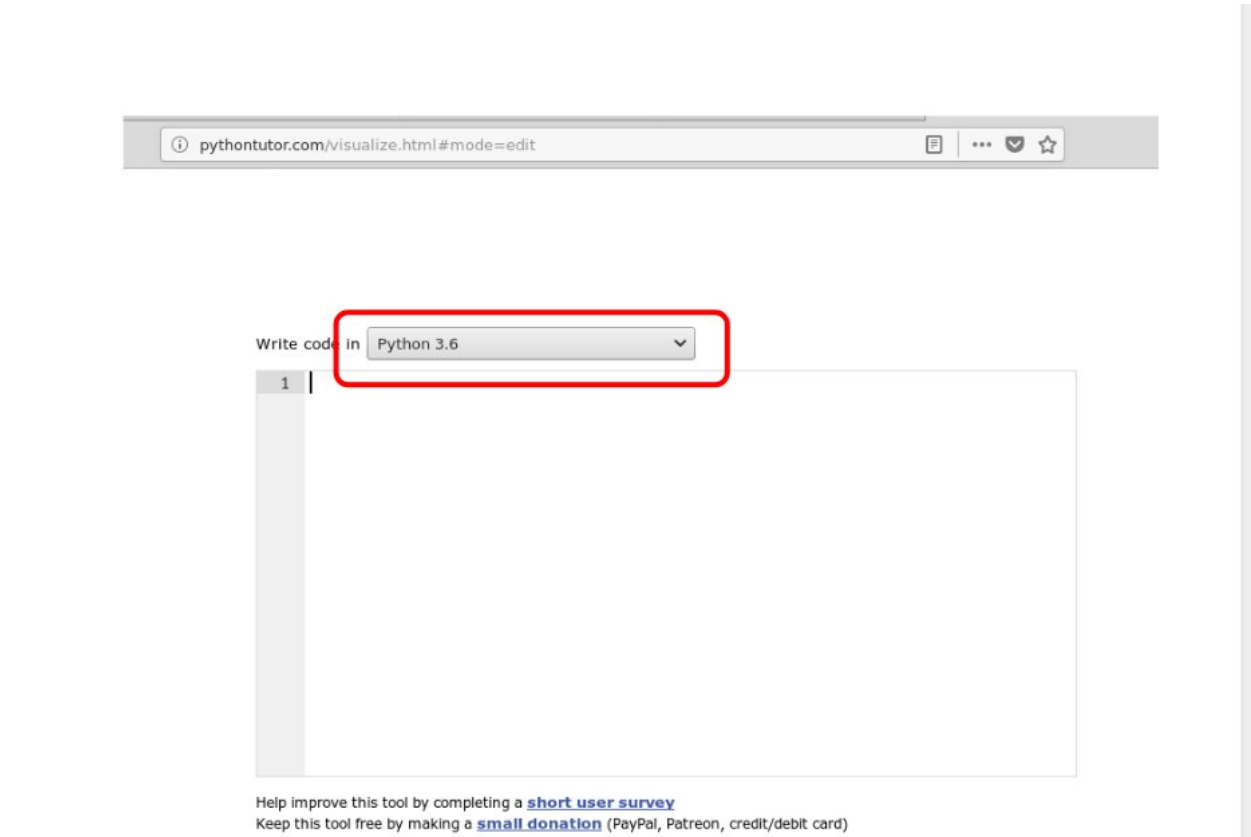
Python tutor 1/4

Go to pythontutor.com[69] and select *Python 3*

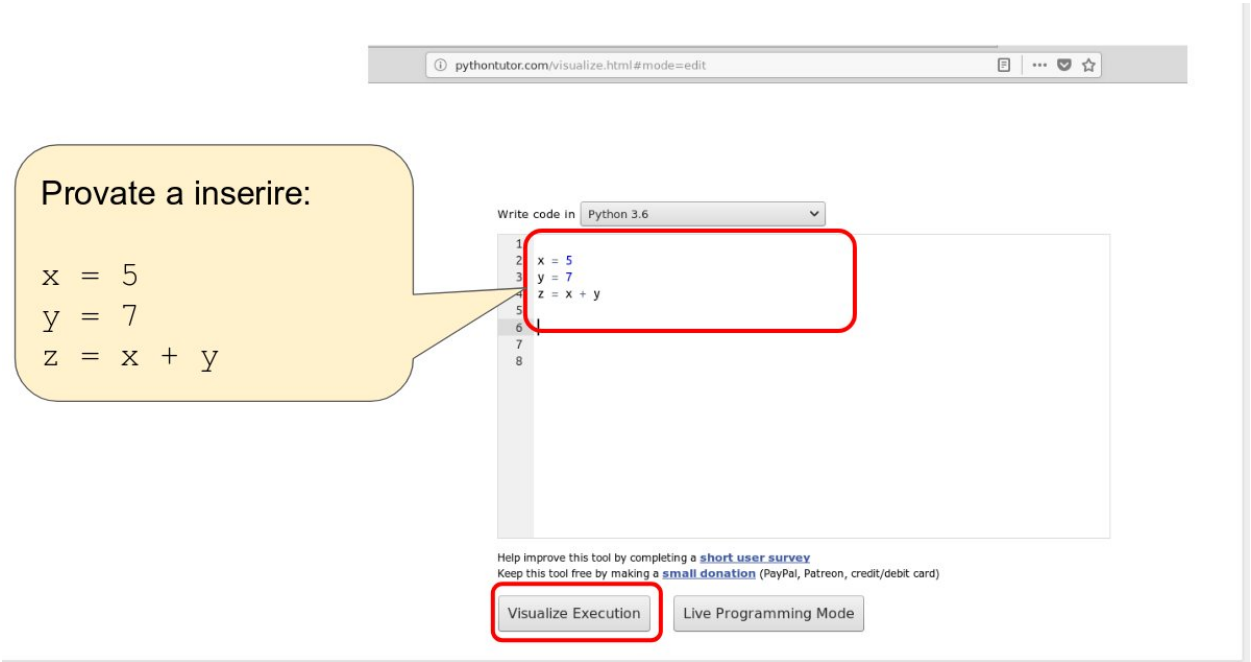
[68] http://pythontutor.com/
[69] http://pythontutor.com/



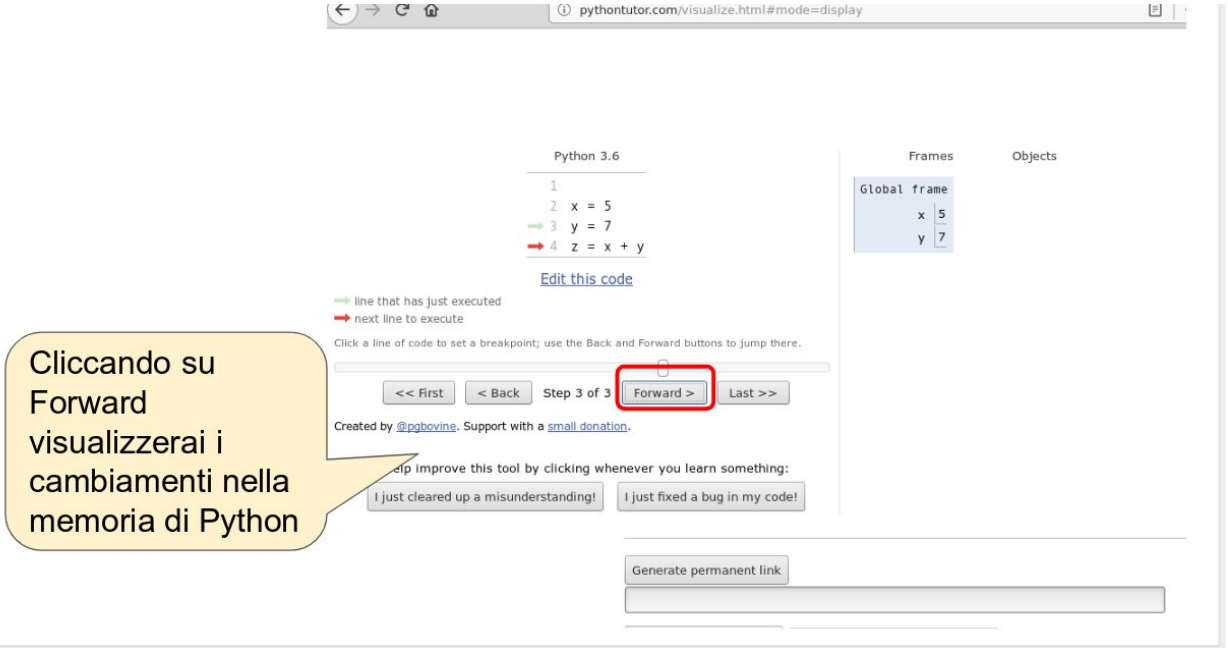
Python tutor 2/4



Python tutor 3/4



Python tutor 4/4



Debugging code in Jupyter

Python Tutor is fantastic, but when you execute code in Jupyter and it doesn't work, what can you do? To inspect the execution, the editor usually make available a tool called *debugger,* which allows to execute instructions one by one. At present (August 2018), the Jupyter debugger is called pdb[70] and it is extremely limited. To overcome limitations, in this book we invented a custom solution which exploits Python Tutor.

If you insert Python code in a cell, and then **at the cell end** you write the instruction `jupman.pytut()`, the preceding code will be visualized inside Jupyter notebook with Python Tutor, as if by magic.

> **WARNING**: `jupman` is a collection of support functions we invented just for this book.
>
> Whenever you see commands which start with `jupman`, to make them work you need first to execute the cell at the beginning of the document. For convenience we report here that cell. If you already didn't, execute it now.

[13]:
```
# Remember to execute this cell with Control+Enter
# These commands tell Python where to find the file jupman.py
import sys;
sys.path.append('../');
import jupman;
```

Now we are ready yo try Python Tutor with the magic function `jupman.pytut()`:

[14]:
```
x = 5
y = 7
z = x + y

jupman.pytut()
```

[14]:
```
<IPython.core.display.HTML object>
```

[70] https://davidhamann.de/2017/04/22/debugging-jupyter-notebooks/

Python Tutor : Limitation 1

Python Tutor is handy, but there are important limitations:

> **ATTENTION**: Python Tutor only looks inside one cell!
>
> Whenever you use Python Tutor inside Jupyter, the only code Python tutors considers is the one inside the cell where the command `jupman.pytut()` is.

So for example in the two following cells, only `print(w)` will appear inside Python tutor without the `w = 3`. If you try clicking *Forward* in Python tutor, you will we warned that `w` was not defined.

[15]:
```
w = 3
```

[16]:
```
print(w)

jupman.pytut()
```

```
3
```

```
Traceback (most recent call last):
  File "../jupman.py", line 2305, in _runscript
    self.run(script_str, user_globals, user_globals)
  File "/usr/lib/python3.5/bdb.py", line 431, in run
    exec(cmd, globals, locals)
  File "<string>", line 2, in <module>
NameError: name 'w' is not defined
```

[16]:
```
<IPython.core.display.HTML object>
```

To have it work in Python Tutor you must put ALL the code in the SAME cell:

[17]:
```
w = 3
print(w)

jupman.pytut()
```

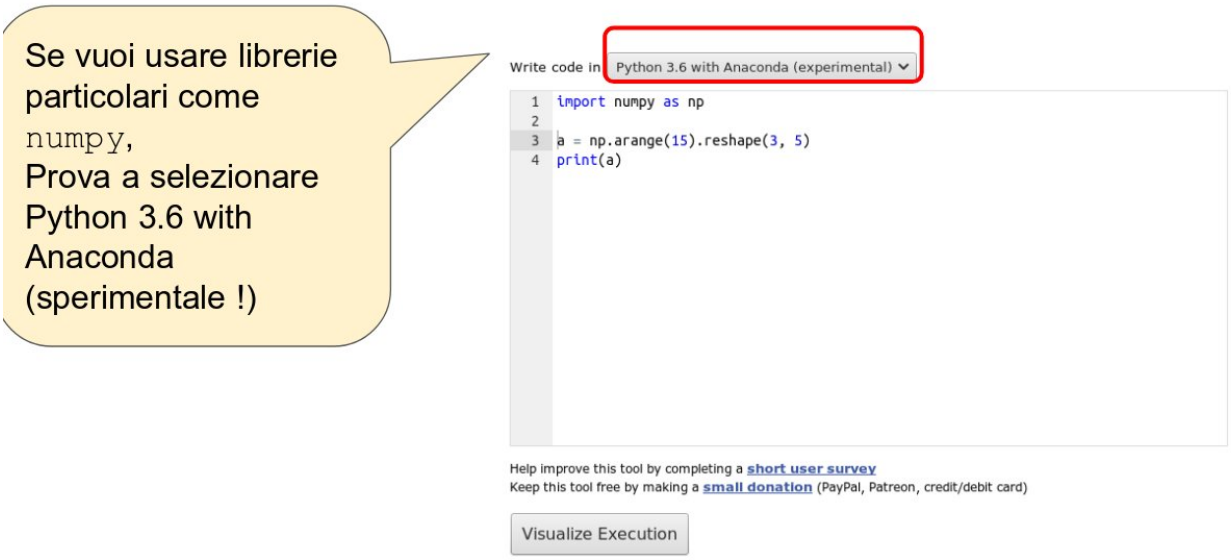
```
3
```

[17]:
```
<IPython.core.display.HTML object>
```

Python Tutor : Limitation 2

> **ATTENZIONE: Python Tutor only uses functions from standrd PYthon distribution**
>
> PYthon Tutor is good to inspect simple algorithms with basic Python functions, if you use libraries from third parties it will not work.

If you use some library like `numpy`, you can try **only online** to select `Python 3.6 with anaconda`



Exercise - tavern

Given the variables

```
pirates = 10
each_wants = 5       # mugs of grog
kegs = 4
keg_capacity = 20  # mugs of grog
```

Try writing some code which prints:

```
Nella taverna ci sono 10 pirati, ognuno vuole 5 boccali di grog
Abbiamo 4 barili di grog pieni
Da ogni barile possiamo prendere 20 boccali
Stasera i pirati berranno 50 boccali, ne avanzano 30 per domani
```

- **DO NOT** use numerical constant in your code, instead try using proposed variables
- To keep track of remaining kegs, make a variable `remaining_mugs`
- if you are using Jupyter, try using `jupman.pytut()` at the cell end to visualize the execution

[18]:
```
pirates = 10
each_wants = 5       # mugs of grog
kegs = 4
keg_capacity = 20  # mugs of grog


# write here
print("In the tavern there are", pirates, "pirates, each wants", each_wants, "mugs of␣
↪grog")
print("We have", kegs, "kegs full of grog")
print("From each keg we can take", keg_capacity,"mugs")
remaining_mugs = kegs*keg_capacity - pirates*each_wants
print("Tonight the pirates will drink", pirates * each_wants, "mugs, and", remaining_
↪mugs, "will remain for tomorrow")
```

(continues on next page)

(continued from previous page)

```
#jupman.pytut()
```

```
In the tavern there are 10 pirates, each wants 5 mugs of grog
We have 4 kegs full of grog
From each keg we can take 20 mugs
Tonight the pirates will drink 50 mugs, and 30 will remain for tomorrow
```

4.1.6 Python Architecture

The following part is not strictly fundamental to understand the book, it's useful to understand what happens under the hood when you execute commands.

Let's go back to Jupyter: the notebook editor Jupyter is a very powerful tool and flexible, allows to execute Python code, not only that, also code written in other programming languages (R, Bash, etc) and formatting languages (HTML, Markdown, Latex, etc).

Se must keep in mind that the Python code we insert in cells of Jupyter notebooks (the files with extension `.ipynb`) is not certainly magically understood by your computer. Under the hood, a lot of transformations are performed so to allow you computer processor to understaned the instructions to be executed. We report here the main transformations which happen, from Jupyter to the processor (CPU):

Python is a high level language

Let's try to understand well what happens when you execute a cell:

1. **source code**: First Jupyter checks if you wrote some Python *source code* in the cell (it could also be other programming languages like R, Bash, or formatting like Markdown …). By default Jupyter assumes your code is Python. Let's suppose there is the following code:

```
x = 3
y = 5
print(x + y)
```

EXERCISE: Without going into code details, try copy/pasting it into the cell below. Making sure to have the cursor in the cell, execute it with `Control + Enter`. When you execute it an `8` should appear as calculation result. The `# write down here` as all rows beginning with a sharp `#` is only a comment which will be ignored by Python

[19]:
```
# write down here
```

If you managed to execute the code, you can congratulate Python! It allowed you to execute a program written in a quite comprehensible language *independently* from your operating system (Windows, Mac Os X, Linux …) and from the processor of your computer (x86, ARM, …)! Not only that, the notebook editor Jupyter also placed the result in your browser.

In detail, what happened? Let's see:

2. **bytecode**: When requesting the execution, Jupyter took the text written in the cell, and sent it to the so-called *Python compiler* which transformed it into *bytecode*. The *bytecode* is a longer sequence of instructions which is less intelligeble for us humans (**this is only an example, there is no need to understand it !!**):

```
2           0 LOAD_CONST               1 (3)
            3 STORE_FAST               0 (x)

3           6 LOAD_CONST               2 (5)
            9 STORE_FAST               1 (y)

4          12 LOAD_GLOBAL              0 (print)
           15 LOAD_FAST                0 (x)
           18 LOAD_FAST                1 (y)
           21 BINARY_ADD
           22 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
           25 POP_TOP
           26 LOAD_CONST               0 (None)
           29 RETURN_VALUE
```

3. **machine code**: The *Python interpreter* took the *bytecode* above one instruction per time, and converted it into *machine code* which can actually be understood by the processor (CPU) of your computer. To us the *machine code* may look even longer and uglier of *bytecode* but the processor is happy and by reading it produces the program results. Example of *machine code* (**it is just an example, you do not need to understand it !!**):

```
mult:
    push rbp
    mov rbp, rsp
    mov eax, 0
mult_loop:
    cmp edi, 0
    je mult_end
    add eax, esi
    sub edi, 1
    jmp mult_loop
mult_end:
    pop rbp
    ret
```

We report in a table what we said above. In the table we explicitly write the file extension ni which we can write the various code formats

- The ones interesting for us are Jupyter notebooks `.ipynb` and Python source code files `.py`
- `.pyc` file smay be generated by the compiler when reading `.py` files, but they are not interesting to us, we will never need to edit the,
- `.asm` machine code also doesn't matter for us

Tool	Language	File	Example
Jupyter Notebook	Python	.ipynb	
Python Compiler	Python source code	.py	`x = 3 y = 5print(x + y )`
Python Interpreter	Python bytecode	.pyc	`0 LOAD_CONST 1 (3)3 STORE_FAST 0 (x)`
Processor (CPU)	Machine code	.asm	`cmp edi, 0je mult _end`

No that we now have an idea of what happens, we can maybe understand better the statement *Python is a high level language,* that is, it's positioned high in the above table: when we write Python code, we are not interested in the generated *bytecode* or *machine code,* we can **just focus on the program logic**. Besides, the Python code we write is **independent from the pc architecture**: if we have a Python interpreter installed on a computer, it will take care of converting the high-level code into the machine code of that particular architecture, which includes the operating system (Windows / Mac Os X / Linux) and processor (x86, ARM, PowerPC, etc).

Performance

Everything has a price. If we want to write programs focusing only on the *high level logic* without entering into the details of how it gets interpreted by the processor, we tyipcally need to give up on *performance.* Since Python is an *interpreted* language has the downside of being slow. What if we really need efficiency? Luckily, Python can be extended with code written in *C language* which typically is much more performant. Actually, even if you won't notice it, many functions of Python under the hood are directly written in the fast C language. If you really need performance (not in this book!) it might be worth writing first a prototype in Python and, once established it works, compile it into *C language* by using Cython compiler[71] and manually optimize the generated code.

```
[ ]:
```

4.2 Python basics

4.2.1 Download exercises zip

Browse online files[72]

PREREQUISITES:

- **Having installed Python 3 and Jupyter:** if you haven't already, look Installazion[73]
- **Having read** Tools and scripts[74]

4.2.2 Jupyter

Jupyter is an editor taht allows to work on so called *notebooks*, which are files ending with the extension `.ipynb`. They are documents divided in cells where for each cell you can insert commands and immediately see the respective output. Let's try to open this.

1. Unzip exercises zip in a folder, you should obtain something like this:

```
basics
    basics-sol.ipynb
    basics.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `basics.ipynb`

WARNING: open the one WITHOUT the `-sol` at the end!

Seeing now the solutions is way too easy ;-)

[71] http://cython.org/
[72] https://github.com/DavidLeoni/softpython-en/tree/master/basics
[73] https://en.softpython.org/installation.html
[74] https://en.softpython.org/tools/tools-sol.html

WARNING: If you don't find Jupyter / something doesn't work, read the installation guide[75]

3. Go on reading the exercises file, sometimes you will find inside exercises which ask you to write Python commands in the following cells. Exercises are graded by difficulty, from one star ⊛ to four ⊛⊛⊛⊛.

WARNING: In this book we use **ONLY PYTHON 3**

If you obtain weird results, check you are using Python 3 and not 2. If by chance your operating system when you write `python` runs version 2, you can try executing the third with the command: `python3`

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

4.2.3 Objects

In Python everything is an object. Objects have **properties** (fields where to save values) and **methods** (things they can do). For example, an object **car** has the *properties* model, brand, color, numer of doors, etc … and the *methods* turn right, turn left, accelerate, brake, shift gear …

According to Python official documentation:

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

For now it's enough to know that Python objects have an **identifier** (like, their name), a **type** (numbers, text, collections, …) and a **value** (the actual value represented by objects). Once the object has been created the *identifier* and the *type* never change, while the *value* may change (**mutable objects**) or remain constant (**immutable objects**).

Python provides these predefined types (*built-in*):

Type	Meaning	Domain	Mutable?
`bool`	Condition	`True`, `False`	no
`int`	Integer	$\mathbb{Z}$	no
`long`	Integer	$\mathbb{Z}$	no
`float`	Rational	$\mathbb{Q}$ (more or less)	no
`str`	Text	Text	no
`list`	Sequence	Collezione di oggetti	yes
`tuple`	Sequence	Collezione di oggetti	no
`set`	Set	Collezione di oggetti	yes
`dict`	Mapping	Mapping between objects	yes

For now we will consider only the simplest ones, later in the book we will deep dive in each of them.

[75] https://en.softpython.org/installation.html#Notebook-Jupyter

4.2.4 Variables

Variables are associations among names and objects (we can call them values).

Variables can be associated, or in a more technical term, *assigned* to objects by using the assignment operator =.

The instruction

[2]:
```
diamonds = 4
```

may represent how many precious stones we keed in the safe. What happens when we execute it in Python?

- an object is created
- its type is set to `int` (an integer number)
- its value is set to 4
- a name `diamonds` is create in the environment and assigned to that object

Detect the type of a variable

When you see a variable or costant and you wonder what type it could have, you can use the predefined function `type`:

[3]:
```
type(diamonds)
```

[3]:
```
int
```

[4]:
```
type(4)
```

[4]:
```
int
```

[5]:
```
type(4.0)
```

[5]:
```
float
```

[6]:
```
type("Hello")
```

[6]:
```
str
```

Reassign a variable

Consider now the following code:

[7]:
```
diamonds = 4
print(diamonds)
```

```
4
```

[8]:
```
diamonds = 5
print(diamonds)
```

```
5
```

The value of `diamonds` variable has been changed from 4 to 5, but as reported in the previous table, the `int` type is **immutable**. Luckily, this didn't prevent us from changing the value `diamonds` from 4 to 5. What happend behind the scenes? When we executed the instructions `diamonds = 5`, a new object of type `int` was created (the integer `5`) and made available with the same name `diamonds`

Riusing a variable

When you reassign a variable to another value, to calculate the new value you can freely reuse the old value of the variable you want to change. For example, suppose to have the variable

[9]:
```
flowers = 4
```

and you want to augment the number of `flowers` by one. You can write like this:

[10]:
```
flowers = flowers + 1
```

What happened? When Python encounters a command with =, FIRST it calculates the value of the expression it finds to the right of the =, and THEN assigns that value to the variable to the left of the =.

Given this order, FIRST in the expression on the right the old value is used (in this case `4`) and `1` is summed so to obtain `5` which is THEN assigned to `flowers`.

[11]:
```
flowers
```

[11]:
```
5
```

In a completely equivalent manner, we could rewrite the code like this, using a helper variable `x`. Let's try it in Python Tutor:

[12]:
```
# WARNING: to use the following jupman.pytut() function,
# it is necessary first execute  this cell with Shift+Enter

# it's enough to execute once, you can also find in all notebooks in the first cell.

import sys
sys.path.append('../')
import jupman
```

[13]:
```

flowers = 4

x = flowers + 1

flowers = x

jupman.pytut()
```

[13]:
```
<IPython.core.display.HTML object>
```

You can execute a sum and do an assignment at the same time with the += notation

[14]:
```
flowers = 4
flowers += 1
print(flowers)
```

```
5
```

This notation is also valid for other arithetic operators:

[15]:
```
flowers = 5
flowers -= 1     # subtraction
print(flowers)
```

```
4
```

[16]:
```
flowers *= 3      # multiplication
print(flowers)
```

```
12
```

[17]:
```
flowers /= 2      # division
print(flowers)
```

```
6.0
```

Assignments - questions

QUESTION: Look at the following questions, and for each try to guess the result it produces (or if it gives an error). Try to verify your guess both in Jupyter and in another editor of `.py` files like Spyder:

1.
```
x = 1
x
x
```

2.
```
x = 1
x = 2
print(x)
```

3.
```
x = 1
x = 2
x
```

4.
```
x = 1
print(x)
x = 2
print(x)
```

5.
```
print(zam)
print(zam)
zam = 1
zam = 2
```

6.
```
x = 5
print(x,x)
```

7.
```
x = 5
print(x)
print(x)
```

8.
```
x = 3
print(x,x*x,x**x)
```

9.
```
3 + 5 = x
print(x)
```

10.
```
3 + x = 1
print(x)
```

11.
```
x + 3 = 2
print(x)
```

12.
```
x = 2
x =+ 1
print(x)
```

13.
```
x = 2
x = +1
print(x)
```

14.
```
x = 2
x += 1
print(x)
```

15.
```
x = 3
x *= 2
print(x)
```

Exercise - exchange

✇ Given two variables `a` and `b`:

```
a = 5
b = 3
```

write some code that exchanges the two values, so that after your code it must result

```
>>> print(a)
3
>>> print(b)
5
```

- are two variables enough? If they aren't, try to introduce a third one.

[18]:
```
a = 5
b = 3

# write here
temp = a    # associate 5 to temp variable, so we have a copy
a = b       # reassign a to the value of b, that is 3
b = temp    # reassign b to the value of temp, that is 5
#print(a)
#print(b)
```

Da fare - ciclare

⊛ Write a program that given three variables with numebers `a`,`b`,`c`, cycles the values, that is, puts the value of `a` in `b`, the value of `b` in `c`, and the value of `c` in `a` .

So if you begin like this:

```
a = 4
b = 7
c = 9
```

After the code that you will write, by running this:

```
print(a)
print(b)
print(c)
```

You should see:

```
9
4
7
```

There are various ways to do it, try to use **only one** temporary variable and be careful not to lose values !

HINT: to help yourself, try to write down in comments the state of the memory, and think which command to do

`python # a b c t which command do I need? # 4 7 9 # 4 7 9 7 t = b # # #`

[19]:
```
a = 4
b = 7
c = 9

# write code here

print(a)
print(b)
print(c)
```

```
4
7
9
```

[20]:
```
# SOLUTION

a = 4
b = 7
c = 9


# a b c t  which command do I need?
# 4 7 9
# 4 7 9 7  t = b
# 4 4 9 7  b = a
# 9 4 9 7  a = c
# 9 4 7 7  c = t


t = b
```

(continues on next page)

(continued from previous page)

```
b = a
a = c
c = t

print(a)
print(b)
print(c)
```

```
9
4
7
```

Changing type during execution

You can also change the type of a variable duting the program execution but normally it is a **bad habit** because it makes harder to understand the code, and increases the probability to commit errors. Let's make an example:

[21]:
```
diamonds = 4          # integer
```

[22]:
```
diamonds + 2
```

[22]:
```
6
```

[23]:
```
diamonds = "four"  # text
```

Now that `diamonds` became text, if by mistake we try to treat it as if it were a number we will get an error !!

```
diamonds + 2

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-6124a47997d7> in <module>
----> 1 diamonds + 2

TypeError: can only concatenate str (not "int") to str
```

Multiple commands on the same line

It is possible to put many commands on the same line (non only assignments) by separating them with a semi-colon `;`

[24]:
```
a = 10; print('So many!'); b = a + 1;
```

```
So many!
```

[25]:
```
print(a,b)
```

```
10 11
```

NOTE: multiple commands on the same line are 'not much pythonic'

Even if sometimes they may be useful and less verbose of explicit definitions, they are a style frowned upon by true Python ninjas.

Multiple initializations

Another thing are multiple initializations, separated by a comma `,` like:

[26]:
```
x,y = 5,7
```

[27]:
```
print(x)
```
```
5
```

[28]:
```
print(y)
```
```
7
```

Differently from multiple commands, multiple assignments are a more acceptable style.

Exercise - exchange like a ninja

Try now to exchange the value of the two variables `a` and `b` in one row with multiple initialization

```
a,b = 5,3
```

After your code, it must result

```
>>> print(a)
3
>>> print(b)
5
```

[29]:
```
a,b = 5,3

# write here
a,b = b,a
#print(a)
#print(b)
```

Names of variables

IMPORTANT NOTE:

You can chose the name that you like for your variables (we advise to pick something reminding their meaning), but you need to adhere to some simple rules:

1. I nome possono solo contenere caratteri in maiuscolo/minuscolo (`A-Z`, `a-z`), numeri (`0-9`) o *underscore* _
2. I nomi non possono iniziare con un numero
3. i nomi di variabile dovrebbero iniziare con lettera minuscola
4. I nomi non possono essere uguali a parole riservate
5. Names can only contain upper/lower case digits (`A-Z`, `a-z`), numbers (`0-9`) or underscores _;
6. Names cannot start with a number;
7. Variable names should start with a lowercase letter

8. Names cannot be equal to reserved keywords:

Reserved words:

`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`		

system functions: beyond reserved words (which are impossible to redefine), Python also offers several predefined system function:

- `bool`, `int`,`float`,`tuple`,`str`,`list`,`set`,`dict`
- `max`, `min`, `sum`
- `next`, `iter`
- `id`, `dir`, `vars`,`help`

Sadly, Python allows careless people to redefine them, but we **do not**:

V COMMANDMENT[76]**: You shall never ever redefine system functions**

Never declare variables with such names !

Names of variables - questions

For each of the following names, try to guess if it is a valid *variable name* or not, then try to assign it in following cell

1. `my-variable`
2. `my_variable`
3. `theCount`
4. `the count`
5. `some@var`
6. `MacDonald`
7. `7channel`
8. `channel7`
9. `stand.by`
10. `channel45`
11. `maybe3maybe`
12. `"ciao"`
13. `'hello'`
14. `as` PLEASE: DO UNDERSTAND THE *VERY IMPORTANT DIFFERENCE* BETWEEN THIS AND FOLLOWING TWOs !!!

[76] https://en.softpython.org/commandments.html#V-COMMANDMENT

15. `asino`
16. `As`
17. `lista` PLEASE: DO UNDERSTAND THE *VERY IMPORTANT DIFFERENCE* BETWEEN THIS AND FOLLOWING TWOs !!!
18. `list` DO NOT EVEN TRY TO ASSIGN THIS ONE IN THE INTERPRETER (like `list = 5`), IF YOU DO YOU WILL BASICALLY BREAK PYTHON
19. `List`
20. `black&decker`
21. `black & decker`
22. `glab()`
23. `caffè` (notice the accented è !)
24. `):-]`
25. `€zone` (notice the euro sign)
26. `some:pasta`
27. `aren'tyouboredyet`
28. `<angular>`

```
[30]: # write here
```

4.2.5 Numerical types

We already mentioned that numbers are **immutable objects**. Python provides different numerical types:

integers (`int`), reals (`float`), booleans, fractions and complex numbers.

It is possible to make arithmetic operations with the following operators, in precedence order:

Operator	Description
`**`	power
`+ -`	Unary plus and minus
`* / // %`	Multiplication, division, integer division, module
`+ -`	Addition and subtraction

There are also several predefined functions:

Function	Description
`min(x,y, ...)`	the minimum among given numbers
`max(x,y, ...)`	the maximum among given numbers
`abs(x)`	the absolute value

Others are available in the math[77] module (remember that in order to use them you must first import the module `math` by typing `import math`):

[77] https://docs.python.org/3/library/math.html

Function	Description
`math.floor(x)`	round `x` to inferior integer
`math.ceil(x)`	round `x` to superior integer
`math.sqrt(x)`	the square root
`math.log(x)`	the natural logarithm of `n`
`math.log(x,b)`	the logarithm of `n` in base `b`

… plus many others we don't report here.

4.2.6 Integer numbers

The range of values that integer can have is only limited by available memory. To work with numbers, Python also provides these operators:

```
[31]: 7 + 4
[31]: 11
```

```
[32]: 7 - 4
[32]: 3
```

```
[33]: 7 // 4
[33]: 1
```

NOTE: the following division among integers produces a **float** result, which uses a **dot** as separator for the decimals (we will see more details later):

```
[34]: 7 / 4
[34]: 1.75
```

```
[35]: type(7 / 4)
[35]: float
```

```
[36]: 7 * 4
[36]: 28
```

NOTE: in many programming languages the power operation is denoted with the cap `^`, but in Python it is denoted with double asterisk `**`:

```
[37]: 7 ** 4   # power
[37]: 2401
```

Exercise - deadline 1

⊛ You are given a very important deadline in:

[38]:
```
days = 4
hours = 13
minutes = 52
```

Write some code that prints the total minutes. By executing it, it should result:

```
In total there are 6592 minutes left.
```

[39]:
```
days = 4
hours = 13
minutes = 52

# write here
print("In total there are", days*24*60 + hours*60 + minutes, "minutes left")
```

```
In total there are 6592 minutes left
```

Modulo operator

To find the reminder of a division among integers, we can use the modulo operator which is denoted with `%`:

[40]:
```
5 % 3  # 5 divided by 3 gives 2 as reminder
```

[40]:
```
2
```

[41]:
```
5 % 4
```

[41]:
```
1
```

[42]:
```
5 % 5
```

[42]:
```
0
```

[43]:
```
5 % 6
```

[43]:
```
5
```

[44]:
```
5 % 7
```

[44]:
```
5
```

Exercise - deadline 2

⊛ For another super important deadline there are left:

```
tot_minutes = 5000
```

Write some code that prints:

```
There are left:
   3 days
   11 hours
   20 minutes
```

[45]:
```
tot_minutes = 5000

# write here
print('There are left:')
print('  ', tot_minutes // (60*24), 'days')
print('  ', (tot_minutes % (60*24)) // 60, 'hours')
print('  ', (tot_minutes % (60*24)) % 60, 'minutes')
```

```
There are left:
   3 days
   11 hours
   20 minutes
```

min and max

The minimum among two numbers can be calculated with the function `min`:

[46]:
```
min(7,3)
```

[46]:
```
3
```

and the maximum with the function `max`:

[47]:
```
max(2,6)
```

[47]:
```
6
```

To `min` and `max` we can pass an arbitrary number of parameters, even negatives:

[48]:
```
min(2,9,-3,5)
```

[48]:
```
-3
```

[49]:
```
max(2,9,-3,5)
```

[49]:
```
9
```

V COMMANDMENT[78]: **You shall never ever redefine system functions like `min` and `max`**

If you use `min` and `max` like they were variables, the corresponding functions will *literally* stop to work!

```
min = 4    # NOOOO !
max = 7    # DON'T DO IT !
```

QUESTION: given two numbers `a` and `b`, which of the following expressions are equivalent?

```
1. max(a,b)
2. max(min(a,b),b)
```

(continues on next page)

[78] https://en.softpython.org/commandments.html#V-COMMANDMENT

(continued from previous page)

```
3. -min(-a,-b)
4. -max(-a,-b)
```

ANSWER: 1. and 3. are equivalent

Exercise - transportation

⊛ A company has a truck that every day delivers products to its best client. The truck can at most transport 10 tons of material. Unfortunately, the roads it can drive through have bridges that limit the maximum weight a vehicle can have to pass. These limits are provided in 5 variables:

```
b1,b2,b3,b4,b5 = 7,2,4,3,6
```

The truck must always go through the bridge `b1`, then along the journey there are three possible itineraries available:

- In the first itinerary, the truck also drives through bridge `b2`
- In the second itinerary, the truck also drives through bridges `b3` and `b4`
- In the third itinerary, the truck also drives though bridge `b5

The company wants to know which are the maximum tons it can drive to destination in a sngle journey. Write some code to print this number.

NOTE: we do not want to know which is the best itinerary, we only need to find the greatest number of tons to ship.

Example - given:

```
b1,b2,b3,b4,b5 = 7,2,4,6,3
```

your code must print:

```
In a single journey we can transport at most 4 tons.
```

[50]:
```
b1,b2,b3,b4,b5 = 7,2,4,6,3   # 4
#b1,b2,b3,b4,b5 = 2,6,2,4,5  # 2
#b1,b2,b3,b4,b5 = 8,6,2,9,5  # 6
#b1,b2,b3,b4,b5 = 8,9,9,4,7  # 8


# write here

print('In a single journey we can transport at most',
       max(min(b1,b2), min(b1,b3,b4),min(b1,b5)),
      'tons')
```

```
In a single journey we can transport at most 4 tons
```

Exercise - armchairs

⊛ The tycoon De Industrionis owns to factories of armchairs, one in Belluno city and one in Rovigo. To make an armchair three main components are needed: a mattress, a seatback and a cover. Each factory produces all required components, taking a certain time to produce each component:

[51]:
```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24
```

Belluno takes 23h to produce a mattress, 54h the seatcback and 12h the cover. Rovigo, respectively, takes 13, 37 and 24 hours. When the 3 components are ready, assembling them in the finished armchair requires one hour.

Sometimes peculiar requests are made by filthy rich nobles, that pretends to be shipped in a few hours armchairs with extravagant like seatback in solid platinum and other nonsense.

If the two factories start producting the components at the same time, De Industrionis wants to know in how much time the first armchair will be produced. Write some code to calculate that number.

- **NOTE 1**: we are not interested in which factory will produce the armchair, we just want to know the shortest time in which we will get an armchair
- **NOTE 2**: suppose both factories **don't** have components in store
- **NOTE 3**: the two factories **do not** exchange components

Example 1 - given:

```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24
```

your code must print:

```
The first armchair will be produced in 38 hours.
```

Example 2 - given:

```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 81,37,32,54,36,91
```

your code must print:

```
The first armchair will be produced in 82 hours.
```

[52]:
```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24   # 38
#b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 81,37,32,54,36,91  # 82
#b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 21,39,47,54,36,91  # 48

# write here

t = min(max(b_mat, b_bac, b_cov) + 1, max(r_mat, r_bac, r_cov) + 1)

print('The first armchair will be produced in', t,'hours.')
```

```
The first armchair will be produced in 38 hours.
```

4.2.7 Booleans

Values of truth in Python are represented with the keywords `True` and `False`. A boolean object can only have the values `True` or `False`. These objects are used in boolean algebra and have the type `bool`.

[53]:
```
x = True
```

[54]:
```
x
```

[54]:
```
True
```

[55]:
```
type(x)
```

[55]:
```
bool
```

[56]:
```
y = False
```

[57]:
```
type(y)
```

[57]:
```
bool
```

Boolean operators

We can operate on boolean values with the operators `not`, `and`, `or`:

[58]:
```
# Expression     Result
not True         # False
not False        # True

False and False  # False
False and True   # False
True  and False  # False
True  and True   # True

False or False   # False
False or True    # True
True  or False   # True
True  or True    # True
```

[58]:
```
True
```

Booleans - Questions with costants

QUESTION: For each of the following boolean expressions, try to guess the result (*before* guess, and *then* try them !):

1. `not (True and False)`
2. `(not True) or (not (True or False))`
3. `not (not True)`
4. `not (True and (False or True))`

5. `not (not (not False))`
6. `True and (not (not((not False) and True)))`
7. `False or (False or ((True and True) and (True and False)))`

Booleans - Questions with variables

QUESTION: For each of these expressions, for which values of `x` and `y` they give `True`? Try to think an answer before trying!

NOTE: there can be many combinations that produce `True`, find them all

1. `x or (not x)`
2. `(not x) and (not y)`
3. `x and (y or y)`
4. `x and (not y)`
5. `(not x) or  y`
6. `y or not (y and x)`
7. `x and ((not x) or not(y))`
8. `(not (not x)) and not (x and y)`
9. `x and (x or (not(x) or not(not(x or not (x)))))`

QUESTION: For each of these expressions, for which values of `x` and `y` they give `False`?

NOTE: there can be many combinations that produce `False`, find them all

1. `x or ((not y) or z)`
2. `x or (not y) or (not z)`
3. `not (x and y and (not z))`
4. `not (x and (not y) and (x or z))`
5. `y or ((x or y) and (not z))`

Booleans - De Morgan

There are a couple of laws that sometimes are useful:

Formula	Equivalent to
`x or y`	`not(not x and not y)`
`x and y`	`not(not x or not y)`

QUESTION: Look at following expressions, and try to rewrite them in equivalent ones by using De Morgan laws, simplifying the result wherever possible. Then verify the translation produces the same result as the original for all possible values of `x` and `y`.

1. `(not x) or y`
2. `(not x) and (not y)`
3. `(not x) and (not (x or y))`

Example:

```
x,y = False, False
#x,y = False, True
#x,y = True, False
#x,y = True, True

orig = x or y
trans = not((not x) and (not y))
print('orig=',orig)
print('trans=',trans)
```

```
[59]: # write here
```

Booleans - Conversion

We can convert booleans into intergers with the predefined function `int`. Each integer can be converted into a boolean (and vice versa) with `bool`:

```
[60]: bool(1)
[60]: True
```

```
[61]: bool(0)
[61]: False
```

```
[62]: bool(72)
[62]: True
```

```
[63]: bool(-5)
```

[63]:
```
True
```

[64]:
```
int(True)
```

[64]:
```
1
```

[65]:
```
int(False)
```

[65]:
```
0
```

Each integer is valued to `True` except `0`. Note that truth values `True` and `False` behave respectively like integers `1` and `0`.

Booleans - Questions - what is a boolean?

QUESTION: For each of these expressions, which results it produces?

1. `bool(True)`
2. `bool(False)`
3. `bool(2 + 4)`
4. `bool(4-3-1)`
5. `int(4-3-1)`
6. `True + True`
7. `True + False`
8. `True - True`
9. `True * True`

Booleans - Evaluation order

For efficiency reasons, during the evaluation of a boolean expression if Python discovers the possible result can only be one, it then avoids to calculate further expressions. For example, in this expression:

```
False and x
```

by reading from left to right, in the moment we encounter `False` we already know that the result of `and` operation will always be `False` independetly from the value of `x` (convince yourself).

Instead, if while reading from left to right Python finds first `True`, it will continue the evaluation of following expressions and as result of the whole `and` will return the evaluation of the **last** expression. If we are using booleans, we will not notice the differences, but by exchanging types we might get surprises:

[66]:
```
True and 5
```

```
[66]: 5
```

```
[67]: 5 and True
[67]: True
```

```
[68]: False and 5
[68]: False
```

```
[69]: 5 and False
[69]: False
```

Let's think which order of evaluation Python might use for the `or` operator. Have a look at the expression:

```
True or x
```

By reading from left to right, as soon as we find the `True` we mich conclude that the result of the whole `or` must be `True` independently from the value of `x` (convince yourself).

Instead, if the first value is `False`, Python will continue in the evaluation until it finds a logical value `True`, when this happens that value will be the result of the whole expression. We can notice it if we use different costants from `True` and `False`:

```
[70]: False or 5
[70]: 5
```

```
[71]: 7 or False
[71]: 7
```

```
[72]: 3 or True
[72]: 3
```

The numbers you see have always a logical result coherent with the operations we did, that is, if you see `0` the expression result is intended to have logical value `False` and if you see a number different from `0` the result is intended to be `True` (convince yourself).

QUESTION: Have a look at the following expressions, and for each of them try to guess which result it produces (or if it gives an error):

1. `0 and True`
2. `1 and 0`
3. `True and -1`
4. `0 and False`
5. `0 or False`
6. `0 or 1`

7.
```
False or -6
```

8.
```
0 or True
```

Booleans - evaluation errors

What happens if a boolean expression contains some code that would generate an error? According to intuition, the program should terminate, but it's not always like this.

Let's try to generate an error on purpose. During math lessons they surely told you many times that dividing a number by zero is an error because the result is not defined. So if we try to ask Python what the result of `1/0` is we will (predictably) get complaints:

```
print(1/0)
print('after')
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-51-9e1622b385b6> in <module>()
----> 1 1/0

ZeroDivisionError: division by zero
```

Notice that `'after'` *is not* printed because the progam gets first interrupted.

What if we try to write like this?

```
[73]: False and 1/0
```

```
[73]: False
```

Python produces a result without complaining ! Why? Evaluating form left to right it found a `False` and so it concluded before hand that the expression result must be `False`. Many times you will not be aware of these potential problems but it is good to understand them because there are indeed situations in which you can event exploit the execution order to prevent errors (for example in `if` and `while` instructions we will see later in the book).

QUESTION: Look at the following expression, and for each of them try to guess which result it produces (or if it gives on error):

1.
```
True and 1/0
```

2.
```
1/0 and 1/0
```

3.
```
False or 1/0
```

4.
```
True or 1/0
```

5.
```
1/0 or True
```

6.
```
1/0 or 1/0
```

7.
```
True or (1/0 and True)
```

8. `(not False) or not 1/0`

9. `True and 1/0 and True`

10. `(not True) or 1/0 or True`

11. `True and (not True) and 1/0`

Comparison operators

Comparison operators allow to build *expressions* which return a boolean value:

Comparator	Description
`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

```
[74]: 3 == 3
[74]: True
```

```
[75]: 3 == 5
[75]: False
```

```
[76]: a,b = 3,5
```

```
[77]: a == a
[77]: True
```

```
[78]: a == b
[78]: False
```

```
[79]: a == b - 2
[79]: True
```

```
[80]: 3 != 5  # 3 is different from 5 ?
[80]: True
```

```
[81]: 3 != 3  # 3 is different from 3 ?
[81]: False
```

```
[82]: 3 < 5
```

```
[82]: True
```

```
[83]: 5 < 5
[83]: False
```

```
[84]: 5 <= 5
[84]: True
```

```
[85]: 8 > 5
[85]: True
```

```
[86]: 8 > 8
[86]: False
```

```
[87]: 8 >= 8
[87]: True
```

Since the comparison are expressions which produce booleans, we can also assign the result to a variable:

```
[88]: x = 5 > 3
```

```
[89]: print(x)
      True
```

QUESTION: Look at the following expression, and for each of them try to guess which result it produces (or if it gives on error):

1.
```
x = 3 == 4
print(x)
```

2.
```
x = False or True
print(x)
```

3.
```
True or False = x or False
print(x)
```

4.
```
x,y = 9,10
z = x < y and x == 3**2
print(z)
```

5.
```
a,b = 7,6
a = b
x = a >= b + 1
print(x)
```

6.
```
x = 3^2
y = 9
print(x == y)
```

Booleans - References

- Think Python, Chapter 5, Conditional instructions and recursion[79] , in particular Sctions 5.2 and 5.3, Boolean expressions[80] You can skip recursion

4.2.8 Real numbers

Python saves the real numbers (floating point numbers) in 64 bit of information divided by sign, expnonent and mantissa (also called significand). Let's see an example:

```
[90]: 3.14
[90]: 3.14
```

```
[91]: type(3.14)
[91]: float
```

WARNING: you must use the dot instead of comma!

So you will write 3.14 instead of 3,14

Be very careful whenever you copy numbers from documents in latin languages, they might contain very insidious commas!

Scientifical notation

Whenever numbers are very big or very small, to avoid having to write too many zeros it is convenient to use scientifical notation with the *e* like xen which multiplies the number *x* by 10^n

With this notation, in memory are only put the most significative digits (the *mantissa*) and the exponent, thus avoiding to waste space.

```
[92]: 75e1
[92]: 750.0
```

```
[93]: 75e2
[93]: 7500.0
```

```
[94]: 75e3
[94]: 75000.0
```

```
[95]: 75e123
[95]: 7.5e+124
```

```
[96]: 75e0
[96]: 75.0
```

[79] http://greenteapress.com/thinkpython2/html/thinkpython2006.html
[80] http://greenteapress.com/thinkpython2/html/thinkpython2006.html#sec59

```
[97]: 75e-1
[97]: 7.5

[98]: 75e-2
[98]: 0.75

[99]: 75e-123
[99]: 7.5e-122
```

QUESTION: Look at the following expressions, and try to find which result they produce (or if they give and error):

1. `print(1.000.000)`
2. `print(3,000,000.000)`
3. `print(2000000.000)`
4. `print(2000000.0)`
5. `print(0.000.123)`
6. `print(0.123)`
7. `print(0.-123)`
8. `print(3e0)`
9. `print(3.0e0)`
10. `print(7e-1)`
11. `print(3.0e2)`
12. `print(3.0e-2)`
13. `print(3.0-e2)`
14. `print(4e2-4e1)`

Too big or too small numbers

Sometimes calculations on very big or extra small numbers may give as a result `math.nan` (Not a Number) or `math.inf`. For the moment we just mention them, you can find a detailed description in the Numpy page[81]

[81] https://en.softpython.org/matrices-numpy/matrices-numpy-sol.html#NaN-e-infinities

Exercise - circle

Calculate the area of a circle at the center of a soccer ball (radius = 9.1m), remember that $area = pi * r^2$

Your code should print as result `263.02199094102605`

[100]:
```
# SOLUTION

r = 9.15
pi = 3.1415926536
area = pi*(r**2)
print(area)
```

```
263.02199094102605
```

Note that the parenthesis around the squared `r` are not necessary because the power operator has the precedence, but htey may help in augmenting the code readability.

We recall here the operator precedence:

Operatore	Descrizione
`**`	Power (maximum precedence)
`+ -`	unary plus and minus
`* / // %`	Multiplication, division, integer division, modulo
`+ -`	Addition and subtraction
`<= < > >=`	comparison operators
`== !=`	equality operators
`not or and`	Logical operators (minimum precedence)

Exercise - fractioning

Write some code to calculate the value of the following formula for `x = 0.000003`, you should obtain `2.753278226511882`

$$-\frac{\sqrt{x+3}}{\frac{(x+2)^3}{\log x}}$$

[101]:
```
x = 0.000003

# write here
import math
- math.sqrt(x+3) / (((x+2)**3)/math.log(x))
```

```
[101]: 2.753278226511882
```

Exercise - summation

Write some code to calculate the value of the following expression (don't use cycles, write down all calculations), you should obtain `20.53333333333333`

$$\sum_{j=1}^{3} \frac{j^4}{j+2}$$

```
[102]: # write here
       ((1**4) / (1+2)) + ((2**4) / (2+2)) + ((3**4) / (3+2))
```

```
[102]: 20.53333333333333
```

Reals - conversion

If we want to convert a real to an integer, several ways are available:

Function	Description	Mathematical symbol	Result
`math.floor(x)`	round x to inferior integer	$\lfloor 8.7 \rfloor$	8
`int(x)`	round x to inferior integer	$\lfloor 8.7 \rfloor$	8
`math.ceil(x)`	round x to superior integer	$\lceil 5.3 \rceil$	6
`round(x)`	round x to closest integer	$\lfloor 2.5 \rceil$	2
		$\lfloor 2.51 \rceil$	3

QUESTION: Look at the following expressions, and for each of them try to guess which result it produces (or if it gives an error).

1. `math.floor(2.3)`
2. `math.floor(-2.3)`
3. `round(3.49)`
4. `round(3.5)`
5. `round(3.51)`
6. `round(-3.49)`
7. `round(-3.5)`
8. `round(-3.51)`
9. `math.ceil(8.1)`
10. `math.ceil(-8.1)`

QUESTION: Given a float `x`, the following formula is:

```
math.floor(math.ceil(x)) == math.ceil(math.floor(x))
```

1. always `True`
2. always `False`
3. sometimes `True` and sometimes `False` (give examples)

ANSWER: 3: for integers like `x=2.0` it is `True`, in other cases like `x=2.3` it is `False`

QUESTION: Given a float `x`, the following formula is:

```
math.floor(x) == -math.ceil(-x)
```

1. always `True`
2. always `False`
3. sometimes `True` and sometimes `False` (give examples)

ANSWER: 1.

Exercise - Invigorate

Excessive studies lead you search on internet recipes of energetic drinks. Luckily, a guru of nutrition just posted on her Instagram channel @DrinkSoYouGetHealthy this recipe of a miracle drink:

> Pour in a mixer 2 decilitres of kiwi juice, 4 decilitres of soy sauce, and 3 decilitres of shampoo of karitè bio. Mix vigorously and then pour half drink into a glass. Fill the glass until the superior deciliter. Swallow in one shot.

You run to shop the ingredients, and get ready for mixing them. You have a measuring cup with which you transfer the precious fluids, one by one. While transfering, you always pour a little bit more than necessary (but never more than 1 decilitre), and for each ingredient you then remove the excess.

- **DO NOT** use subtractions, try using only rounding operators

Example - given:

```
kiwi = 2.4
soia = 4.8
shampoo = 3.1
measuring_cup = 0.0
mixer = 0
glass = 0.0
```

Your code must print:

```
I pour into the measuring cup 2.4 dl of kiwi juice, then I remove excess until␣
↪keeping 2 dl
I transfer into the mixer, now it contains 2.0 dl
I pour into the measuring cup 4.8 dl of soia, then I remove excess until keeping 4 dl
I transfer into the mixer, now it contains 6.0 dl
I pour into the measuring cup 3.1 dl of shampoo, then I remove excess until keeping 3␣
↪dl
I transfer into the mixer, now it contains 9.0 dl
I pour half of the mix ( 4.5 dl ) into the glass
I fill the glass until superior deciliter, now it contains: 5 dl
```

[103]:
```
kiwi = 2.4
soy = 4.8
shampoo = 3.1
measuring_cup = 0.0
mixer = 0.0
glass = 0.0


# write here
print('I pour into the measuring cup', kiwi, 'dl of kiwi juice, then I remove excess␣
↪until keeping', int(kiwi), 'dl')
mixer += int(kiwi)
print('I transfer into the mixer, now it contains', mixer, 'dl')
print('I pour into the measuring cup', soy, 'dl of soia, then I remove excess until␣
↪keeping', int(soy), 'dl')
mixer += int(soy)
print('I transfer into the mixer, now it contains', mixer, 'dl')
print('I pour into the measuring cup', shampoo, 'dl of shampoo, then I remove excess␣
↪until keeping', int(shampoo), 'dl')
mixer += int(shampoo)
print('I transfer into the mixer, now it contains', mixer, 'dl')
bicchiere = mixer/2
print('I pour half of the mix (', glass, 'dl ) into the glass')
print('I fill the glass until superior deciliter, now it contains:', math.ceil(glass),
↪ 'dl')
```

```
I pour into the measuring cup 2.4 dl of kiwi juice, then I remove excess until␣
↪keeping 2 dl
I transfer into the mixer, now it contains 2.0 dl
I pour into the measuring cup 4.8 dl of soia, then I remove excess until keeping 4 dl
I transfer into the mixer, now it contains 6.0 dl
I pour into the measuring cup 3.1 dl of shampoo, then I remove excess until keeping 3␣
↪dl
```

(continues on next page)

(continued from previous page)

```
I transfer into the mixer, now it contains 9.0 dl
I pour half of the mix ( 0.0 dl ) into the glass
I fill the glass until superior deciliter, now it contains: 0 dl
```

Exercise - roundminder

Write some code to calculate the value of the following formula for `x = -5.50`, you should obtain `41`

$$||\lceil x \rceil|| + \lfloor x \rceil^2$$

[104]:
```
x = -5.50   # 41
#x = -5.49  # 30

# write here
abs(math.ceil(x)) + round(x)**2
```

[104]:
```
41
```

Reals - equality

WARNING: what follows is valid for *all* programming languages!

Some results will look weird but this is the way most processors (CPU) operates, independently from Python.

When floating point calculations are performed, the processor may introduce rounding errors due to limits of internal representation. Under the hood the numbers like floats are memorized in a sequence of binary code of 64 bits, according to *IEEE-754 floating point arithmetic* standard: this imposes a physical limit to the precision of numbers, and sometimes we ight get surprises due to conversion from decimal to binary. For example, let's try printing `4.1`:

[105]:
```
print(4.1)
```

```
4.1
```

For our convenience Python is showing us `4.1`, but in reality in the processor memory ended up a different number! Which one? To discover what it hides, with `format` function we can explicitly format the number to, for example 55 digits of precision by using the `f` format specifier:

[106]:
```
format(4.1, '.55f')
```

[106]:
```
'4.0999999999999996447286321199499070644378662109375000000'
```

We can then wonder what the result of this calculus might be:

[107]:
```
print(7.9 - 3.8)
```

```
4.1000000000000005
```

We note the result is still different from the expected one! By investigating further, we notice Python is not even showing all the digits:

```
[108]: format(7.9 - 3.8, '.55f')
[108]: '4.1000000000000005329070518200751394033432006835937500000'

[ ]:
```

What if wanted to know if the two calculations with float produce the 'same' result?

WARNING: AVOID THE == WITH FLOATS!

To understand if the result between the two calculations with the flots is the same, **YOU CANNOT** use the == operator !

```
[109]: 7.9 - 3.8 == 4.1     # TROUBLE AHEAD!
[109]: False
```

Instead, you should prefer alternative that evaluate if a float number is *close* to anoter, like for example the handy function math.isclose[82]:

```
[110]: import math

       math.isclose(7.9 - 3.8, 4.1)    # MUCH BETTER
[110]: True
```

By default `math.isclose` uses a precision of `1e-09`, but, if needed, you can also pass a tolerance limit in which the difference of the numbers must be so to be considered equal:

```
[111]: math.isclose(7.9 - 3.8, 4.1, abs_tol=0.000001)
[111]: True
```

QUESTION: Can we perfectly represnt the number $\sqrt{2}$ as a `float`?

ANSWER: $\sqrt{2}$ is irrational so there's no hope of a perfect representation, any calculation will always have a certain degree of imprecision.

QUESTION: Which of these expressions give the same result?

```
import math
print('a)', math.sqrt(3)**2 == 3.0)
print('b)', abs(math.sqrt(3)**2 - 3.0) < 0.0000001)
print('c)', math.isclose(math.sqrt(3)**2, 3.0, abs_tol=0.0000001))
```

ANSWER: b) and c) give `True`. a) gives `False`, because durting floating point calculations rounding errors are made.

[82] https://docs.python.org/3/library/math.html#math.isclose

Exercise - quadratic

Write some code to calculate the zeroes of the equation $ax^2 - b = 0$

- Show numbers with **20 digits** of precision
- At the end check that by substituting the value obtained x into the equation you actually obtain zero.

Example - given:

```
a = 11.0
b = 3.3
```

after your code it must print:

```
11.0 * x**2 - 3.3 = 0  per x1 = 0.54772255750516607442
11.0 * x**2 - 3.3 = 0  per x2 = -0.54772255750516607442
0.5477225575051661 is a solution? True
-0.5477225575051661 is a solution? True
```

[112]:
```
a = 11.0
b = 3.3

# write here

import math

x1 = math.sqrt(b/a)
x2 = -math.sqrt(b/a)

print(a, "* x**2 -",b,"= 0  per x1 =", format(x1, '.20f'))
print(a, "* x**2 -",b,"= 0  per x2 =", format(x2, '.20f'))

# we need to change the default tolerance value
print(format(x1, '.20f'), "is a solution?", math.isclose(a*(x1**2) - b,0, abs_tol=0.
↪00001))
print(format(x2, '.20f'), "is a solution?", math.isclose(a*((x2)**2) - b,0, abs_tol=0.
↪00001))
```

```
11.0 * x**2 - 3.3 = 0  per x1 = 0.54772255750516607442
11.0 * x**2 - 3.3 = 0  per x2 = -0.54772255750516607442
0.54772255750516607442 is a solution? True
-0.54772255750516607442 is a solution? True
```

Exercise - trendy

You are already thinking about next vacations, but there is a big problem: where do you go, if you don't have a selfie-stick. You cannot leave with this serious anxiety: to uniform yourself to this mass phenomena you must buy the stick most similar to others. You then conduct a rigourous statistical survey among turists obssessed by selfie sticks with the goal to find the most frequent brands of selfie sticks, in other words, the *mode* of the frequencies. You obtain these results:

[113]:
```
b1,b2,b3,b4,b5 = 'TooManyLikes', 'Boombasticks', 'Timewasters Inc', 'Vanity 3.0',
↪'TrashTrend' # brand
f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.3   # frequencies (as percentages)
```

We deduce that masses love selfie-sticks of the brand `'Boombasticks'` and `TrashTrend`, both in a tie with 30% turists each. Write some code which prints this result:

```
TooManyLike is the most frequent? False ( 25.0 % )
ErMejo United è il più frequente? True ( 30.0 % )
Perditempo Inc è il più frequente? False ( 10.0 % )
Vanità 3.0 è il più frequente? False ( 5.0 % )
TronistiPerCaso è il più frequente? True ( 30.0 % )
```

- **WARNING**: your code must work with **ANY** series of variables !!

[114]:
```
b1,b2,b3,b4,b5 = 'TooManyLikes', 'Boombasticks', 'Timewasters Inc', 'Vanity 3.0',
↪'TrashTrend'    # brand

f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.3  # frequencies (as percentages)  False␣
↪True False False True
# CAREFUL, they look the same but it must work also with these!
#f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.1 + 0.2  #  False True False False True

# write here
mx = max(f1,f2,f3,f4,f5)
print(b1, 'is the most frequent?', math.isclose(f1,mx), '(', format(f1*100.0, '.1f'),'
↪% )')
print(b2, 'is the most frequent?', math.isclose(f2,mx), '(', format(f2*100.0, '.1f'),'
↪% )')
print(b3, 'is the most frequent?', math.isclose(f3,mx), '(', format(f3*100.0, '.1f'),'
↪% )')
print(b4, 'is the most frequent?', math.isclose(f4,mx), '(', format(f4*100.0, '.1f'),'
↪% )')
print(b5, 'is the most frequent?', math.isclose(f5,mx), '(', format(f5*100.0, '.1f'),'
↪% )')
```

```
TooManyLikes is the most frequent? False ( 25.0 % )
Boombasticks is the most frequent? True ( 30.0 % )
Timewasters Inc is the most frequent? False ( 10.0 % )
Vanity 3.0 is the most frequent? False ( 5.0 % )
TrashTrend is the most frequent? True ( 30.0 % )
```

4.2.9 Decimal numbers

For most applications float numbers are sufficient, if you are conscius of their limits of representation and equality. If you really need more precision and/or preditability, Python offers a dedicated numeric type called `Decimal`, which allows arbitrary precision. To use it, you must first import `decimal` library:

[115]:
```
from decimal import Decimal
```

You can create a `Decimal` from a string:

[116]:
```
Decimal('4.1')
```

[116]:
```
Decimal('4.1')
```

WARNING: if you create a Decimal from a costant, use a string!

If you pass a `float` you risk losing the utility of Decimals:

```
[117]: Decimal(4.1)  # this way I keep the problems of floats ...
[117]: Decimal('4.09999999999999964472863211994990706443786621093750')
```

Operations between `Decimals` produce other `Decimals`:

```
[118]: Decimal('7.9') - Decimal('3.8')
[118]: Decimal('4.1')
```

This time, we can freely use the equality operator and obtain the same result:

```
[119]: Decimal('4.1') == Decimal('7.9') - Decimal('3.8')
[119]: True
```

Some mathematical functions are also supported, and often they behave more predictably (note we are **not** using `math.sqrt`):

```
[120]: Decimal('2').sqrt()
[120]: Decimal('1.4142135623730950488016887242')
```

Remember: computer memory is still finite!

Decimals can't be solve all problems in the universe: for example, $\sqrt{2}$ will never fit the memory of any computer! We can verify the limitations by squaring it:

```
[121]: Decimal('2').sqrt()**Decimal('2')
[121]: Decimal('1.9999999999999999999999999999')
```

The only thing we can have more with Decimals is more digits to represent numbers, which if we want we can increase at will[83] until we fill our pc memory. In this book we won't talk anymore about Decimals because typically they are meant only for specific applications, for example, if you need to perform fincancial calculations you will probably want very exact digits!

4.2.10 Challenges

We now propose some (very easy) exercises without solutions.

Try to execute them both in Jupyter and a text editor such as Spyder or Visual Studio Code to get familiar with both environments.

[83] https://docs.python.org/3/library/decimal.html

Challenge - which booleans 1?

Find the row with values such that the final print prints `True`. Is there only one combination or many?

[122]:
```
x = False; y = False
#x = False; y = True
#x = True; y = False
#x = True; y = True

print(x and y)
```

```
False
```

Challenge - which booleans 2?

Find the row in which by assigning values to `x` and `y` it prints `True`. Is there only one combinatin or many?

[123]:
```
x = False; y = False; z = False
#x = False; y = True; z = False
#x = True; y = False; z = False
#x = True; y = True; z = False
#x = False; y = False; z = True
#x = False; y = True; z = True
#x = True; y = False; z = True
#x = True; y =True; z =True

print((x or y) and (not x and z))
```

```
False
```

Challenge - Triangle area

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. Your code should show `Triangle area is: 120.0`

[124]:
```
# write here
```

Challenge - square area

Compute the area of a square having side (`s`) equal to 145 units. Assign the result to a variable named `area` and print it, it should show `Square area is: 21025`

[125]:
```
# write here
```

Challange - area from input

Modify the program at previous point. to acquire the side `s` from the user at runtime.

Hint: use the input[84] function and remember to convert the acquired value into an `int`). NOTE: from our experimentations, `input` tends to have problems in Jupyter so you'd better try in some other editor.

Try also to put the two previous scripts in two separate files (e.g. `triangle_area.py` and `square_area.py` and execute them from the terminal)

```
[126]: # write here
```

Challenge - trapezoid

Write a small script (`trapezoid.py`) that computes the area of a trapezoid having major base (`mj`) equal to 30 units, minor base (mn) equal to 12 and height (`h`) equal to 17. Print the resulting area. Try executing the script from a text editor like Spyder or Visual Studio Code and from the terminal.

It should print `Trapezoid area is: 357.0`

```
[127]: # write here
```

Challenge - first n numbers

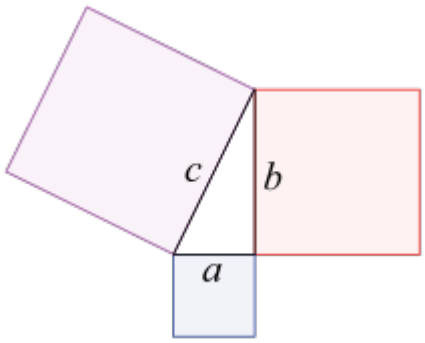
- Rewrite the example of the sum of the first 1200 integers by using the following equation: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Then modify the program to make it acquire the number of integers to sum N from the user at runtime

It should show `Sum of first 1200 integers: 720600.0`

```
[128]: # write here
```

challenge - hypotenuse

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). It should print `Hypotenuse: 151.23822268196622`



[84] https://www.geeksforgeeks.org/taking-input-in-python/

[129]:
```
# write here
```

Challenge - which integers 1?

Assign numerical values to `x` `y` e `z` to have the expression print `True`

[130]:
```
x = 0  # ?
y = 0  # ?
print(max(min(x,y), x + 20) > 20)
```

```
False
```

Challenge - which integers 2?

Assign to `x` and `y` values such that `True` is printed

[131]:
```
x = 0  # ?
y = 0  # ?

print(x > 10 and x < 23 and ((x+y) == 16 or (x + y > 20)))
```

```
False
```

Challenge - which integers 3?

Assign to `z` and `w` values such that `True` is printed.

[132]:
```
z = 0 # ?
w = 1 # ?
(z < 40 or w < 90) and (z % w > 2)
```

[132]:
```
False
```

Challenge - airport

You finally decide to take a vacation and you go to the airport, but you already know you will need to go through various queues. Luckily, you only have carry-on bag, so you directly go to security checks, where you can choose among three rows of people `sec1`, `sec2`, `sec3`. Each person an average takes 4 minutes to be examinated, you included, and obviously you choose the shortest queue. Afterwards you go to the gate, where you find two queues of `ga1` and `ga2` people, and you know that each person you included an average takes 20 seconds to pass: again you choose the shortes queue. Luckily the aircraft is next to the gate so you can directly choose whether to board at the queue at the head of the aircraft with `bo1` people or at the queue at the tail of the plane with `bo2` people. Each passenger you included takes an average 30 seconds, and you choose the shortest queue.

Write some code to calculate how much time you take in total to enter the plane, showing it in minutes and seconds.

Example - given:

```
sec1,sec2,sec3,ga1,ga2,bo1,bo2 = 4,5,8,5,2,7,6
```

your code must print:

```
24 minutes e 30 seconds
```

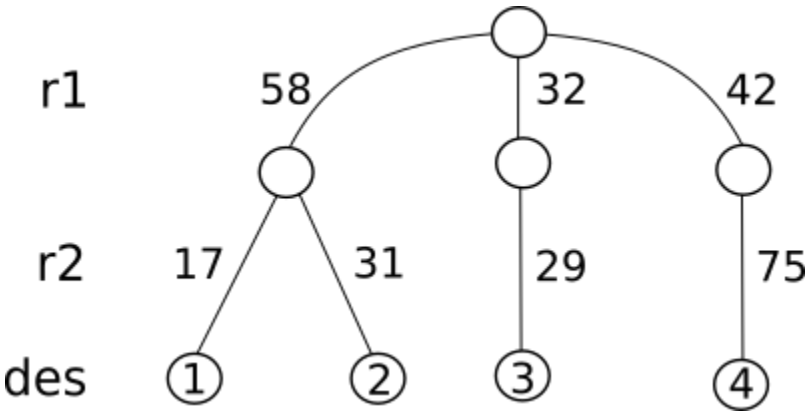
[133]:
```
sec1,sec2,sec3,ga1,ga2,bo1,bo2 = 4,5,8,5,2,7,6    # 24 minutes e 30 seconds
#sec1,sec2,sec3,ga1,ga2,bo1,bo2 = 9,7,1,3,5,2,9  # 10 minutes e 50 seconds

# write here
```

Challenge - Holiday trip

While an holiday you are traveling by car, and in a particular day you want to visit one among 4 destinations. Each location requires to go through two roads `r1` and `r2`. Roads are numbered with two digits numbers, for example to reach destination `1` you need to go to road `58` and road `17`.

Write some code that given `r1` and `r2` roads shows the number of the destination.

- If the car goes to a road it shouldn't (i.e. road 666), put `False` in `destination`
- **DO NOT** use summations
- **IMPORTANT: DO NOT use** `if` **commands** (it's possible, think about it ;-)



Example 1 - given:

```
r1,r2 = 58,31
```

After your code it must print:

```
The destination is 2
```

Example 2 - given:

```
r1,r2 = 666,31
```

After your code it must print:

```
The destination is False
```

[134]:
```
r1,r2 = 58,17   # 1
r1,r2 = 58,31   # 2
r1,r2 = 32,29   # 3
r1,r2 = 42,75   # 4
```

(continues on next page)

(continued from previous page)

```
r1,r2 = 666,31  # False
r1,r2 = 58,666  # False
r1,r2 = 32,999  # False

# write here
```

4.2.11 References

- Think Python - Chapter 1[85]: The way of the program
- Think Python - Chapter 2[86]: Variables, expressions and statements

[ ]:

4.3 Commandments

The Supreme Committee for the Doctrine of Coding has ruled important Commandments you shall follow.

If you accept their wise words, you shall become a true Python Jedi.

WARNING: if you don't follow the Commandments, you will end up in *Debugging Hell* !

4.3.1 I COMANDAMENT

You shall write Python code

Who does not writes Python code, does not learn Python

4.3.2 II COMANDAMENT

Whenever you insert a variable in a `for` cycle, such variables must be new

If you defined the variable before, you shall not reintroduce it in a `for`, because doing so might bring confusion in the minds of the readers.

So avoid such sins:

[1]:
```
i = 7
for i in range(3):  # sin, you lose variable i
    print(i)

print(i)    # prints 2 and not 7  !!
```

[85] http://greenteapress.com/thinkpython2/html/thinkpython2002.html
[86] http://greenteapress.com/thinkpython2/html/thinkpython2003.html

```
0
1
2
2
```

[2]:
```
def f(i):
    for i in range(3): # sin, you lose parameter i
        print(i)

    print(i)  # prints 2, not the 7 we passed!

f(7)
```

```
0
1
2
2
```

[3]:
```
for i in range(2):

    for i in  range(5):  # debugging hell, you lose the i of external cycle
        print(i)

    print(i)  # prints 4 !!
```

```
0
1
2
3
4
4
0
1
2
3
4
4
```

4.3.3 III COMANDAMENT

You shall never reassign function parameters

You shall never ever perform any of these assignments, as you risk losing the parameter passed during function call:

[4]:
```
def sin(my_int):
    my_int = 666            # you lost the 5 passed from external call!
    print(my_int)           # prints 666

x = 5
sin(x)
```

```
666
```

Same reasoning can be applied to all other types:

[5]:
```
def evil(my_string):
    my_string = "666"
```

[6]:
```
def disgrace(my_list):
    my_list = [666]
```

[7]:
```
def delirium(my_dict):
    my_dict = {"evil":666}
```

For the sole case when you have composite parameters like lists or dictionaries, you can write like below IF AND ONLY IF the function description requires to MODIFY the internal elements of the parameter (like for example sorting a list in-place or changing the field of a dictionary.

[8]:
```
# MODIFY my_list in some way
def allowed(my_list):
    my_list[2] = 9

outside = [8,5,7]
allowed(outside)
print(outside)
```

```
[8, 5, 9]
```

On the other hand, if the function requires to RETURN a NEW object, you shall not fall into the temptation of modifying the input:

[9]:
```

# RETURN a NEW sorted list
def pain(my_list):
    my_list.sort()     # BAD, you are modifying the input list instead of creating a␣
↪new one!
    return my_list
```

[10]:
```
# RETURN a NEW list
def crisis(my_list):
    my_list[0] = 5           # BAD, as above
    return my_list
```

[11]:
```
# RETURN a NEW dictionary
def tormento(my_dict):
    my_dict['a'] = 6  # BAD, you are modifying the input dictionary instead of␣
↪creating a new one!
    return my_dict
```

[12]:
```
# RETURN a NEW class instance
def desperation(my_instance):
    my_instance.my_field = 6  # BAD, you are modifying the input object
                              #      instead of creating a new one!
    return istanza_di_classe
```

4.3.4 IV COMANDAMENT

You shall never ever reassign values to function calls or mmethods

WRONG:

```
mia_funzione() = 666
mia_funzione() = 'evil'
mia_funzione() = [666]
```

CORRECT:

```
x = 5
y = my_fun()
z = []
z[0] = 7
d = dict()
d["a"] = 6
```

Function calls like `my_function()` return calculations results and store them in a box in memory which is only created for the purposes of the call, and Python will not allow us to reuse it like it were a variabile.

Whenever you see `name()` in the left part, it *cannot* be followed by the equality sign = (but it can be followed by due equals sign == if you are doing a comparison).

4.3.5 V COMMANDMENT

You shall never ever redefine system functions

Python has several system defined functions. For example `list` is a Python type: as such, you can use it for example as a function to convert some type to a list:

```
[13]: list("ciao")
```

```
[13]: ['c', 'i', 'a', 'o']
```

When you allow the Forces of Evil to take the best of you, you might be tempted to use reserved words like `list` as a variable for you own miserable purposes:

```
[14]: list = ['my', 'pitiful', 'list']
```

Python allows you to do so, but we do **not**, for the consequences are disastrous.

For example, if you now attempt to use `list` for its intended purpose like casting to list, it won't work anymore:

```
list("ciao")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-c63add832213> in <module>()
----> 1 list("ciao")

TypeError: 'list' object is not callable
```

In particular, we recommend to **not redefine** these precious functions:

- `bool`, `int`,`float`,`tuple`,`str`,`list`,`set`,`dict`
- `max`, `min`, `sum`
- `next`, `iter`
- `id`, `dir`, `vars`,`help`

4.3.6 VI COMMANDMENT

You shall use `return` command only if you see written RETURN in function description!

If there is no `return` in function description, the function is intended to return `None`. In this case you don't even need to write `return None`, as Python will do it implicitly for you.

4.3.7 VII COMMANDMENT

You shall also write on paper!

If staring at the monitor doesn't work, help yourself and draw a representation of the state sof the program. Tables, nodes, arrows, all can help figuring out a solution for the problem.

4.3.8 VIII COMANDAMENT

You shall never ever reassing `self` !

You shall never write horror such as this:

[15]:
```
class MyClass:
    def my_method(self):
        self = {'my_field':666}     # SIN
```

Since `self` is a kind of a dictionary, you might be tempted to write like above, but to external world it will bring no effect.

For example, let's suppose somebody from outside makes a call like this:

[16]:
```
mc = MyClass()
mc.my_method()
```

After the call `mc` will not point to `{'my_field':666}`

[17]:
```
mc
```

[17]:
```
<__main__.MyClass at 0x7f547c35bf28>
```

and will not have `my_field`:

mc.my_field

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-5c4e6630908d> in <module>()
----> 1 mc.male

AttributeError: 'MyClass' object has no attribute 'my_field'
```

For the same reasoning, you shall never reassign `self` to lists or others things:

[18]:
```
class MyClass:
    def my_method(self):
        self = ['evil']      # YET ANOTHER SIN
        self = 666           # NO NO NO
```

4.3.9 IX COMMANDMENT

You shall test!

Untested code by definition *does not work*. For ideas on how to test it, have a look at Errors and testing[87]

4.3.10 X COMMANDMENT

You shall never ever add or remove elements from a sequence you are iterating with a `for` !

Falling into such temptations **would produce totally unpredictable behaviours** (do you know the expression *pulling the rug out from under your feet ?*)

Do not add, because you risk to walk on a tapis roulant that never turns off:

```
my_list = ['a','b','c','d','e']
for el in my_list:
    my_list.append(el)   # YOU ARE CLOGGING COMPUTER MEMORY
```

Do not remove, because you risk to corrupt the natural order of things:

[19]:
```
my_list = ['a','b','c','d','e']

for el in my_list:
    my_list.remove(el)    # VERY BAD IDEA
```

Look at the code. You think we removed eveything, uh?

[20]:
```
my_list
```

[20]:
```
['b', 'd']
```

`O_o'` Do not even try to make sense of such sorcery - nobody can, because it is related to internal implmentation of Python.

My version of Python gives this absurd result, yours may give another. Same applies for iteration on sets and dictionaries. **You are warned**.

If you really need to remove stuff from the sequence you are iterating on, use a while cycle[88] or make first a copy of the original sequence.

[87] https://en.softpython.org/errors-and-testing/errors-and-testing-sol.ipynb
[88] https://en.softpython.org/control-flow/flow3-while-sol.html

CHAPTER

FIVE

CHAPTER

SIX

INDEX